FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jaroslav Nejedlý

Neural representations for differentiable volume rendering

Department of Software and Computer Science Education

Supervisor of the master thesis:	Tobias Rittig, PhD
Study programme:	Computer Science
Study branch:	Computer Graphics and Game Development

Prague 2024

I'm extremely grateful to my supervisor Dr. Tobias Rittig. Without consultations with him, his helpful insights and his patience, this work would be impossible to make. I am also thankful to the Computer Graphics Group at Charles University for letting me use their infrastructure, especially the compute servers. Lastly, I'd like to mention my wife Sarah, who thoroughly supported me while I was working on this thesis.

Název práce: Reprezentační neuronové sítě pro diferencovatelné renderování objemu

Autor: Jaroslav Nejedlý

Katedra: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Tobias Rittig, PhD, Katedra softwaru a výuky informatiky

Abstrakt: Tato práce prozkoumává možnost použití reprezentačních neuronových sítí jako datových struktur pro modely předpovídající vzhled. Představujeme representační síť, která je adaptací populární NeRF sítě. Tato reprezentační síť je studována na 2D obrázcích i trojrozměrných volumetrických datech. Tato práce také vyhodnocuje výstupy modelu, který předpovídá vzhled, do kterého je použita reprezentační síť jako zdroj dat.

Naše analýza ukazuje, že reprezentace 2D obrázků a jednoduchých volumetrických útvarů je realizováno s obstojnou kvalitu. Nicméně, výstupy vzhled předpovídající sítě jsou suboptimální.

Vyvodili jsme, že reprezentační sítě představené v této práci by měli být vylepšeny a že model předpovídající vzhled by měl být doladěn s použitím reprezentační sítě na vstupu.

Klíčová slova: reprezentační sítě, neurální renderování, MLP, NeRF

Title: Neural representations for differentiable volume rendering

Author: Jaroslav Nejedlý

Department: Department of Software and Computer Science Education

Supervisor: Tobias Rittig, PhD, Department of Software and Computer Science Education

Abstract: This thesis investigates the possibility of using a representation neural network as a data structure for an appearance prediction model. We present the representation network as an adaptation of the popular neural radiance field network. The representation network is studied on 2D images as well as volumetric data. This thesis also evaluates the outputs of the appearance prediction network which uses the representation network.

Our analysis shows a decent quality of the representation network for 2D images and simple volumetric data. However, the outputs of the appearance prediction network are suboptimal.

We conclude that the representation network presented in this thesis should be improved and the appearance prediction model should be fine-tuned to the representation network as its input.

Keywords: representation network, neural rendering, MLP, NeRF

Contents

Introduction

In the past few years, 3D printing has become more popular among the general public as well as among various research groups. A lot of attention is paid to color 3D printing as it has some challenges associated with it. Given the small size of the individual droplets, during the calibration of the color, special attention has to be provided to the volumetric interactions between the light and the material. As the light can travel through many droplets before being absorbed or scattered away. This makes it computationally intensive to predict the appearance of the final product. Rittig et al. [2021] accelerated the prediction of light scattering by using a neural network. In our thesis, we attempt to swap the data structure used by this scattering prediction network with our solution, the representation network.

We aim to represent the volumetric data with a shallow neural network, that given a spatial position as its input outputs desired volumetric properties. Traditionally, volumetric data is represented by grids. Given the sparse nature of the data, appropriate representation of grids can be memory efficient. On the other hand, the neural network approach promises the data structure to be differentiable. This could open the door for various optimization algorithms that would optimize the distribution of 3D printer droplets to match some desired look. For example, Elek et al. [2017] try to optimize this droplet distribution using more traditional techniques like Monte Carlo rendering and dense gird representation. Differentiability is a desired property for optimization as shown by Nindel et al. [2021].

The field of representation neural networks is subject to very active investigation. Since the start of our research, there have been several advancements. For example, Karnewar et al. [2022] combine grid-based and neural network approaches to achieve promising results. Müller et al. [2022] use spatial hashing functions to accelerate the positional encoding, which enables nearly instant training times on high-end hardware. However, we decided to use only purely neural concepts.

Our representation network is based on the earlier work of Mildenhall et al. [2020] and Barron et al. [2021] who introduced a network that can store high-frequency details by utilizing so-called Fourier feature networks. These work by enhancing the input to the network by positional encoding. We focus on exploring the ideas behind Fourier feature networks and frequency-based encoding. We also try to connect the representation network with the scattering prediction network to evaluate the possibility of future applications and improvements.

In the first chapter, we will introduce the necessary background and theoretical knowledge about signal processing, particularly about Fourier transforms and convolutions. We will also include information about modern machine learning algorithms and strategies as well as neural network architectures. The first chapter also includes information about the light transport equation and rendering.

The second chapter focuses in detail on the previous work that our solution builds upon. The first is the light scattering prediction with neural networks, we will explore how the network works in detail, especially what kind of inputs it requires. The second is the Neural radiance fields and other works derived from them. We will focus on the exact inner workings of such representation networks and how can we adapt them to our intended usage.

The third chapter will discuss our solution in detail from a theoretical perspective. Mainly focusing on the architecture of the representation networks, their training, and the connection to the scattering prediction network. The fourth chapter will briefly discuss the actual implementation details of the solution described in the third chapter. The implemented solution is provided in Attachment A.1. The documentation of our software is presented in Attachment A.2.

In the fifth chapter, we present the results of the representation network. We take a look at representing two-dimensional images as well as representing volumes. The results of appearance prediction network that uses our representation network are also unveiled in the fifth chapter. And in the final, sixth, chapter we discuss these results. We also discuss limitations and issues present in our method and possible future work that would improve upon our result.

1. Background

In this chapter, we introduce terminology, equations, and principles that are fundamental to our work. These topics include machine learning, rendering, and signal processing. The last mentioned is used in various parts of the thesis as it includes equations that describe operations on signals. These operations have important properties that are used in our method as well as in previous work. An understanding of the fundamentals of machine learning and artificial neural networks is important as our work proposes a new way of representing voxel data with representation neural networks. We use this representation to predict the appearance of color 3D prints, so a brief introduction to light transport algorithms is included in the rendering section.

1.1 Signal processing

Signal processing plays an important role in analyzing, capturing, and manipulating visual data. Various algorithms in this field can give us information about signal quality or they can help us to understand the frequency composition. Other functions can modify the signal in ways that correspond to various effects. Let's explore some of these functions and algorithms.

Fourier transform A key function in signal processing is a Fourier transform that facilitates a transition between spatial and frequency domains. These two domains refer to the variables that are used to parameterize the signal. In the spatial domain, we use three-dimensional coordinates that describe the actual position in the space, usually denoted as x, y, and z. In the frequency domain, we use frequencies of the signal to describe the amplitude and phase at that frequency. For example, a pure sine signal would be represented by a single point in the frequency domain.

For multi-dimensional domains, we use wave vectors instead of frequencies. These describe the direction as well as the frequency of a signal. To obtain amplitude and phase in a single number we describe these parameters as a complex number, so it shouldn't be a surprise that the output of Fourier transform is a complex number.

To better understand these terms, let's focus on a simple one-dimensional signal first. The following equation describes the Fourier transform of a one-dimensional function:

$$\mathcal{F}(\xi) = \int_{-\infty}^{\infty} f(x) \exp\left(-i2\pi\xi x\right)\mathrm{d}x \tag{1.1}$$

Where $f(x)$ is the function being transformed, $F(\xi)$ is the result of the transform. Note that the original function operates on real numbers, whereas its Fourier counterpart is a complex function.

If the original function f operated on time ($x = t$) then the parameter ξ of the transform is frequency. By evaluating the Fourier transform for all frequencies we get a frequency spectrum of the original function f.

