# PixOrNet+OpNet: Superpixels Learning for Fast Volume Visualization

**Abstract**—We propose a novel approach for ray classification utilizing a Deep Neural Network architecture consisting of two small neural networks, which we term PixOrNet+OpNet. This network is designed to extract a latent space representation of rays, which can be used for ray clustering. Our approach accelerates volume rendering by 10 to 12-fold by only performing ray casting on critical points while producing higher image quality when compared to superpixel segmentation. In addition to enhancing rendering performance, the latent space generated by PixOrNet proves advantageous for compressing 2D images more efficiently, often rivals results by JPEG while being lossless, and improving the performance of applications that rely on neural network-based reconstruction, such as neural network-based super-resolution techniques.

**Index Terms**—Volume Rendering Acceleration, Ray classification, Deep Learning based.

## 1 Introduction

Volume rendering is a crucial visualization technique widely employed in fields such as medical imaging, scientific computing, and engineering. In medical imaging, it allows clinicians to visualize complex anatomical structures from CT or MRI scans, facilitating accurate diagnosis and surgical planning. In scientific computing, volume rendering helps researchers explore and analyze volumetric datasets generated by simulations of physical phenomena, such as fluid dynamics, weather patterns, and astrophysical processes. In engineering, it is used to inspect and validate the integrity of components in non-destructive testing and to visualize the internal features of materials and assemblies in CAD models. By providing a means to convert intricate three-dimensional (3D) data into interpretable two-dimensional (2D) images, volume rendering serves as an indispensable tool for visual analysis and decision-making across these diverse fields.

At its core, volume rendering has two important components: 3D volume and transfer function (TF). A 3D volume represents a volumetric data set, traditionally structured as a grid of voxels (i.e., volumetric pixels) or recently represented by Neural Network [21]. Each voxel contains data corresponding to specific attributes, such as density, intensity, or other scalar values, derived from various sources like medical scans (e.g., CT or MRI), scientific simulations, or industrial imaging techniques. This 3D volume serves as the foundation for volume rendering, providing the raw data that needs to be visualized.

On the other hand, TFs play a crucial role in transforming raw voxel data into visual information. A TF maps the scalar values of a 3D volume to optical properties such as color and opacity. By assigning different colors and opacity levels to specific value ranges within a data set, TFs enable the visualization of distinct structures and features within the volume. For instance, in medical imaging, TFs can be tailored to highlight tissues, such as bones, tissues, or blood vessels, by assigning appropriate colors and opacities to the corresponding intensity ranges. The effectiveness of volume rendering heavily depends on the TF design, as it determines how well the internal structures of a volume are visualized and differentiated. Therefore, during data exploration, users will often experiment with various TFs to figure out which one works best for their 3D volumes. However, because any modifications in TFs can lead to new rendering outcomes, users may end up spending significant time on rendering rather than focusing on understanding the 3D volumes. Although approaches are proposed to provide meaningful and guided TF designs [17, 18], they often lack interaction with users for task-specific explorations. This prompts us to explore ways that allow users to quickly compute the new image maps whenever the TF changes without going through the ray-casting process again for all rays.

One approach for addressing this problem is through ray classification. By clustering rays into groups, we can reduce rendering time by avoiding the re-computation of similar rays. The challenge lies in determining an effective method for clustering rays. Relying solely on raw volumetric values is unlikely to produce useful results, as it does not account for the light-obstruction properties of each sample. Conversely, performing ray clustering after rendering may result in inaccurate clusters because rays that appear similar under one TF may differ significantly under another. This challenge motivated us to develop a novel representation of rays. Such representation can be used as a clustering signal to cluster rays that produce similar pixels under most, if not all, color and opacity TFs.

In this work, we created a deep-neural network-based renderer, namely PixOrNet+OpNet, which contains two components: the PixOrNet encodes a representation of a ray, and then OpNet takes that representation and sampled alpha values on a ray to compute the final pixel. If two rays share a similar representation computed by PixOrNet, they will end up with similar final pixels regardless of the TF. Therefore, the ray clustering problem becomes a standard clustering problem on PixOrNet's representation of rays. This new problem can be solved through a myriad of clustering techniques. In this work, we used a k-means clustering algorithm and compared our results against Superpixel Linear Iterative Clustering (SLIC), a popular superpixel segmentation method. By computing only 8.5% of rays, our approach can produce an image with PSNR around 30 with up to 40 PSNR for a small dataset and can be up to 13 PSNR points higher compared to images by SLIC. Our contributions are as follows:

- PixOrNet+OpNet as an alternative way to render rays without ray casting
- A novel ray-clustering technique utilizing PixOrNet to consider the light-obstruction properties of samples.
- Applications of PixOrNet such as accelerating rendering pipelines, enhancing image compression, and speeding up neural network-based image synthesis processes.

## 2 Related works

### 2.1 Volume Rendering

Volume rendering involves creating a 2D projection of a 3D data set. This process is essential for visualizing complex structures within the data, such as tissues in medical scans or phenomena in fluid simulations. Traditional volume rendering methods employ TFs to map data values

*Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx*

to optical properties such as color and opacity values so that researchers can extract meaningful visual information from volumetric data. One of the fundamental challenges in volume rendering is achieving a balance between computational efficiency and image quality, as high-fidelity renderings can be computationally expensive.

Ray casting, a fundamental technique in volume rendering, casts rays from a viewpoint through a volume data set to construct the final image. Each ray accumulates color and opacity values as it traverses the data, integrating them to produce the image pixel. This method provides high-quality visualizations and can capture intricate details within the volume. Over the years, various optimizations have been proposed to enhance the performance of ray casting, such as early ray termination and space leaping. Additionally, advancements in GPU computing have significantly accelerated ray casting processes, making real-time volume rendering feasible [3].

A recent development in volume rendering is using Neural Networks to represent 3D volumes. Among many innovations, Mildenhall et al. [21] introduced Neural Radiance Fields (NeRF), a groundbreaking technique for 3D scene representation and rendering. NeRF uses a deep neural network to model a continuous volumetric scene by training on a limited set of 2D images. Through this, NeRF learns to predict color and density at any point in 3D, enabling the synthesis of photorealistic views from new perspectives. This approach marked a significant departure from traditional volumetric rendering methods, which often relied on explicit geometric representations and were limited in handling complex lighting and view-dependent effects.

The success of NeRF has spurred a variety of subsequent research efforts aimed at improving and extending its capabilities. For instance, several studies have focused on enhancing the efficiency of NeRF. For example, FastNeRF [7] and PlenOctrees [30] accelerate rendering for real-time applications. Other works, such as RecolorNERF [8] and PaletteNeRF [15], allow users to change the colors of models.

Similar to NeRF, we also used a feed-forward neural network to predict visualization results. However, unlike NeRF, which predicts the color and alpha of each sample, our PixOrNet+OpNet approach predicts the contributing alpha of each color in a color palette. We will elaborate more on this in Sec. 3.

### 2.2 Superpixel Segmentation

Superpixel segmentation is a popular technique in computer vision that groups pixels in an image into perceptually meaningful atomic regions, known as superpixels. Unlike traditional pixel-based processing, where each pixel is treated individually, superpixel segmentation reduces the complexity of image data by clustering pixels into contiguous regions based on similarity in color, texture, and spatial proximity. These superpixels often follow natural boundaries within the image, allowing for a more structured representation of visual information. The segmentation process facilitates various image processing tasks such as object recognition, image segmentation, and edge detection by providing a more manageable set of regions for further analysis.

Many techniques for superpixel segmentation have been explored in literature and could be summarized into four main categories: Watershed-based, Graph-based, Clustering-based, and Energy optimization, described in [16]. One notable approach is Superpixel Linear Iterative Clustering (SLIC). SLIC uses a k-means clustering algorithm where the distance between two pixels is a combination of their grey space distance and spatial distance. SLIC first initializes various centroids through the image before iteratively refined by assigning nearby pixels to the clusters that minimize the distance. The process is repeated until the error converges. SLIC is often employed in various computer vision tasks because of its simplicity and linear complexity, especially when the number of superpixels is high [26]. Chen et al. [5] introduced Linear Spectral Clustering (LSC), which maps the image to a ten-dimensional feature vector. Then, seed points are uniformly selected across the image, and their feature vectors are the clusters' initial weighted means. Each pixel is then assigned to a cluster whose weight mean is closest to its feature vector. The process is repeated until the clusters stabilize. Compared to SLIC, LSC preserves the image's global properties and produces more regular pixels but has slightly worse runtime [16, 26]. In terms of performance, LSC often performs better than SLIC in many segmentation quality metrics. However, similar to other clustering-based approaches, LSC and SLIC do not perform well in terms of segmentation accuracy compared to other graph-based approaches. Nevertheless, they remained popular options because of their linear complexity [26].

While superpixel segmentation algorithms produce impressive results and can considerably accelerate the rendering process, they often only consider a 2D image generated from a specific TF. Thus, when users change the color or opacity of a TF, the clusters computed previously may not transfer well to a new image, leading to artifacts. Our approach performs clustering by leveraging a novel representation of rays' samples that is independent of TFs. Such representation allows us to more accurately capture which rays, and subsequently the pixels, are likely to look similar across different TFs.

### 2.3 Image Synthesis

Recently, deep learning-based image synthesis [19] has gained prominence in improving the rendering performance of volumetric datasets across both spatial and temporal domains. Berger et al. [2] use a generative adversarial network (GAN) framework to generate visualization images from view parameters and TFs directly. He et al. [12] design a convolutional regression model to learn the mapping from the simulation and visualization parameters to the visualization results. Weiss et al. [27] provides a framework to render a high-resolution volumetric isosurface rendering from a low-resolution rendering through deep learning-based super-resolution. Han et al. [10, 11] extend the approach to address spatiotemporal super-resolution volumes. Kaplanyan et al. [13] and Bauer et al. [1] utilize reconstruction neural networks to speed up the rendering by inferencing the missing detail from sparsely rendered images with Foveated pattern through inpainting. Renderers utilizing reconstruction neural networks also give state-of-the-art rendering latency in volume visualization. However, there is no existing work on optimizing the generation of partially rendered images, which the reconstruction neural networks rely on. Our approach provides a solution to speed up the generation of partially rendered images through the clustering result given by PixOrNet.

## 3 METHOD

In this section, we first introduce the concept of color groups as an alternative method for representing samples on a ray and demonstrate how it enables fast re-rendering of the image when only the color TF is changed. Next, we introduce PixOrNet+OpNet and show how it facilitates quick re-rendering when the opacity TF is changed. Specifically, PixOrNet generates a representation of a ray, and then OpNet uses that representation and alpha values assigned to samples on the ray to compute the final pixel. A notable advantage of PixOrNet is that if two rays share a similar representation computed by PixOrNet, their final pixels will be similar regardless of the TF used. After that, we describe how PixOrNet's latent space can be used to cluster rays, compress images more efficiently, and accelerate low-resolution image generation for other neural network image synthesis.

### 3.1 Color Groups

We begin by introducing the concept of *color groups* and demonstrate its application to ray casting.

Consider a ray consisting of a set of discrete samples, each assigned with a pair of RGB color $C_i$ and opacity $\alpha_i$ values, denoted as $\{(C_0, \alpha_0), \cdots, (C_i, \alpha_i), \cdots, (C_n, \alpha_n)\}$, where $C_i \in [0,1]^3$ and $\alpha_i \in [0,1]$. Based on the volume rendering integral [20], the closed-form formula for the final color $C_{final}$ obtained by compositing along each ray is

$$C_{final} = \sum_{i=1}^{n} C_i \alpha_i \prod_{j=0}^{i-1} (1-\alpha_j) \tag{1}$$

In this work, we categorize samples of rays into color groups of a palette, similar to palette-based recoloring methods [4, 24]. The number of color groups is denoted by $m$. All samples within a color group share the same color or RGB values, though their alpha values may vary. Typically, each color group generalizes a range of volumetric values.

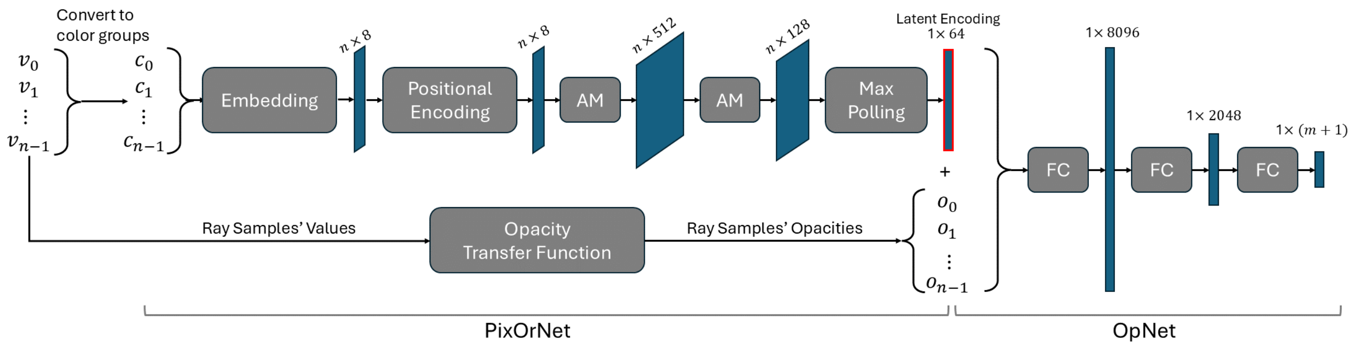


Fig. 1: Architecture of PixOrNet + OpNet for $m$ color groups and $n$ max samples per ray. PixOrNet is responsible for encoding the samples of rays into a latent representation. This encoded representation, along with the alpha values of the ray samples, serves as input to OpNet, which then computes the final pixel results of the rays. The representation generated by PixOrNet is subsequently utilized as a clustering signal. The principle behind this is that rays with similar representations are likely to produce similar pixel results. Thus, rays that share comparable latent space representations are grouped, facilitating faster volume rendering and space-efficient lossless image compression.

In real-world applications, users may find it useful to assign one or more color groups to represent specific objects, such as muscle, skin, or bone, enabling them to adjust volume rendering results to highlight or exclude particular objects. For example, in Gaussian TFs [14], each Gaussian typically corresponds to a distinct object. Users can assign a single color group to a Gaussian to represent the object as a cohesive entity, or they can assign multiple color groups to a Gaussian for enhanced control over the different regions of the object.