However, for image processing, we are working with sampled values of a function and computing such integral for *all* frequencies is not feasible. Instead, we can compute *discrete Fourier transform* (DFT) which for each sample of the investigated function produces a complex number each corresponding to a different frequency. Given a sequence of numbers $\{f_n\}_0^N$, we can compute DFT by the following equation:

$$\mathcal{F}_n(f) = \sum_{k=0}^{N} f_k \exp\left(-i2\pi \frac{k}{N+1} n\right) \tag{1.2}$$

As you can see, Equation 1.2 is a finite discrete special case of Fourier transform (1.1). From a known identity $\exp(-ix) = \cos(x) + \imath \sin(x)$, we can conclude that the sum in the DFT equation is a sum of sine and cosine functions of increasing frequency.

Convolution Another important operation used in signal processing is a convolution of two functions. To intuitively understand convolutions we can think of blurring a 2D image. This process involves combining the neighboring values in some way. For example, the Gaussian blur weights the values by the Gaussian function.

Mathematically, the convolution can be expressed by the following integral:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(s) g(s-t) \mathrm{d}s \tag{1.3}$$

Where f and g are two generalized functions, usually one is a signal while the other is a kernel. The shape of a kernel determines the output of the convolution operation. For example, if g is a Dirac delta function, the output will be identically function f. As mentioned above, another example is Gaussian distribution, which is commonly used to smooth out signals. The other smoothing kernel that can be employed is a box-shaped kernel, whose output averages the signal with equal weighting. These smoothing kernels have to be normalized in a way that leads to the integral across the domain of the kernel evaluating as one. This ensures that the amplitude of the signal isn't affected.

Figure 1.1 shows the two different kernels. The Gaussian with a mean value of zero and standard deviation (SD) of one in red and a box-shaped kernel with a mean value of zero and radius of two. Equation 1.4 is expressing a Gaussian curve g. The two parameters are mean μ and standard deviation σ. In this thesis, we work with these two kernels.

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) \tag{1.4}$$

$$\mathcal{F}(\xi) = \exp\left(-\frac{1}{2}\xi^2\sigma^2\right) \tag{1.5}$$

Equation 1.5 shows a Fourier transform of a Gaussian kernel with a standard deviation of σ. We can see that apart from the normalization constant the Fourier transform of the Gaussian curve is the same curve except the standard deviation is reciprocal.

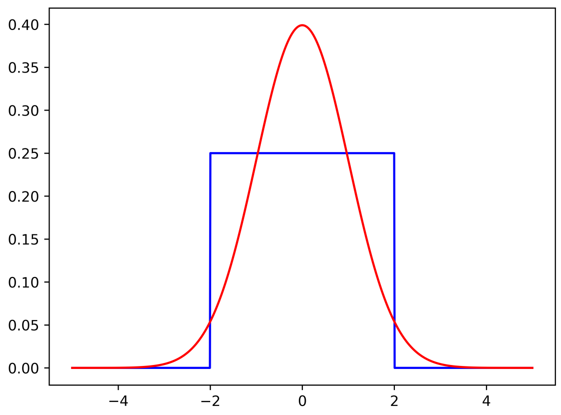


Figure 1.1: Comparison of a box kernel (blue) and Gaussian kernel (red).

When applying convolution to a finitely sampled signal with a bounded domain, we encounter a problem: How should we handle the edges where the signal starts and ends? Typically, this is handled by one of the following strategies:

1. Use a *constant* value for the points outside of the domain of the signal. When a value of zero is used this would correspond to a direct application of the mathematical formula described by Equation 1.3.

2. *Mirroring* the signal outside of its domain.

3. *Wrapping* the signal outside of its domain. This is especially useful for periodic signals.

4. *Cropping* the signal or the kernel. When a signal is cropped the result of a convolution will have smaller dimensions than the original signal sample which might be undesired. When a kernel is cropped additional normalization of the cropped kernel might be required.

5. Using the *nearest* value of the signal. For a one-dimensional signal, this strategy would lead to repeating the final and last samples as many times as needed. This is the strategy that we use in our work.

One of the advantages of the Fourier transform is that a convolution of two functions is equal to the multiplication of their Fourier transforms. If we would like to compute a convolution of any function with the Gaussian kernel in the frequency domain, we could do so by computing the Fourier transform of said function and multiplying it with the function described in Equation 1.5.

Signal metrics To compare different signals we can establish a metric that would quantify the similarity of the two signals. Let's define two signals: a noise-free signal S and a signal, which is compromised with noise S'. Given N samples of the two signals, we can compute the *mean squared error* (*MSE*) of the noise signal as follows:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(S_i - S'_i)^2 \tag{1.6}$$

Sometimes, instead of using the MSE value as is, the square root of it is used. In that case, we call it *root-mean-square error* (RMSE). We can easily extend Equation 1.6 into two dimensions. Given a noise-free image I with dimensions w and h and its noisy counterpart I' MSE value can be computed by the following equation:

$$MSE = \frac{1}{w \cdot h} \sum_{x=1}^{w} \sum_{y=1}^{h} (I(x, y) - I'(x, y))^2 \tag{1.7}$$

To determine how noisy the image I' is, we can define a value that would quantify the noisiness. A common term that is used to describe this phenomenon is a *peak signal-to-noise ratio* (PSNR). Given that our image is normalized (e.g. maximum value of the individual pixels never exceeds 1) we can compute PSNR using this equation:

$$PSNR = -10 \cdot \log_{10}(MSE) \tag{1.8}$$

In the rest of our work, most of the equations described in this section will be used on two- or even three-dimensional signals. The generalization into higher dimensions shouldn't be problematic as the signal dimensions are independent of each other. We will also use normalized values where applicable. To normalize a signal, we look for the largest value and divide all signal values by it.

1.2 Machine learning

Machine learning is a process of optimizing a function by a set of parameters so that the function best approximates a given data set. Basically, the machine is the computer and by optimizing the function it learns the target data. When the training data contains inputs as well as the target values, the process is called supervised machine learning. The basic technique that illustrates this process is called *gradient descent*.

During training, the function is calculated on the provided inputs. A *loss function*, that measures the distance to the desired value, is evaluated. From the value of the loss function, a gradient with respect to the parameters is constructed and a small step in the direction opposite to the gradient is taken. After sufficient steps were taken, the loss function should be at a local minimum and thus the optimized function approximates the given data.

Loss function There is a wide variety of functions that can be used to measure loss. There is one constraint: the loss has to be differentiable, in order to compute the gradient.

A common function that is widely used is MSE (Equation 1.6). Another function that can be used is mean absolute percentage error (MAPE). The benefit of MAPE is that it is a percentage error, which works great for both large and small absolute values of the optimized function. Given N samples of optimized function $\{f\}_0^{N-1}$ and N samples of predicted values $\{f'\}_0^{N-1}$, we can compute MAPE as follows:

$$MAPE = \frac{100}{N} \sum_{i=0}^{N-1} \left| \frac{f_i - f'_i}{f_i} \right| \tag{1.9}$$

Neural network building blocks Now, let's focus on the actual architecture of neural networks. A simple network architecture is a feed-forward network consisting of layers that connect to the outputs of previous layers. The basic layer type that is widely used is the *dense layer* also sometimes called the *fully connected layer*. Dense layers are set of weights and biases that alter their input by the following equation:

$$\mathbf{D} = f_a(\mathbf{W} \cdot \mathbf{i} + \mathbf{b}) \tag{1.10}$$

Where $\mathbf{D}$ is the output of the dense layer, f_a is the activation function, $\mathbf{W}$ is the weights matrix, $\mathbf{i}$ is the input of the dense layer and $\mathbf{b}$ are biases of the dense layer. Note that the input might not be a vector but rather a tensor of any dimensions. Weighs and biases then have appropriate dimensions. Matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ are the parameters that are subject to optimization. We take the derivatives with respect to these values to get the gradient along which these values should be changed.