Once a palette is constructed, ray samples sharing the same color are grouped together, allowing the color components in Eq. (1) to be factored. For example, consider the following samples along a ray, $\{([1,0,0],\alpha_0),([0,1,0],\alpha_1),([0,0,1],\alpha_2),([1,0,0],\alpha_3),([0,1,0],\alpha_4)\}$. According to Eq. (1), the resulting color can be expressed as:

$$\begin{aligned} C_{final} &= [1,0,0]\times(\alpha_0\cdot 1+\alpha_3(1-\alpha_0)(1-\alpha_1)(1-\alpha_2)) \\ &\quad +[0,1,0]\times(\alpha_1(1-\alpha_0)+\alpha_4(1-\alpha_0)(1-\alpha_1)(1-\alpha_2)(1-\alpha_3)) \\ &\quad +[0,0,1]\times(\alpha_2(1-\alpha_0)(1-\alpha_1)) \end{aligned} \tag{2}$$

In this format, the RGB values of each color group are multiplied by the accumulated alpha values of all samples assigned to that group, which we refer to as the alpha contribution of the color group. For example, if the color of the first group changes from $[1,0,0]$ to $[1,1,0]$, the final color can be recalculated without needing to recompute the entire ray. However, if the alpha value of a sample changes, updating $C_{final}$ becomes more complex because such a change to a sample affects all subsequent samples behind it. To address this, we employ a deep neural network OpNet trained to predict the alpha contribution of each color group.

### 3.2 PixOrNet

The first neural network is PixOrNet, short for Pixel-Order-Network, which encodes samples along a ray. Although it would be more accurate to refer to each sample as a voxel, making the name Voxel-Order-Network more fitting, PixOrNet was chosen for its natural sound and the legacy of its origin. PixOrNet takes a sequence as input, where each element within the array denotes the color group to which the corresponding sample along a ray belongs. We call such a sequence a *ray pattern*. In Fig. 1, this array corresponds to $[c_0,c_1,\cdots,c_{n-1}]$ where $c_i$ denotes a color group ID if $c_i>0$ or no sample if $c_i=0$. The output of PixOrNet is a 64-element array that represents the encoded rays. In the embedding layer, the number of vocab is $m+1$, and the size of each vocab is 8. While a one-hot encoding can be used, we found it to perform inconsistently, so we opted for a learnable layer instead. Additionally, because not all samples along a ray are equal–the latter samples are often occluded by those in front–a positional encoding layer is introduced to differentiate samples sharing the same color group. Following the embedding and positional encoding layer are three 2D affine-mapping (AM) layers. The output then passes through a 1D max-pooling layer and is transposed to create the latent space of each ray, which has the dimensions of $(1,64)$. All layers use the ReLU activation function, except for the final layer, which uses the sigmoid activation function. A visual depiction of PixOrNet is shown in Fig. 1.

### 3.3 OpNet

The second neural network is OpNet, short for Opacity-Network. This network is responsible for computing the opacity contribution of each color group. OpNet takes as input a sequence of samples' opacities (i.e., alpha values) along a ray, which has the same size as the input to PixOrNet, along with the pattern encoding (i.e., the output of PixOrNet). In Fig. 1, the opacities is illustrated as $[o_0,o_1,\cdots,o_{n-1}]$. The input is processed through three fully connected (FC) layers. OpNet's output layer has $m+1$ neurons, where each output neuron represents the opacity contribution of each color group, except for the last one, which measures the total alpha accumulation $\alpha_{acc}$ of the ray or the final alpha of the pixel (more accurately, the neuron value is $1-\alpha_{acc}$). All layers use the ReLU activation function, except for the last layer, which uses the sigmoid activation function. A visual representation of OpNet is shown Fig. 1. Because OpNet is not used in the rendering pipeline, it is possible to increase the number of neurons to reduce loss even further.

### 3.4 PixOrNet+OpNet

PixOrNet and OpNet together form a viable rendering system. PixOrNet encodes a novel representation of rays, while OpNet decodes that representation using the information from user-specified TFs to produce RGBA values of a ray. Therefore, if two rays share similar representations after being processed by PixOrNet, they are likely to produce similar RGBA value when passing through OpNet.

Next, we discuss how PixOrNet+OpNet can be trained to be *view-independent* and *TF-independent*. In the model's architecture, the only input to PixOrNet is a ray pattern, which is simply an array of numbers between 0 and $m$, where $m$ is the number of color groups. A brute-force method would generate all possible permutations of such arrays for the model to learn. However, for a 3D volume, not all permutations are feasible. Instead, a more practical approach is to collect ray pattern permutations by randomly sampling rays across multiple views. With this approach, a trained model PixOrNet+OpNet can be applied to any view of the 3D volume. However, any changes that alter the ray pattern, such as adjusting the sampling rate, will require retraining the model. Additionally, the opacities of the ray samples, which serve as the input for OpNet, are randomly generated between 0 and 1 to account for the varying values these samples might take based on chosen opacity TFs. As a result, the entire network becomes both view-independent and TF-independent.

Lastly, we address two key hyper-parameters in the model training process: the number of test cases and the unique ray patterns used for training. The number of test cases refers to the total size of the training, validation, and testing datasets. The number of unique patterns refers to the number of permutations sampled from multiple views, as discussed earlier. These parameters differ because the same ray pattern may appear in multiple test cases with different alpha values, allowing the model to learn how a ray behaves under various TFs. Details on how test cases are generated will be elaborated in Sec. 4.1. We have conducted some experiments to explore the impact of these parameters on model training time and image quality.

## 3.5 Application/Use Case

### 3.5.1 Ray classification

The first application of our method is ray classification. We utilize PixOrNet's signals as metrics for k-means clustering to classify rays. By grouping rays into clusters, we only need to compute the opacity contribution for one representative ray per cluster group. Specifically, as shown in Fig. 1, each sampled ray is initially processed through PixOrNet to obtain its latent encoded features. K-means clustering is then applied, where the distance between two rays is determined by the mean-squared differences in their encoded features. The rationale behind using encoded features is that if two rays have similar encoded features, their outputs after processing through OpNet will also be similar.

Another advantage of this strategy is its view-independence. Similar to how we design PixOrNet+OpNet to be view-independent, because clustering is based on encoded features, it is possible to initially sample numerous views around the 3D volume to establish a set of comprehensive clusters. The decision is up to the users to balance the trade-off between upfront and per-view computational costs.

We acknowledge that incorporating spatial features alongside our encoded features in the clustering process produces similar results. However, this would make the clustering view-dependent, requiring re-clustering whenever the view changes. We aim to classify rays based on properties specific to each ray, such as its encoded features.

While ray classification was initially developed in the context of OpNet, it is not restricted to this framework. In our rendering process, we apply traditional direct volume rendering (DVR) to the centroids of all clusters instead of using OpNet, as traditional DVR is significantly more time-efficient for each ray. Compared to traditional methods that use volumetric values or final pixels of 2D-generated images as classification criteria, our approach considers the contribution of each sample to the final pixel. Moreover, our method can distinguish rays that may appear similar under the current TF but differ significantly under another TF.

The last knot left is how alpha accumulation (or final pixel alpha) changes as the TF changes. In k-means clustering, it is possible to use the alpha accumulation of the centroids for all other rays within the clusters. However, this often results in a drop in performance. Thus, we chose to use a simple approximation of the alpha accumulation. Given

$$1-\alpha_{\text{acc}} = \prod_{i=1}^{n}(1-\alpha_i) \tag{3}$$

where $\alpha_i$ are the samples on the ray, one simple method is to leverage the color groups we have. Since each sample belongs to a color group, we can approximate the alpha accumulation as:

$$1-\alpha_{\text{acc}} \approx (1-\alpha_{c_1})^{n_1}\cdot(1-\alpha_{c_2})^{n_2}\cdots(1-\alpha_{c_m})^{n_m} \tag{4}$$

where $\alpha_{c_i}$ and $n_i$ are the average alpha values of color group $i$ and the number of samples with color group $i$ on the ray, respectively. For a TF, this can be the average of the maximum and minimum alpha values of a color group. With this method, whenever the TF changes, we update the values for $\alpha_{c_i}$ and recalculate the new alpha accumulation. The method leverages fast exponentiation to compute the alpha accumulation more efficiently. The time complexity is $O(m\cdot\log(n/m))$, where $m$ and $n$ are the number of color groups and the number of samples on the ray, respectively. The worst-case time complexity occurs when the number of samples is evenly distributed across all $m$ color groups. In practice, the runtime is often much faster because of the memoization of exponentiation results. Additionally, early ray termination can be implemented, which, as shown in Sec. 5, can significantly improve the efficiency of alpha accumulation updates for rays with a large number of samples.