Activation function f_a can be chosen from many widely used functions. Its role is that it brings non-linearity into the system. The most popular mathematical functions used as activation functions are tanh, relu, softplus, sigmoid, and others. Equations (1.11), (1.12), (1.13), and (1.14) describe these functions:

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{1.11}$$

$$ReLU(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{1.12}$$

$$Softplus(x) = \ln(1 + \exp(x)) \tag{1.13}$$

$$Sigmoid(x) = \sigma(x) = \frac{1}{1 + \exp(-x)} \tag{1.14}$$

Connecting multiple dense layers together, we can construct an intricate function with many parameters that can be optimized to represent a wide variety of data. To get a meaningful output of the network we can connect a dense layer with a smaller number of outputs. At a minimum, we would need three layers: input, hidden, and output. Usually, more layers are used. Such a configuration of a neural network is commonly referred to as a *multilayer perceptron* (MLP). Figure 1.2 shows the typical architecture of MLP. Note, that the number of hidden layers might differ as well as their size. Also, notice that the final output layer might use a different activation function.

Stochastic gradient descent To successfully optimize a neural network we have to know the all partial derivatives of the nodes involved. Usually, we can obtain the derivative of all functions involved in the optimization process by automatic differentiation. From the derivative, we can compute the gradient with

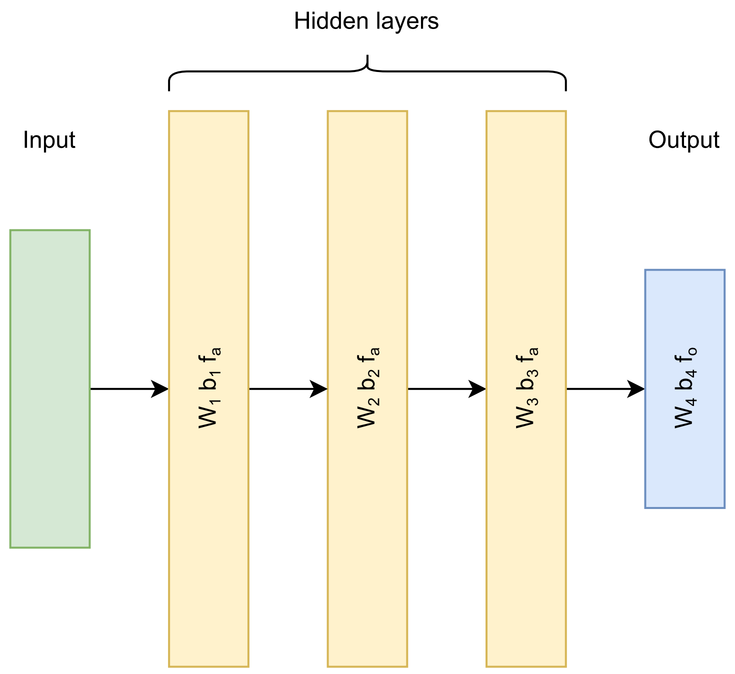


Figure 1.2: Typical architecture of MLP networks.

respect to the optimized parameters and use it for the *stochastic gradient descent* (SGD) algorithm. The SGD algorithm works by sampling a so-called minibatch of m examples from the training dataset along with their target values. Using these samples, the gradient $\mathbf{g}$ is calculated. The samples are taken randomly and in theory, the batching should provide better gradients that take into account more data at once. Once the gradient is calculated, the parameters of the network are changed by $-\epsilon \cdot \mathbf{g}$, where ϵ is a scalar value called *learning rate*.

According to Goodfellow et al. [2016], the learning rate is a crucial parameter that determines the performance of the SGD optimization. Choosing a proper rate is often done by trial and error as there is no scientific way of establishing optimal value. The ideal tool for setting a learning rate is observing the learning curves that plot the loss against the number of steps taken. Changing the sample size m of the minibatch might require a change in the learning rate.

It is also important to decay the learning rate over the steps taken. Sampling the minibatch from the training dataset introduces noise into the gradient calculation. So even though mathematically, the magnitude of the gradient should be decreasing as the parameters are reaching optimal value and at the minimum, the gradient is zero. The noise causes the gradient to never be zero. The decayed learning rate helps to reach the optimum without overshooting.

A simple way to decay the learning rate is to linearly interpolate between the larger initial learning rate and the smaller final learning rate as suggested by Goodfellow et al. [2016]. Another way is to use so-called cosine decay presented by Loshchilov and Hutter [2017]:

$$\epsilon = \epsilon_{min} + \frac{1}{2}(\epsilon_{max} - \epsilon_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{tot}}\pi\right)\right) \tag{1.15}$$

Where ϵ is the learning rate for the current step T_{cur}; ϵ_{min} and ϵ_{max} are minimal and maximal learning rates that we can choose and T_{tot} is the total amount of steps to be taken. Note, that the original article describes also the process of restarting the learning rate. This means that the learning rate is reset to a higher value. For more detail refer to the article by Loshchilov and Hutter [2017].

An update of the neural weights and biases is taken at every step. Once all of the dataset is processed and all steps were taken, we call that an *epoch*. The training of neural networks usually takes multiple epochs to arrive at the optimum. The learning rate can be decayed based on epochs or the learning rate can be updated every step. In the case of updating the learning rate every step, the value T_{cur} represents all steps over all epochs.

Given how crucial the learning rate is for SGD, there are other optimization algorithms that try to introduce an adaptive learning rate. Kingma and Ba [2014] proposed an optimization *algorithm with adaptive moments* (Adam). In general, *momentum* can be thought of as a way to carry over the gradient from previous steps. Imagine a ball rolling downhill, the shape of the hill is our loss function, the direction of the slope is our gradient and the velocity of the ball is the momentum. Eventually, the ball will reach the valley, which means that it reached a minimum of the loss function. Optimization algorithms with momentum require an extra parameter that controls how much the momentum is affected by the gradient. Goodfellow et al. [2016] states that in Adam, momentum is calculated as an estimate of the first-order moment of the gradient. Adam also includes bias corrections to the estimates of both first-order and second-order moments which are computed as exponentially weighted moving averages. The smoothing factors for the exponential weighting are parameters of the Adam algorithm.

There are multiple reasons to sample data in minibatches. The first reason, as previously mentioned, is that it provides a better estimate of the gradient than just taking one example from the dataset. The other is that it provides a performance advantage as larger batches tend to utilize modern multicore hardware. The ideal size of the minibatches is affected by several factors. The larger the batch size is, the less frequent the updates are over one epoch. This balances out the effect of a more precise gradient. Larger batches can also be limited by the amount of available memory. Wilson and Martinez [2003] argues that smaller batch size introduces a regularization effect in the neural network.

Generalization performance The power of modern machine learning algorithms and neural networks lies in the ability to produce a reasonable prediction even for inputs it was never trained on before. For that reason, it is a common practice to split the available data into training and validation parts. The training part is used during the learning process, while the validation part is used to evaluate the loss function along with any metrics without updating the weights and biases of the neural network. Ideally, we want the loss function to be decreasing even in the validation part of the data with each epoch. When the network is doing well even with unseen inputs, we say that it is *generalizing* well. The opposite phenomenon is overfitting, which happens when the network is trained with a very small loss on training data, but validation data show a higher loss.

To monitor the performance of our optimization process as well as generalization capabilities we can define more functions that similarly to loss measure the difference between prediction and actual values. However, for various reasons, we don't want to include them in the SGD optimization process. These functions are referred to as *metrics*. As metrics don't participate in the training process they don't need to be differentiable. For example for a network that is used to recognize shapes, we can use prediction accuracy as a metric.

One of the techniques that can improve the training of feed-forward MLP networks is the utilization of *skip connections*. They work by skipping some hidden layers and later concatenating with the output of those layers. This concatenated vector is used as an input to the next layer.

1.3 Rendering

The term rendering in computer graphics refers to the process of creating a (usually) 2D image that represents some three-dimensional virtual scene. Physically-based rendering tries to simulate how the light interacts with objects by evaluating physically correct equations and as such tries to predict the real-world appearance of said virtual scene. To successfully predict the appearance of such a scene, we have to understand the basic physical principles of light propagation. To compute the pixel values of an image, we have to compute the incoming light intensity hitting the pixel. The incoming light intensity can be expressed by the following equation:

$$L_o(\mathbf{x}, \omega_o, \lambda) = L_e(\mathbf{x}, \omega_o, \lambda) + \int_\Omega f_{brdf}(\mathbf{x}, \omega_i, \omega_o, \lambda) L_i(\mathbf{x}', \omega_i, \lambda)(\omega_i \cdot \mathbf{n}) \mathrm{d}\omega_i \tag{1.16}$$

Where L_o is the radiance coming from a point $\mathbf{x}$ in a direction ω_o at wavelength λ. It consists of two terms, the emission L_e and reflected radiance expressed by the integral. Notice that the integral evaluates recursively as the L_i is essentially the radiance from a point $\mathbf{x}'$ coming in a direction ω_i. The integral integrates over solid angle Ω which corresponds to the hemisphere defined by surface normal $\mathbf{n}$ at point $\mathbf{x}$. The bidirectional reflectance distribution function (BRDF) f_{brdf} describes the interaction between light rays and the surface. It dictates the distribution of directions of reflected light. For example, a matte surface would have the distribution roughly equal in all directions and glossy material will have a small cone where the majority of light will be reflected. To be more precise the function f_{brdf} evaluates to the probability of a ray of a wavelength λ reflecting at point $\mathbf{x}$ from direction ω_i to direction ω_o.

Equation 1.16 was introduced by Kajiya [1986] and serves as one of the fundamental equations in computer graphics. In this form, it describes the way light interacts with surfaces. The integral is usually evaluated by the Monte Carlo integration method. The intersection between light rays and the scene is computed by ray tracing. This technique is computationally intensive and thus reserved for offline rendering. Even though in recent years attempts were made to enable some ray-traced effects in real-time applications.

To obtain a color image, we should get three values for each pixel (red, green, and blue subpixels). To do so, we can evaluate the equation over many different wavelengths λ to obtain full spectral data at a given pixel and then multiply the result with the channel spectral response curve to get the intensity of the color subpixel we are interested in. Another option is to evaluate it over three color primaries instead of full spectral rendering. Using the latter approach makes it significantly easier to evaluate but at the cost of accuracy. Note that this form of rendering equation wouldn't allow us to account for spectral effects, such as fluorescence.

As stated above, the rendering equation in the form presented in Equation 1.16 accounts only for interaction between light and the surfaces it hits. However, in the real world, the light also interacts volumetrically with participating media. The most common example is the atmosphere, this interaction turns the sky blue. Other examples are liquids such as milk which has a white color thanks to the light scattering inside its volume. And even solid materials experience some level of light scattering. The most notable examples are skin, where longer wavelengths (red color) travel further into the material than others. Other examples are marble, wax, or plastic.

The participating media affects the light contribution by three processes as stated in Pharr et al. [2016]:

1. *Emission*: The participating volume adds intensity from light-emitting particles. For example, a fire emits light because of ionized air particles. As this effect isn't the primary focus of our work we will ignore it.

2. *Absorption*: The light intensity is reduced as light is absorbed by the volume. For example, a cloud of thick black smoke is absorbing a lot of light.

3. *Scattering*: The light might change its direction due to collision with a particle. For example, clouds have their distinct appearance because the light that enters the cloud volume is bounced multiple times.

To precisely describe the effects of the interaction between participating media and light we can use the following set of equations introduced by Kajiya and Herzen [1984]:

$$L(\mathbf{x}, \omega_o) = \int_0^s T_r(\mathbf{x} \leftrightarrow \mathbf{x_t}) \sigma_s(\mathbf{x_t}) L_i(\mathbf{x_t}, \omega_o) \mathrm{d}t + T_r(\mathbf{x} \leftrightarrow \mathbf{x_s}) L(\mathbf{x_s}, \omega_o) \tag{1.17}$$

$$\mathbf{x_t} = \mathbf{x} - \omega_o t \tag{1.17a}$$

$$T_r(\mathbf{x} \leftrightarrow \mathbf{x}') = \exp\left(-\int_x^{x'} \sigma_t(u) \mathrm{d}u\right) \tag{1.17b}$$

$$L_i(\mathbf{x}, \omega) = \int_\Omega p(\mathbf{x}, \omega_i, \omega_o) L(\mathbf{x}, \omega_i) \mathrm{d}\omega_i \tag{1.17c}$$

$$\sigma_t(\mathbf{x}) = \sigma_s(\mathbf{x}) + \sigma_a(\mathbf{x}) \tag{1.17d}$$

Note that this equation is simplified by omitting the emission term as well as the fact that all functions depend on wavelength, similar to Equation 1.16. Figure 1.3 illustrates the position of some points from the equation. The term $L(\mathbf{x}, \omega_o)$ is the radiance coming into the point $\mathbf{x}$ from direction ω_o. It consists of the radiance reflected from point $\mathbf{x_s}$ attenuated by the participating media. The attenuation between points $\mathbf{x_s}$ and $\mathbf{x}$ is expressed by the term $T_r(\mathbf{x} \leftrightarrow \mathbf{x_s})$. The overall radiance then consists also from the in-scattered light. The in-scattering

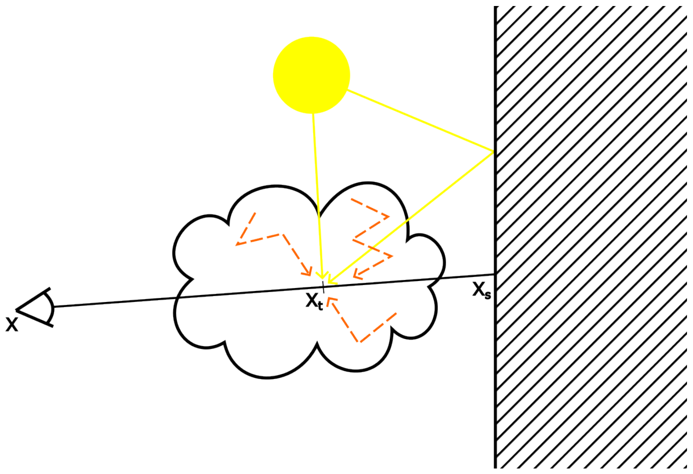


Figure 1.3: Multiple scattering inside a participating medium.

is expressed by the integral over the length s of the ray between points $\mathbf{x}$ and $\mathbf{x_s}$. Inside this integral, there is the attenuation term $T_r(\mathbf{x} \leftrightarrow \mathbf{x_t})$ again, but in this case, it expresses the attenuation between points $\mathbf{x}$ and $\mathbf{x_t}$. The point $\mathbf{x_t}$ is computed as described in Equation 1.17a where t is the variable of the integral from Equation 1.17. The amount of in-scattering is controlled by the scattering coefficient σ_s.

To evaluate all the in-scattered light we have to compute the radiance incoming from the volume $L_i(\mathbf{x_t}, \omega_o)$ which is described in Equation 1.17c. For every point $\mathbf{x_t}$ along the light ray, we have to integrate over all directions ω_i across a complete sphere Ω. Notice the difference to Equation 1.16, where we integrated only across a hemisphere. The probability that the light will be scattered into the direction ω_o is controlled by the distribution function $p(\mathbf{x}, \omega_i, \omega_o)$, which is called a *phase function*.