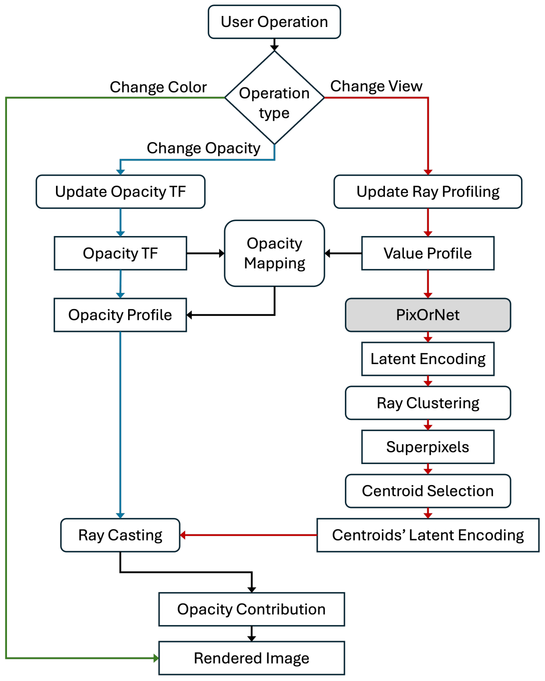


Fig. 2: Flow chart for rendering pipeline.

**Algorithm 1** PixOrNet Renderer

**Input:** Rays, new TF, and Kmeans

**Output:** RGBA values of each ray.

```
 1: for $i \leftarrow 0$ to $len(kmeans.centroids)$ do
 2:     $sample = kmeans.centroids[i].sample$
 3:     $kmeans.centroids[i].rgba = rayCasting(sample, newTF)$
 4: end for
 5: for $i \leftarrow 0$ to $len(rays)$ do
 6:     $ray = rays[i]$
 7:     if $ray.encodingFeatures$ is None then
 8:         $ray.encodingFeatures = PixOrNet(ray.samples)$
 9:         $ray.centroid = minDist(kmeans, ray.encodingFeatures)$
10:     end if
11:     $rays[i].rgba = kmeans.centroids[ray.centroid].rgba$
12:     $rays[i].alpAcc = updateAlpAcc(ray.alpAcc, newTF)$
13: end for
```

Fig. 2 illustrates the step-by-step pipeline of our rendering process, and Algorithm 1 outlines the corresponding high-level pseudocode.

### 3.5.2 Image compression

Another application of PixOrNet is image compression, which naturally extends from ray classification. Instead of storing four components (RGBA) as floating-point values or 8-bit integers for each pixel, we replace the RGB values with the identifier of the cluster to which each pixel belongs. For regions consisting of white space or background, this identifier is set to 0. This produces a 2D array containing only cluster identifiers. When this 2D array is flattened into a 1D array, the resulting array comprises integers ranging from 0 to the total number of clusters. This allows for the use of various standard compression algorithms, such as Run-Length Encoding (RLE), Huffman Encoding, or Lempel-Ziv-Welch (LZW), to further compress the array.

In this work, we employ a lossless compression scheme to demonstrate our results. The image is first compressed using Huffman Encoding, followed by RLE. The choice of RLE after Huffman Encoding is due to the significant amount of empty space or background in the images, which leads to inefficiencies if not addressed. In the RLE process, patterns of length 1 do not have their counts appended to the compressed data to avoid unnecessary space usage. Additionally, we perform simple pattern replacement, where certain substrings—specifically, a series of patterns of length 1 from the RLE output—are substituted with a single code. The procedure is similar to that of the JPEG compression scheme without the Wavelet Cosine. However, unlike JPEG, our compression is lossless.

In summary, the compression process consists of two primary steps: first, converting RGB values to cluster identifiers, and second, applying the aforementioned compression scheme independently to the cluster identifier array and the alpha component. Compared to traditional image compression methods, our approach is particularly suited for images generated from 3D datasets, leveraging information from the 3D space to achieve a higher compression rate.

#### 3.5.3 Super-Resolution

As for the third application of PixOrNet, we use ray classification to speed up the generation of the low-resolution image, which will be used as input to the super-resolution neural network. Recent deep learning-based super-resolution neural networks are capable of speeding up the generation of high-res visualization images with high fidelity by interpolating missing pixels from low-res input rendering images [9, 11, 27]. However, the pattern of the low-res input is predefined from an evenly distributed sampling pattern. The superpixels learned by our method can help speed up the generation of this low-res image by only rendering the centroid of each cluster covered by the sampling pattern.

## 4 Experiments

### 4.1 Datasets

We used the Tooth, Vertebra, Miranda, and Richtmyer datasets. Details about the number of rays per view and samples per ray for each dataset are outlined in Tab. 1. All volumes are rescaled to the range of $[-1,1]$ for the x, y, and z dimensions. We select $512\times512$ as the image resolution for our experiments. Because the goal is to allow users to vary the color and opacity TF, we will exclude background rays in our runtime computation. Background rays refer to rays that only go through the empty space before hitting a background or exiting the 3D volume. These rays' color results often do not change when the users change the TF so that they can be memorized and reused.

Regarding how we generated our training, validation, and testing dataset, we first select the number of color groups, denoted as $m$, for the model to train on. For this experiment, $m$ is chosen to be 9. As detailed in Sec. 3.4, we randomly sample the model from multiple viewpoints to collect a set of unique ray patterns. The number of collected unique rays for each dataset is listed in Tab. 1. If a ray pattern is smaller than the maximum possible number of samples, we pad the remaining elements of the array with zeros.

Besides the experiment to study the impact of different numbers of test cases, for most datasets, we generated approximately 3 million test cases total, split among 2 million test cases for the training dataset, 500,000 test cases for the validation dataset, and 500,000 test cases for the testing dataset. Miranda dataset is the exception with 4, 1, and 1 million test cases for training, validation, and testing datasets, respectively. Because the number of unique rays of the Miranda dataset is more than double that of other datasets, the number of test cases for Miranda is also doubled accordingly. A test case consists of 1) a ray pattern randomly selected among a pool of unique ray patterns and 2) an array of randomly generated alpha values. Each alpha value represents the opacity of the corresponding sample in the ray pattern. The procedure for generating the alpha values is as follows:

- For each color group, generate a range of possible alpha values, denoted as $[a,b]$, where $0 \leq a,b \leq 1$. This mimics the fact that each color group is responsible for a range of alpha values.
- For each occurrence of a color group in the ray sample, an alpha value is randomly selected within the range $[a,b]$. If the ray has fewer than the maximum possible samples, the remaining positions in the array are padded with zeros.

Table 1: Datasets and their ray-casting related information. Ray count only includes non-background rays. Samples per Ray is the maximum number of samples on a ray.

| Dataset | Resolution | Data Type | Size | Unique Ray Collected | Max Samples per Ray |
|---|---|---|---|---|---|
| Tooth | $10\times94\times161$ | Uint8 | 1.5 MB | 60000 | 187 |
| Vertebra | $256\times256\times256$ | Uint8 | 16 MB | 80000 | 366 |
| Miranda | $1024\times1024\times1024$ | Float32 | 4 GB | 170000 | 449 |
| Richtmyer | $2048\times2048\times1920$ | Uint8 | 7.5 GB | 80000 | 433 |

### 4.2 System Setup

The hardware platform is a desktop with Intel(R) Core(TM) i9-14900K CPU with 98 GB DDR5 DRAM 5200MHz. The neural network is trained and tested on the Nvidia RTX4090 with 24 GB GDDR6X VRAM. The operating system on the desktop is Ubuntu 22.04.4 LTS. PyTorch software stack is used to accelerate the training and inference of the proposed network.

### 4.3 Model Training

The model architecture follows Fig. 1. Batch size was set at 2048. The models were trained for 1,000 epochs, with early stopping set to a patience of 10 epochs and a loss difference of $3*10^{-6}$. A model is considered better than models from previous epochs if the validation loss is at least $3*10^{-6}$ lower. The loss function used was the Mean Squared Error. The learning rates for PixOrNet and OpNet were set to 0.0004 and 0.00008, respectively. These learning rates were determined through experiments.

Additionally, we also observed how the model performed in terms of PSNR results after ray clustering and model training time when the number of test cases and unique patterns vary. In both experiments, we used the Tooth dataset at 187 max samples per ray. All other factors, such as model architecture and learning rates, remain constant.

### 4.4 Ray Classification

For k-means implementation, we leveraged an existing implementation of k-means in the Sklearn python package [22]. The number of clusters is set to be around 8.5% of non-background rays of each dataset. Other parameters are set to default values. After the algorithm finishes running, for each cluster, we find a ray that is closest in the distance to the centroid and assign it as the centroid ray of the cluster. This centroid ray will share its RGB values computed through ray-casting to all other member rays of the cluster. All computations were performed without GPU acceleration.

Given the lack of existing literature on ray classification before ray casting, we utilized superpixel segmentation as a proxy to compare our approach. Specifically, we utilized Superpixel Linear Iterative Clustering (SLIC) as a baseline comparison. In all experiments, we maintain the same number of clusters between PixOrNet and Superpixel segmentation to ensure equivalent ray rendering time. The performance is measured based on the image quality post-clustering. Regarding the implementation of SLIC, we used existing code from the Scikit-learn package in Python [25]. Most parameters are left as default except compactness, which was set to 1 after hyper-parameter tuning experiments. Additionally, a mask is also passed in as a parameter to let the algorithm know where the white-space pixels are to avoid considering the backgrounds as part of a superpixel.

### 4.5 Image compression

For image compression evaluation, we assumed that RGBA values of pixels are first normalized to four 8-bit unsigned integer values from four 32-bit float values. Cluster id uses a 16-bit unsigned integer. We also included the compression results using JPEG for reference. However, since JPEG employs a slightly different method for compressing images compared to our approach for compressing cluster IDs, the results can differ significantly. Therefore, it is important to note that this comparison may not be entirely analogous.

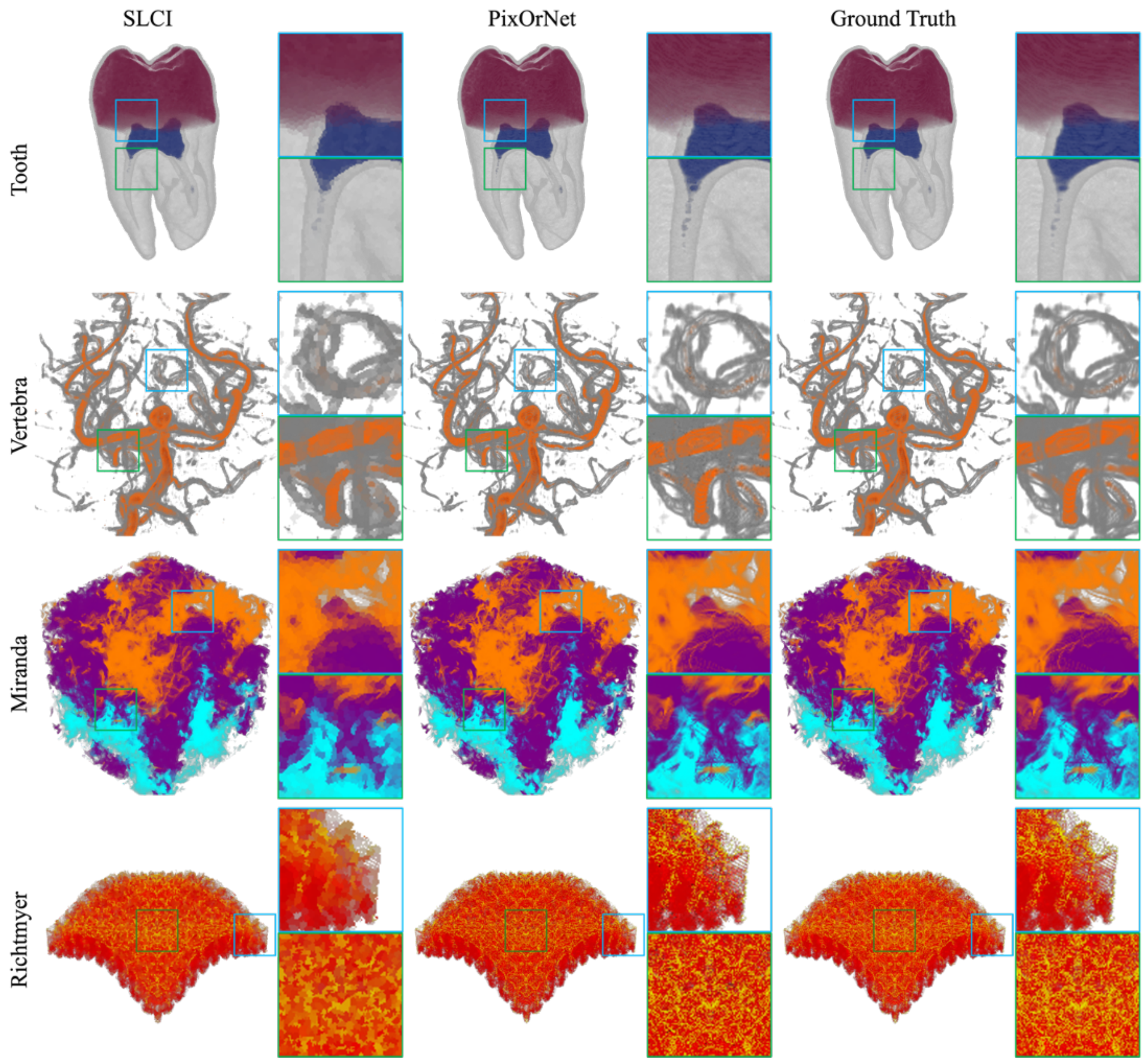


Fig. 3: Image quality comparisons between Ground Truth, PixOrNet, and SLIC close-up example of Chameleon low sampling rate dataset.

## 5 Results & Evaluation

### 5.1 Training

Each model required between 2 and 5 hours to train, depending on the ray sampling rate and the size of training and validation datasets. The training time of the model for the Miranda dataset is much higher than the rest because its training and validation datasets are twice the size of the other. In addition to using MSE as the loss function during training, we also evaluated the mean absolute error after completing the training process. Upon plotting the errors of the testing dataset on a histogram, we observed that the errors followed a normal distribution. Thus, we collected the 1-sd, 2-sd, and 3-sd ranges as additional metrics. Tab. 2 presents the mean, variance, different SD ranges, and training times for each dataset. For instance, with the vertebra dataset, 99.7% of the time, the average absolute difference between the model's predicted alpha contribution and the ground truth falls within $(-0.010, 0.013)$ in the test dataset. Given that the alpha contribution values range within $[0,1]$, the result indicates that the model is highly accurate. After assessing the model's performance, we evaluated its image quality. As shown in Tab. 3, PixOrNet+OpNet achieved high fidelity in terms of PSNR and SSIM, demonstrating that our approach is a viable method for ray casting. The results further indicate that PixOrNet serves as an effective encoder for rays.