Equation 1.17b shows how the attenuation term T_r is computed. This term combines the effects of absorption and out-scattering. It describes how much the radiance is attenuated between points $\mathbf{x}$ and $\mathbf{x}'$ and it is controlled by the extinction coefficient σ_t. Note how we have to accumulate the extinction coefficient over the length of a ray. This can be easily computed for homogeneous media, but for inhomogeneous media, the computation is more difficult. As the extinction coefficient combines the effects of absorption and out-scattering it is simply a sum of absorption coefficient σ_a and scattering coefficient σ_s as stated in Equation 1.17d.

The volume properties are controlled by three parameters: scattering coefficient σ_s, absorption coefficient σ_a, and a phase function p. Values of these parameters can be constant across a volume (typical for liquids and some solids), in that case, we call the volume homogeneous, or the values might vary across the volume (most gaseous volumes like smoke or clouds or solid materials composed of smaller parts like color 3D prints), in that case, we call the volume inhomogeneous.

Phase function $p(\omega_i, \omega_o)$ is a distribution function that dictates the probability that the direction ω_i is scattered into the direction ω_o. As it is a distribution, it has to integrate over a whole sphere Ω to 1. The simplest phase function is the uniform phase function, which has equal probability for all directions. Another phase function is Rayleigh scattering, which depends on the wavelength and prefers forward and backward directions in opposition to perpendicular directions. The wavelength dependency in Rayleigh scattering causes the sky to be blue. The Rayleigh scattering is caused by smaller particles while larger particles cause Mie scattering. Mie scattering phase function is complicated and is often approximated by the Henyey-Greenstein phase function which has a single parameter g that controls whether the phase function is forward- or back-scattering. The Henyey-Greenstein function is described by Equation 1.18 as stated by Jarosz [2008]. It depends only on the angle between incoming and outgoing directions Θ and the anisotropy parameter g. Note that the phase function in Equation 1.17c together with the scattering coefficient in Equation 1.17 acts similarly to the BRDF from Equation 1.16.

$$p_{HG}(\Theta) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g\cos(\Theta)^{1.5})} \tag{1.18}$$

Absorption coefficient σ_a determines how much light intensity is irretrievably lost by volume absorption. Absorption coefficients usually depend on wavelength.

The *scattering coefficient* σ_s determines how much light is scattered inside the participating volume. The light can be scattered out which lowers the overall light intensity passing through the volume. Or the light might be scattered into the path which increases the intensity.

Another property that can be used to describe the volume is *albedo*. Albedo is a unitless ratio of scattering and extinction coefficients, which can be expressed as $\frac{\sigma_s}{\sigma_s+\sigma_a}$. This property describes the ratio of light being absorbed and light being scattered in each interaction. Materials with high albedo are particularly difficult to render as there are longer paths inside them with lots of scattering events.

Evaluating Equation 1.17 is computationally even more intensive than ray tracing as we have to evaluate the absorption and scattering coefficients in every position across the length of a light ray. This technique is known as ray marching. Given the computation intensity, there are attempts to estimate the volumetric interaction with machine learning approaches as presented in the next chapter.

2. Related Work

This chapter will include an overview of the previous work that is used in our thesis. These can be split into two categories: scattering prediction and representation networks. The work presented in the section on scattering prediction focuses on approximating light transport in participating media with neural networks. While the research on representation networks is mostly about networks that describe the volumetric properties of different scenes.

2.1 Scattering prediction

As stated in the previous chapter, the evaluation of Equation 1.17 is computationally intensive. To speed up the estimation of the integral, Kallweit et al. [2017] introduced a technique to compute the in-scattered radiance in clouds by using a neural network. Later, Rittig et al. [2021] adapted this technique to 3D printing application.

Kallweit et al. [2017] technique used a stencil with multiple scales that sampled a cloud volume around a point where the in-scattered radiance was computed. Also, the stencil was aligned with the oncoming light direction as seen in Figure 2.1. In each point of the stencil, scattering and absorption coefficients are stored. For the coarser levels of the stencil, the coefficients are averaged over the appropriate area.

The finer scales of the stencil help to focus the network on a fine detail around the point of interest while the more coarse scales of the stencil help with the approximation of in-scattered light from more distant areas. The stencil is then inputted into the prediction network that produces a number that describes the in-scattered radiance in the point of interest.

Architecture changes proposed by Rittig et al. [2021] include an axis-aligned stencil instead of a stencil that is aligned towards the light direction. The alignment isn't necessary as the stencils are centered around the point of interest and the lighting of the 3d print is assumed to be diffuse without any preferential direction. Another difference to the Kallweit et al. [2017] is that this network is used to predict the whole appearance of the volume rather than to approximate the indirect lightning contribution.

The architecture of the prediction network consists of multiple radiance prediction neural networks (RPNN) blocks where each block accepts a different stencil level. Each block also shares its weights with others which leads to better generalization. The network also benefits from the radial symmetry of the problem at hand, so the portions for each octet of the stencil are shared as well. The architecture of the network, as well as the relation between different stencil levels, are depicted in Figure 2.2.

As the intended application is in 3D printing, the stencil dimensions are linked to the voxel resolution of resin droplets produced by the 3D printer. These printers have a different resolution in the three spatial axes, this result is that stencils have nonuniform dimensions as well as all training data have nonuniform voxel dimensions.

The neural network introduced by this paper is directly used in our work and

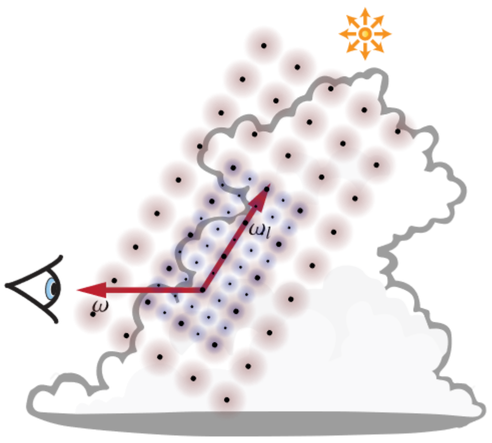


Figure 2.1: A stencil rotation and placement used by Kallweit et al. [2017]. Two stencil levels are shown here. Image taken directly from Kallweit et al. [2017].

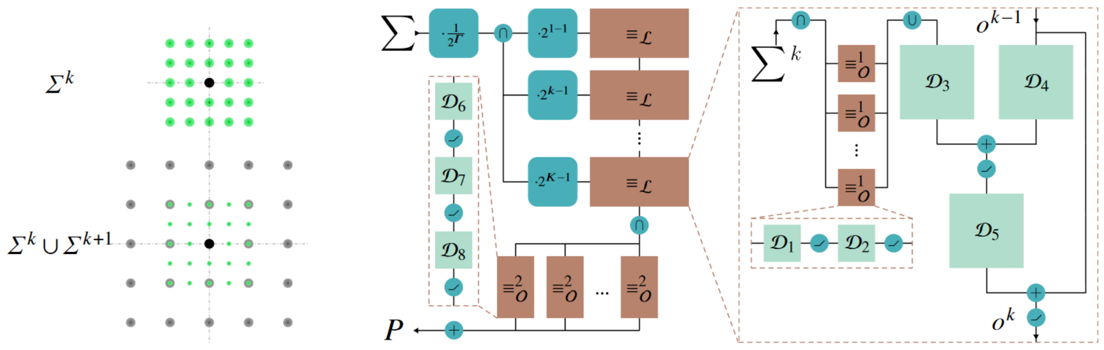


Figure 2.2: The architecture of the prediction network. The image on the left shows how different stencil levels sample the volume around the point of interest. The image in the middle shows an overview of the architecture of the network. $\mathcal{D}$ denotes dense layers with trainable weights, $\equiv$ blocks denote shared weights and round blocks denote mathematical operations. The image on the right shows the detail of one RPNN block. Image taken directly from Rittig et al. [2021].

it is referred to as the appearance prediction network. Our work focuses on the data representation that is being fed into the network. The original architecture used a *summed-area table* (SAT) (see Crow, Franklin C. [1984]) as its data structure to quickly evaluate an average value across multiple voxels for coarser stencil levels. This averaging corresponds to using a box-shaped kernel filter for blurring the finer stencil levels. We have completely switched the data structure for our representation network. This would in theory enable a differentiable scene representation.