### 5.2 Ray Classification

We then examined the performance of ray classification. The runtime for k-means clustering typically ranges between 30-60 seconds. If MiniBatchKmeans is used, runtime can get as low as 10-15 seconds. As mentioned in Sec. 3.5.1, the runtime is considered an overhead computation because k-means clustering needs to be computed only once per model. Table 4 and Table 5 summarize our rendering time result and image quality performance. Our method produces images with PSNR around 30 and SSIM around 0.94 while only using 8.5-10% of the runtime compared to ray casting. For small datasets such as the tooth dataset, our method can go up to 45 in PSNR. For datasets with high sampling rates, the time required for alpha updates is often negligible compared to that of the RGB components, resulting in an

Table 2: Model training results on the test dataset. Training time for the Miranda dataset is double that of the other datasets because it has doubled the number of training and validation test cases.

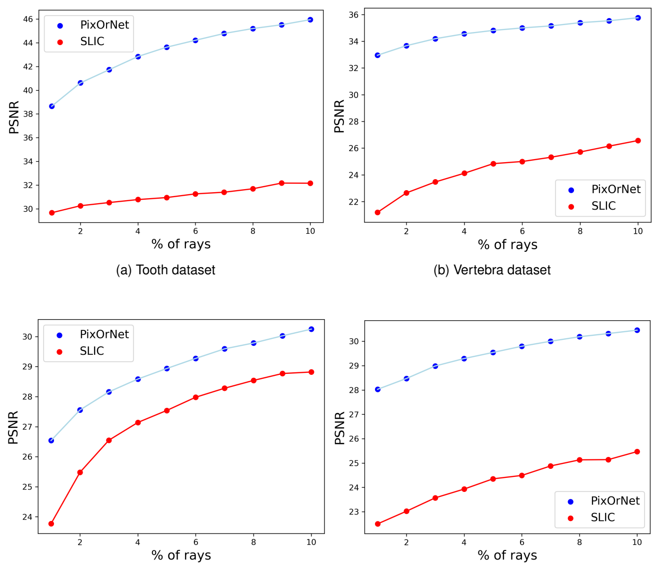
| | Mean | Variance | 1-sd Range | 2-sd Range | 3-sd Range | Training Time |
|---|---|---|---|---|---|---|
| Tooth | 0.00135 | 0.00001 | (-0.002, 0.004) | (-0.005, 0.007) | (-0.007, 0.010) | 1.4 hours |
| Vertebra | 0.00165 | 0.00001 | (-0.002, 0.006) | (-0.006, 0.009) | (-0.010, 0.013) | 2.4 hours |
| Miranda | 0.00281 | 0.00003 | (-0.002, 0.005) | (-0.005, 0.009) | (-0.009, 0.012) | 5.2 hours |
| Richtmyer | 0.00251 | 0.00002 | (-0.002, 0.007) | (-0.007, 0.012) | (-0.012, 0.016) | 3.7 hours |

Table 3: Image quality of PixOrNet+OpNet based on ground truth

| | | PixOrNet+OpNet |
|---|---|---|
| Tooth | PSNR | 47.66 |
| | SSIM | 0.9952 |
| Vertebra | PSNR | 42.68 |
| | SSIM | 0.9908 |
| Miranda | PSNR | 39.73 |
| | SSIM | 0.9814 |
| Richtmyer | PSNR | 42.98 |
| | SSIM | 0.9901 |

Table 4: Runtime result using ray classification. The number in parenthesis is the time to update alpha accumulation.

| | Our Method's Re-render Time | Ground Truth Re-render Time | % of Original Ray | % of Ground Truth Re-render Time |
|---|---|---|---|---|
| Tooth | 0.0228s (+0.003124s) | 0.268s | 8.51% | 9.67% |
| Vertebra | 0.0410s (+0.000005s) | 0.483s | 8.49% | 8.51% |
| Miranda | 0.0958s (+0.000302s) | 1.13s | 8.45% | 8.50% |
| Richtmyer | 0.0382s (+0.000075s) | 0.468s | 8.16% | 8.18% |



(a) Tooth dataset (b) Vertebra dataset (c) Miranda dataset (d) Richtmyer dataset

Fig. 4: PSNR values vs the percentages of rays of various datasets

acceleration of 12-fold. For users who want to explore models through varying opacity TFs quickly, our method offers ways to compute a new 2D image in a fraction of the time.

Figure 3 showcased the results of four datasets after being processed by SLIC and our approach. Metrics-wise, from Tab. 5, images produced by PixOrNet have higher PSNR and SSIM scores in all four tested datasets when compared to those of SLIC. The largest and smallest PSNR differences are in the Tooth dataset at 13 points and in the Miranda dataset at 2 points, respectively. As for SSIM results, PixOrNet is better than SLIC in preserving the structural integrity of the images, especially for complex datasets. For example, in the close-up view of the Miranda dataset, SLIC often blurs some of the small and fine lines together, while PixOrNet was able to preserve them.

Furthermore, Fig. 4 illustrates the results of ray classification across various cluster sizes and data models. The blue and red lines represent the PSNR of images using PixOrNet and SLIC, respectively. The feature encoded by PixOrNet demonstrates superior performance as a criterion for clustering rays. Both approaches exhibited performance improvements as the percentage of rays increased. However, the increase in image quality diminishes as more and more clusters are considered.

## 5.3 Impact of Hyper-parameters to Model Performance

Before proceeding to other applications of PixOrNet, we want to evaluate the impact of varying the number of test cases and unique patterns used on PixOrNet's performance. We observed two key metrics: PSNR point after ray classification and model training time.

Table 5: PixOrNet vs SLIC image quality comparison

| | PixOrNet PSNR | SLIC PSNR | PixOrNet SSIM | SLIC SSIM |
|---|---|---|---|---|
| Tooth | 45.40 | 32.14 | 0.99 | 0.91 |
| Vertebra | 33.45 | 25.11 | 0.97 | 0.81 |
| Miranda | 29.90 | 27.98 | 0.91 | 0.84 |
| Richtmyer | 30.23 | 24.62 | 0.97 | 0.89 |

First, we examine the results when the number of test cases changes. The number of test cases includes training, validation, and testing test cases, split at a 4:1:1 ratio. Figure 5 (a) illustrates the relationship between PSNR and Training Time across different numbers of test cases. The x-axis is the number of test cases, ranging from 800$k$ to 6300$k$. The blue line represents PSNR, while the red line represents Training Time in hours. The PSNR values show a relatively stable increase as the number of test cases increases. However, the rise increases logarithmically, suggesting diminishing returns in image quality improvement with an increasing number of test cases. On the other hand, the training time shows a roughly linear increase as the number of test cases grows. While both PSNR and training time increase with the number of test cases, the training time grows at a much steeper rate than the PSNR. This suggests that beyond a certain point, improvements in PSNR may not justify the exponential increase in training time.

After examining the impact of test case count, we evaluate how models perform if we vary the number of unique patterns that the model has access to during training. The model was tested at from 1k to 60k unique patterns. The results are shown in Figure 5 (b). The blue line represents PSNR, while the red line represents Training Time in hours. From the graph, the training time exhibits an overall upward trend as the number of unique patterns increases. Although the runtime of each epoch remains at about 52-56 seconds, models trained on a larger set of unique patterns often require more epochs to converge, resulting in longer training time. Unlike the training time result, the PSNR result increases almost linearly as the number of unique patterns at a rate of about 0.26 PSNR pt per 1000 unique patterns. A notable observation from the plot is that the PSNR performance plateaus after approximately 50,000 unique patterns. This trend suggests that the model has reached a saturation point. In other words, for the tooth

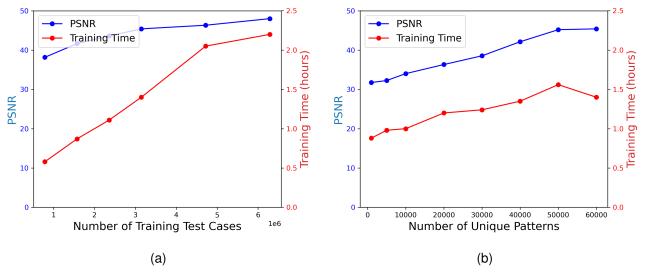


Fig. 5: Image quality after ray clustering and training time (hours) across (a) different numbers of test cases and (b) different numbers of unique patterns

dataset, 50000 unique rays were sufficient for the model to learn ray representation. This insight is valuable for optimizing computational resources because increasing the unique pattern count beyond this point leads to additional training time without an equivalent improvement in image quality.

After assessing the impact of the number of test cases and unique patterns used for model training, we conclude that while larger test cases and unique patterns result in higher image quality, it is more advantageous to increase the number of unique patterns used because the image quality increase per additional time spent training is much more significant.

## 5.4 Compression

We also evaluated the compression ratios achieved using our approach, as discussed in Sec. 3.5.2 and compared the results to JPEG compression, one of the premier options whenever image compression is necessary. Unlike JPEG, which is lossy, our compression scheme is lossless. As shown in Tab. 6, the compression scheme employing our clustering method uses similar footprints in 3 out of 4 tested datasets. By itself, it is a considerable achievement because our approach is lossless. For the Richtmyer dataset, the compressed space of our approach is 40% of that of JPEG. The primary reason behind this is that JPEG relies on spatial information during the Discrete Cosine Transform step to maintain the visual appeal of the image. While spatial information is useful, it makes JPEG prone to constant changes at the pixel level, which is similar to SLIC. On the other hand, because our method relies on information encoded from rays that produce the pixels, we can better identify which pixels are related to each other. Moreover, we observed that the alpha channel compression was generally less effective than compressing the entire RGBA data together. For example, in the vertebra dataset, the compression rate of the alpha channel is only half of that of cluster id. Thus, the performance of our approach is often limited by the poor performance of Run-Length Encoding on the alpha channel. Hence, future research into a more effective compression scheme for the alpha channel can significantly enhance our results.

## 5.5 Reconstruction Neural Network

Finally, we evaluate how our method can help to improve the rendering latency of already fast volume visualization methods leveraging reconstruction neural networks. We select EnhanceNet [23] as the super-resolution network for its state-of-the-art reconstruction quality and efficiency observed in volume visualization applications [28]. In this work, we consider a super-resolution method to generate a high-res volume rendering visualization image from a low-res partially rendering image. First, the super-resolution network is trained on low-res and full-res image pairs generated using ray-casting volume rendering from randomly selected view angles for the tooth dataset. Second, during the rendering stage, both methods run on the same hardware platform to generate the partially rendered image using 8 threads of the CPU. We qualitatively compare both the quality and performance using our method and the traditional ray-casting approach. The overall rendering latency is the sum of the time used to generate the partially

Table 6: Results when using ray classification to compress images. All results are measured based on an original size of 4 8-bit data used for RGBA values. The third and fourth rows of each dataset break down the compression efficiency of our compression scheme. Unlike our compression scheme, JPEG is lossy.

| | | Compression Rate | % of Original Data |
|---|---|---|---|
| Tooth | JPEG compression | 7.69 | 13.01% |
| | Our proposed compression | **9.09** | 11.00% |
| | Cluster Id compression | 11.58 | 8.63% |
| | alpha channel compression | 5.52 | 18.11% |
| Vertebra | JPEG compression | **6.42** | 15.58% |
| | Our proposed compression | 6.37 | 15.70% |
| | Cluster Id compression | 8.18 | 12.22% |
| | alpha channel compression | 3.82 | 26.15% |
| Miranda | JPEG compression | **6.01** | 16.63% |
| | Our proposed compression | 4.86 | 20.57% |
| | Cluster Id compression | 4.73 | 21.16% |
| | alpha channel compression | 5.33 | 18.78% |
| Richtmyer | JPEG compression | 3.86 | 25.89% |
| | Our proposed compression | **9.62** | 10.40% |
| | Cluster Id compression | 9.87 | 10.13% |
| | alpha channel compression | 8.93 | 11.20% |

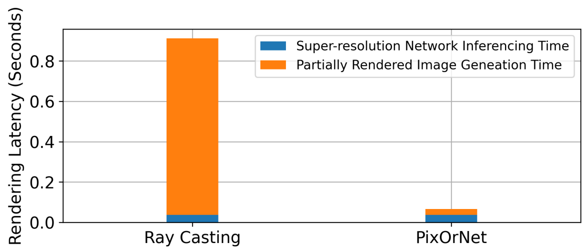


Fig. 6: Overall rendering latency comparison between traditional Ray Casting and PixOrNet.

rendered image and the inferencing time of the super-resolution network. Fig. 6 shows PixOrNet method can generate the partially renderer image about 22 times faster than the Ray Casting. Fig. 7 shows that although PixOrNet generates the partially rendered image faster, the final rendering quality from the super-resolution network is very similar to the baseline, which uses Ray Casting. We can observe that the superpixel derived from PixOrNet can further improve the rendering time of volume visualization methods using reconstruction neural networks, like super-resolution networks, without noticeable rendering quality degradation.

## 6 Discussion

**OpNet as a replacement for ray casting**: In this work, we explored how PixOrNet+OpNet is an alternative renderer that does not use ray casting to generate 2D images. While we have shown that PixOrNet can be a ray encoder for ray classification, we did not fully utilize OpNet because of its slow inference speed in a CPU setting. While we were able to reduce inference speed significantly, it is still about 33 times slower compared to traditional DVR when GPU is not used.

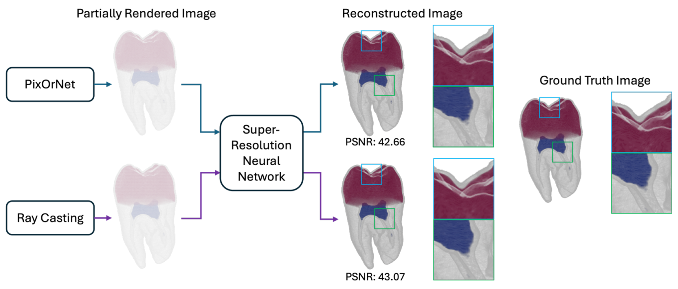


Fig. 7: Quality comparison of reconstructed visualization using traditional Ray Casting or PixOrNet to generate the partially rendered image for the super-resolution neural network.

If GPU is allowed, OpNet can be as competitive, if not better, than traditional DVR because OpNet, as a neural network, can leverage GPU acceleration more efficiently, especially when OpNet only has 3 hidden layers.