2.2 Neural radiance fields

Neural radiance fields (NeRF) were introduced by Mildenhall et al. [2020]. And since then many authors improved upon it. For our purposes, the most notable improvement was Barron et al. [2021] which deals with aliasing artifacts for distant objects.

NeRF network utilizes a positional encoding of the input parameters. The usual approach is to use sine and cosine functions. However, Müller et al. [2021] proposed a triangle function for better performance as transcendental functions are notoriously slow to evaluate on a GPU. Also, Müller et al. [2022] proposed a different type of positional encoding that relies on spatial hashing functions which are even faster to evaluate.

Frequency-based positional encoding functions transform the input into output with more dimensions, where each corresponds to a different frequency. This allows the network to focus on higher-frequency detail. In the NeRF paper, the positional encoding had 5-dimensional input, 3 for spatial coordinates and 2 for direction. The output was a concatenation of the original input and the encoded values computed as in Equation 2.1. The number of frequencies used for encoding could be parameterized. The positional encoded input is fed to the MLP network that essentially stores the information about a scene in its weights.

$$\boldsymbol{\gamma}(\boldsymbol{p}) = \begin{pmatrix} \boldsymbol{p} \\ \sin\left(s^0\boldsymbol{p}\right) \\ \cos\left(s^0\boldsymbol{p}\right) \\ \sin\left(s^1\boldsymbol{p}\right) \\ \cos\left(s^1\boldsymbol{p}\right) \\ \sin\left(s^2\boldsymbol{p}\right) \\ \cos\left(s^2\boldsymbol{p}\right) \\ \dots \\ \sin\left(s^L\boldsymbol{p}\right) \\ \cos\left(s^L\boldsymbol{p}\right) \end{pmatrix} \tag{2.1}$$

Equation Equation 2.1 describes the positional encoding used by NeRFs, where γ is the positional encoding function, $\boldsymbol{p}$ is position (or more generally the variable that is being encoded) and s is a hyperparameter that specifies the spacing of frequencies and is always greater than one. Originally Mildenhall et al. [2020] used $s = 2$. L controls the length of the output vector and it is also a hyperparameter. Comparing this equation with Equation 1.2 we can see why these networks are sometimes called Fourier feature networks. The individual elements of the output vector are essentially terms of the sum from the discrete Fourier transform equation.

As noted in the NeRF paper, a similar technique is used in large language model transformer networks Vaswani et al. [2017]. However, in this application, we use positional encoding to provide a higher-frequency representation of the continuous space. The transformer networks use positional encoding to add information about the position of a token.

As we stated in section 1.2, usually the goal of machine learning is to provide good generalization capabilities. However, in neural radiance field application, the model is purposefully overfitted to the given scene. That means, that for each given scene, we have to fit a separate model. If we would like to represent a different scene we will have to fit the model again. This fact might limit potential applications as nontrivial time has to be spent when fitting the model. The time spent on training the neural network for the particular scene might be reduced by utilizing the meta-learning technique proposed by Tancik et al. [2021], which optimizes the initialization of a particular representation network to an average of many different scenes. This creates a starting point from where the convergence is faster than from a random initialization.

The advantage of using a representation neural network is that it provides differentiable scene representation. Or it can be used as an alternative to point cloud scene representation.

Tancik et al. [2020] showed that the frequencies used in positional encoding (Equation 2.1) can be drawn from a random distribution. Where the standard deviation of such distribution is much more important than its shape. Their work also showed the importance of positional encoding for representational MLP networks.

Barron et al. [2021] introduced a technique that deals with aliasing artifacts by sampling conical sections instead of points. The frequencies in positional encoding are weighted based on the approximation of a conical volume. We exploit this technique to produce scattering and absorption coefficients at different scales similar to how mipmaps work for textures, hence the name Mip-NeRF. The conical section volume is approximated by a multivariate Gaussian. The covariance matrix of the multivariate Gaussian is proportional to the shape and volume of the conical section. Given the covariance matrix and the mean we can compute the integrated positional encoding by the following equation:

$$\boldsymbol{\gamma}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \begin{pmatrix} \sin(s^0\boldsymbol{\mu}) \cdot \exp(-\frac{1}{2}\mathrm{diag}(\boldsymbol{\Sigma})s^0) \\ \cos(s^0\boldsymbol{\mu}) \cdot \exp(-\frac{1}{2}\mathrm{diag}(\boldsymbol{\Sigma})s^0) \\ \sin(s^1\boldsymbol{\mu}) \cdot \exp(-\frac{1}{2}\mathrm{diag}(\boldsymbol{\Sigma})s^2) \\ \cos(s^1\boldsymbol{\mu}) \cdot \exp(-\frac{1}{2}\mathrm{diag}(\boldsymbol{\Sigma})s^2) \\ \ldots \\ \sin(s^L\boldsymbol{\mu}) \cdot \exp(-\frac{1}{2}\mathrm{diag}(\boldsymbol{\Sigma})s^{2L}) \\ \cos(s^L\boldsymbol{\mu}) \cdot \exp(-\frac{1}{2}\mathrm{diag}(\boldsymbol{\Sigma})s^{2L}) \end{pmatrix} \tag{2.2}$$

Where γ is integrated positional encoding, $\boldsymbol{\mu}$ is the mean value of the position, and $\boldsymbol{\Sigma}$ is the covariance matrix of the multivariate Gaussian. The diagonal of the covariance matrix contains standard deviations of each input coordinate. Comparing integrated positional encoding (Equation 2.2) to the original positional

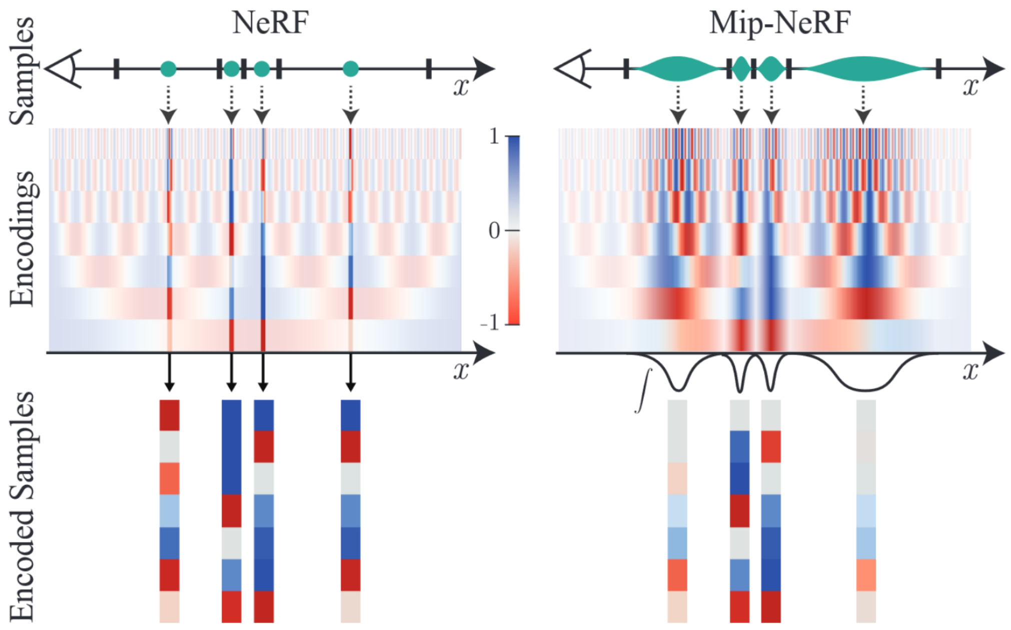


Figure 2.3: A figure illustrating the difference of positional encoding between NeRF and Mip-NeRF. Image taken directly from Barron et al. [2021].

encoding (Equation 2.1), we can see that the only difference is the attenuation of vector components by the exponential factor. The attenuation is scaled by the standard deviation of the multivariate Gaussian and the frequency. Amplitudes of higher frequencies are clipped more with higher values of standard deviation. This equation can be seen as a direct application of the convolution of the Fourier transform of the original input with the Gaussian in the frequency domain, as presented in section 1.1.

The difference between the two positional encoding functions is illustrated in Figure 2.3. The figure shows a one-dimensional example of the two different encoding functions. Notice that the higher frequencies in larger conical frustums are attenuated compared to NeRF point samples. Also, note that the attenuation is dependent on the size of the frustums.

2.2.1 Training

The training of the NeRF network is done using a dataset with multiple different images of the same scene. For each image, there is a camera position and camera direction, so that ray origin points and ray directions can be computed for every pixel in the image. Using a ray marching technique the output of the NeRF network is sampled along each ray. The opacity and color information are computed and compared to the actual value of the pixel. It is important that the ray marching algorithm is differentiable so that a gradient descent step can be made.

As stated above, Mip-NeRF uses cones instead of rays with conical volume samples instead of point samples. This means that the further away we are from the camera, the larger the conical volume is, which in turn means that the high frequencies in the positional encoding function (Equation 2.1) are attenuated.

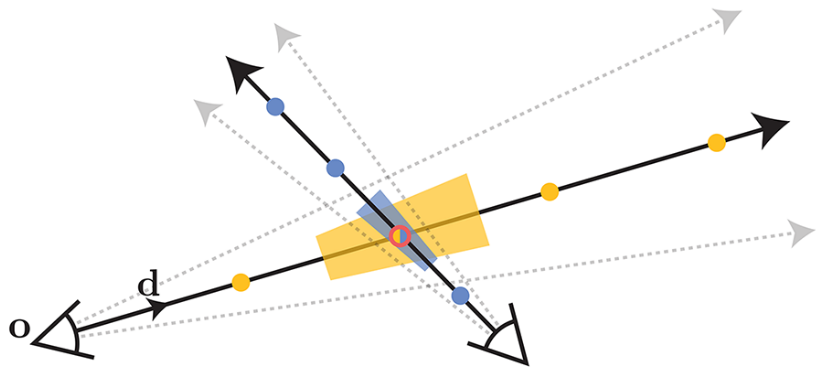


Figure 2.4: A figure illustrating sampling techniques used by NeRF and Mip-NeRF networks. The points represent sampling utilized by NeRF, while the trapezoids represent sampling utilized by Mip-NeRF. Image taken directly from Barron et al. [2021].

This ensures that the underlying MLP doesn't hallucinate high-frequency output when there shouldn't be any, greatly reducing aliasing. Other than the difference in sampling, the training approach is similar to the NeRF.

Figure 2.4 shows two sampling techniques used by NeRF and Mip-NeRF. It shows two viewpoints that would correspond to two different images during training. The dots show sample points of NeRF, while the conical frustums show sample areas of Mip-NeRF.

3. Method

Our goal is to replace the data structure used by the original appearance prediction network by Rittig et al. [2021]. To reiterate, summed-area tables were used originally. We are aiming to replace it with the representation network inspired by Mip-NeRF introduced by Barron et al. [2021]. For that reason, we conducted a series of tests that would evaluate the performance of such a representation network.

At first, we attempted to represent simple two-dimensional images. Next, we focused on representing sets of images, where each set consists of the same image that is blurred by increasing amounts. After these experiments, we attempted to represent volumes. We designed a volumetric representation network and tested it on volumes with different complexity. The last step was connecting the representation and the appearance prediction networks in such a way that we could predict the appearance of a 3D print, thus completing our goal of substituting the data structure of the prediction network. Our overall pipeline is illustrated in Figure 3.1 along with the training loop and the connection between the two networks.

This chapter provides details about the architecture of these networks and describes the connection between the two networks.

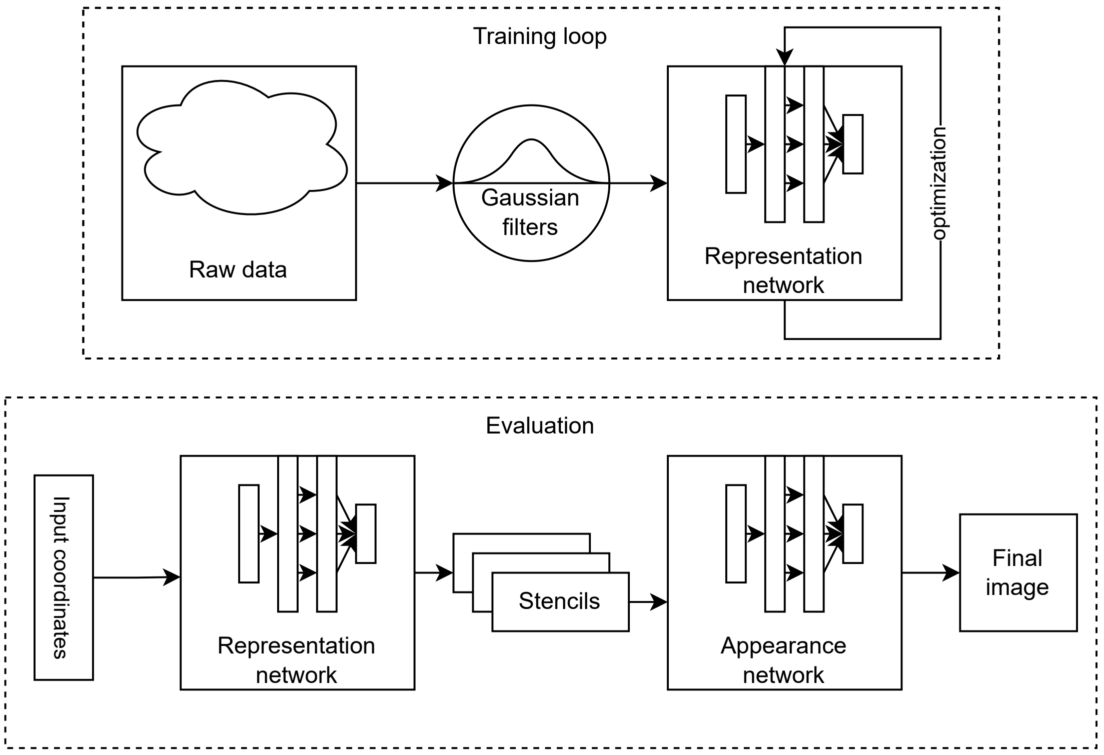


Figure 3.1: The final pipeline, where the representation and the appearance prediction networks are connected.