**Artifacts and alternative clustering method**: Similar to other optimization strategies for volume rendering, our approach suffers from artifacts. For example, in the close-up view of the Richtmyer dataset, there are patches of small dark-colored dots. From our testing, the primary reason is the clustering method. More specifically, the problem lies in how encoded features are compared against each other. In our paper, we pick Kmeans as a clustering strategy because it is straightforward to implement and test. As proof of concept, Kmeans has showcased the effectiveness of our strategy in ray classification. However, in k-means, all 64 features are treated equally when calculating the similarity or distance between two data points (or rays in our case). On the other hand, because these same 64 features are neurons in PixOrNet + OpNet, the significance of each neuron can be uneven. Therefore, in the future, we would like to explore other more robust clustering strategies [6, 29] for clustering and possibly integrate state-of-the-art neural network analysis to better weigh the importance of each neuron.

## 7 Conclusion

In this paper, we have presented PixOrNet and OpNet as an alternative way to render rays without conducting ray casting and PixOrNet as a feature encoder for rays. By leveraging the encoded features, we can perform ray classification, enabling image synthesis in a fraction of the time when users adjust the color and opacity transfer function. We showed that our ray classification produced images that are more similar to the ground truth compared to those generated by Superpixel segmentation by considering the alpha contribution of the ray's samples behind each pixel. Besides accelerating rendering pipelines, we also demonstrated other possible applications of PixOrNet such as in image compression and speeding up other neural-network-based image synthesis. In the future, we would like to explore if we can enhance the inference speed of PixOrNet+OpNet to rival that of ray casting, allowing for faithful image synthesis while bypassing the linear processing property of ray casting.

## References

[1] D. Bauer, Q. Wu, and K.-L. Ma. Fovolnet: Fast volume rendering using foveated deep neural networks. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):515–525, 2023. doi: 10.1109/TVCG.2022.3209498 2

[2] M. Berger, J. Li, and J. A. Levine. A generative model for volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 25(4):1636–1650, 2019. doi: 10.1109/TVCG.2018.2816059 2

[3] J. Beyer, M. Hadwiger, and H. Pfister. State-of-the-art in gpu-based large-scale volume visualization. *Computer Graphics Forum*, 34(8):13–37, 2015. doi: 10.1111/cgf.12605 2

[4] H. Chang, O. Fried, Y. Liu, S. DiVerdi, and A. Finkelstein. Palette-based photo recoloring. *ACM Trans. Graph.*, 34(4), article no. 139, 11 pages, jul 2015. doi: 10.1145/2766978 2

[5] J. Chen, Z. Li, and B. Huang. Linear spectral clustering superpixel. *IEEE Transactions on Image Processing*, 26(7):3317–3330, 2017. doi: 10.1109/TIP.2017.2651389 2

[6] K.-L. Du. Clustering: A neural network approach. *Neural Networks*, 23(1):89–107, 2010. doi: 10.1016/j.neunet.2009.08.007 9

[7] 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. 2

[8] B. Gong, Y. Wang, X. Han, and Q. Dou. Recolornerf: Layer decomposed radiance fields for efficient color editing of 3d scenes. In *Proceedings of the 31st ACM International Conference on Multimedia*, MM '23, 12 pages, p. 8004–8015. Association for Computing Machinery, New York, NY, USA, 2023. doi: 10.1145/3581783.3611957 2

[9] 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, 2020. doi: 10.1109/TVCG.2019.2934255 5

[10] J. Han and C. Wang. Ssr-tvd: Spatial super-resolution for time-varying data analysis and visualization. *IEEE Transactions on Visualization and Computer Graphics*, 28(6):2445–2456, 2022. doi: 10.1109/TVCG.2020.3032123 2

[11] 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, 2022. doi: 10.1109/TVCG.2021.3114815 2, 5

[12] W. He, J. Wang, H. Guo, K.-C. Wang, H.-W. Shen, M. Raj, Y. S. G. Nashed, and T. Peterka. Insitunet: Deep image synthesis for parameter space exploration of ensemble simulations. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):23–33, 2020. doi: 10.1109/TVCG.2019.2934312 2

[13] A. S. Kaplanyan, A. Sochenov, T. Leimkühler, M. Okunev, T. Goodall, and G. Rufo. Deepfovea: neural reconstruction for foveated rendering and video compression using learned statistics of natural videos. *ACM Trans. Graph.*, 38(6), article no. 212, 13 pages, nov 2019. doi: 10.1145/3355089.3356557 2

[14] J. Kniss, M. Ikits, A. Lefohn, C. Hansen, E. Praun, et al. Gaussian transfer functions for multi-field volume visualization. In *IEEE Visualization, 2003. VIS 2003.*, pp. 497–504. IEEE, 2003. 3

[15] Z. Kuang, F. Luan, S. Bi, Z. Shu, G. Wetzstein, and K. Sunkavalli. Palettenerf: Palette-based appearance editing of neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 20691–20700, June 2023. 2

[16] B. V. Kumar. An extensive survey on superpixel segmentation: A research perspective. *Archives of Computational Methods in Engineering*, 30:3749–3767, 2023. doi: 10.1007/s11831-023-09919-8 2

[17] Y. Lan, Y. Ding, X. Luo, Y. Zhang, C. Huang, E. Y. K. Ng, W. Huang, X. Zhou, J. Su, Y. Peng, Z. Wang, Y. Cheng, and W. Che. A clustering based transfer function for volume rendering using gray-gradient mode histogram. *IEEE Access*, 7:80737–80747, 2019. doi: 10.1109/ACCESS.2019.2923080 1

[18] P. Ljung, J. Krüger, E. Groller, M. Hadwiger, C. D. Hansen, and A. Ynnerman. State of the art in transfer functions for direct volume rendering. In *Computer graphics forum*, vol. 35, pp. 669–691. Wiley Online Library, 2016. 1

[19] G. Lochmann, B. Reinert, A. Buchacher, and T. Ritschel. Real-time novel-view synthesis for volume rendering using a piecewise-analytic representation. In *VMV'16 Proceedings of the Conference on Vision, Modeling and Visualization*, vol. 2016. Eurographics, 2016. 2

[20] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995. 2

[21] 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. *Communications of the ACM*, 65(1):99–106, 2021. 1, 2

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 5

[23] M. S. M. Sajjadi, B. Schölkopf, and M. Hirsch. Enhancenet: Single image super-resolution through automated texture synthesis. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 4501–4510, 2017. doi: 10.1109/ICCV.2017.481 8

[24] J. Tan, J.-M. Lien, and Y. Gingold. Decomposing images into layers via rgb-space geometry. *ACM Trans. Graph.*, 36(1), article no. 7, 14 pages, nov 2016. doi: 10.1145/2988229 2

[25] S. J. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, June 2014. doi: 10.7717/peerj.453 5

[26] M. Wang, X. Liu, Y. Gao, X. Ma, and N. Q. Soomro. Superpixel segmentation: A benchmark. *Signal Processing: Image Communication*, 56:28–39, 2017. doi: 10.1016/j.image.2017.04.007 2

[27] S. Weiss, M. Chu, N. Thuerey, and R. Westermann. Volumetric isosurface rendering with deep learning-based super-resolution. *IEEE Transactions on Visualization and Computer Graphics*, 27(6):3064–3078, 2021. doi: 10.1109/TVCG.2019.2956697 2, 5

[28] S. Weiss, M. IşIk, J. Thies, and R. Westermann. Learning adaptive sampling and reconstruction for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 28(7):2654–2667, 2022. doi: 10.1109/TVCG.2020.3039340 8

[29] R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005. 9

[30] A. Yu, R. Li, M. Tancik, H. Li, R. Ng, and A. Kanazawa. Plenoctrees for real-time rendering of neural radiance fields, 2021. 2