3.1 Representation network

3.1.1 2D image representation

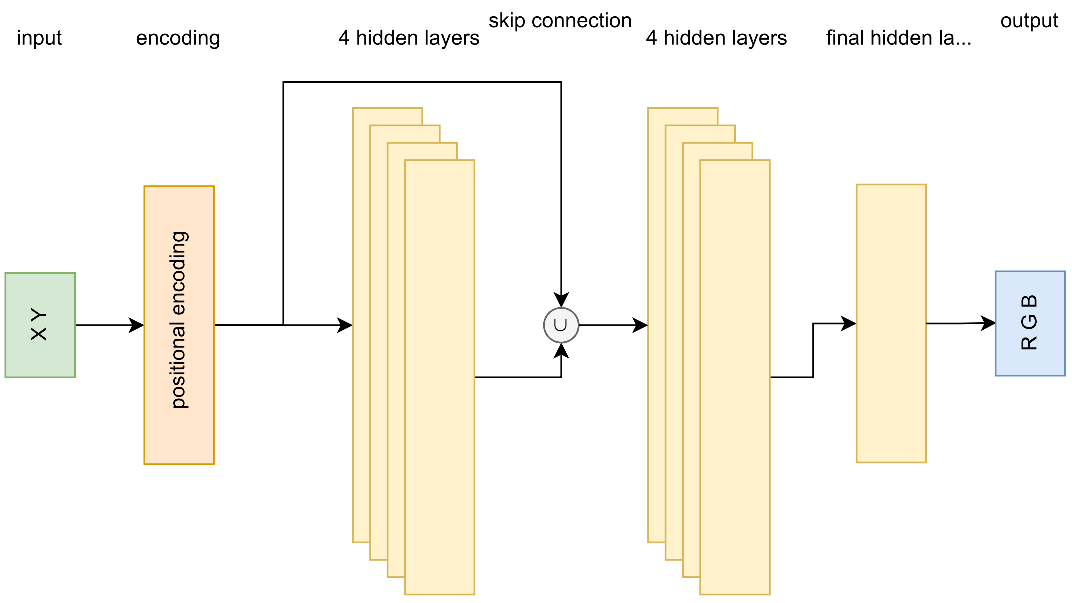


Figure 3.2: Architecture of a network for single image representation. Note that the group of 4 hidden layers might repeat based on the hyperparameters.

To evaluate the feasibility of our approach we decided to first experiment with representing 2D images. The architecture of these networks was similar to the architecture of NeRF by Mildenhall et al. [2020]. Unlike in the upcoming example, positional encoding wasn't attenuated as this isn't needed when fitting a single 2D image.

Architecture The input to the network is a two-dimensional point whose values are rescaled to lie between 0 and 1. This normalization of input coordinates helps with the training process as it constrains the gradient magnitude to a meaningful level. Our earlier experiments showed that using non-normalized coordinates had poor results. The two dimensions are then processed by positional encoding as described in Equation 2.1. The output of the encoding is fed into an MLP network with skip connections, which outputs three numbers. Each number corresponds to one of the three color channels.

The network architecture is controlled via a set of hyperparameters. Two hyperparameters of the positional encoding function, where the first controls the size of the output vector while the other is the scaling parameter s in the Equation 2.1. The parameter s of the positional encoding influences how spread apart the frequencies will be. Figure 3.2 illustrates the architecture of this simple network.

Another set of hyperparameters controls the shape of the MLP network. We can control the dimension of the first set of dense layers as well as their count. The skip connections are realized as a concatenation of the positional encoded inputs with the output of every fourth layer. After the first set of fully connected layers, there is one dense layer that has also an adjustable dimension and connects directly to the output layer which has three outputs for the red, the green, and the blue channel. The activation function for all layers is ReLU (Equation 1.12).

Training The images we used for training were downscaled to a resolution of 128 by 128 pixels. The training process consists of randomizing the pixel positions of the target image and feeding them into the network along with their target pixel values. This process is repeated for as many epochs as requested. We use the MSE as a loss function and PSNR as a metric. The learning rate is decayed every epoch by using the cosine decay formula Equation 1.15, where the maximal learning rate ϵ_{max} is variable and minimal learning rate ϵ_{min} is fixed to be $\frac{1}{100}$ of the ϵ_{max}.

The represented image is rendered by generating a list of ordered positions that correspond to appropriate pixel coordinates. We use these positions to evaluate the representation network and obtain the resulting image.

3.1.2 Representation of a set of blurred images

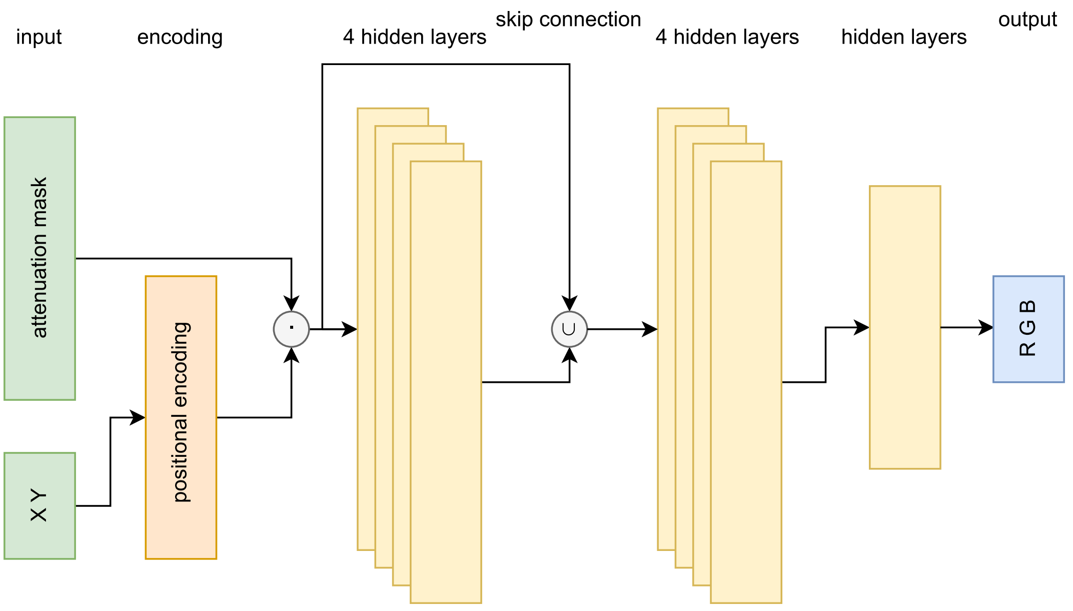


Figure 3.3: Architecture of a network for single image representation. Note that the group of 4 hidden layers might repeat based on the hyperparameters.

The next step in evaluating the feasibility of our approach was to test the representation network with attenuated positional encoding as described by Equation 2.2. The architecture of this representation network was similar to the simple one from the previous experiment. The only change was the attenuated positional encoding.

Architecture The network has the same number of hyperparameters as the MLP design after the position encoding step is the same as in the previous case.

In fact, the network is almost identical to the one used for single-image representation. The only difference is the attenuation mask. This fact can be seen in Figure 3.3, which shows the architecture of this representation network.

The inputs to the network are positions and the attenuation mask, which can be computed based on the size of the Gaussian kernel used to blur the images. The size of such kernel is determined by its standard deviation, which for multivariate Gaussian can be found on the diagonal of the covariance matrix. More precisely, the square of standard deviation is on the diagonal of the matrix, which appears directly in Equation 2.2. Therefore, for each blur level, we have a single attenuation mask.

Training For training, we attempted several different strategies. Ordering the images from coarsest details to finest while maintaining a similar strategy as with single image representation, that is, randomizing input pixel positions and feeding them to the network for training. The idea behind this was that the convergence of the training with coarser detail is faster so the network would spend less time training on the coarser levels. However, in practice, the results of this strategy were suboptimal. Another training strategy was interleaving the blur levels so that the level with the finest details is in-between different levels, while still ordering the blur levels from coarsest to finest. The final strategy was to sample positions as well as levels randomly, this yielded the best results in terms of speed of convergence. This was our chosen strategy that we used in the rest of our work.

3.1.3 Volume representation

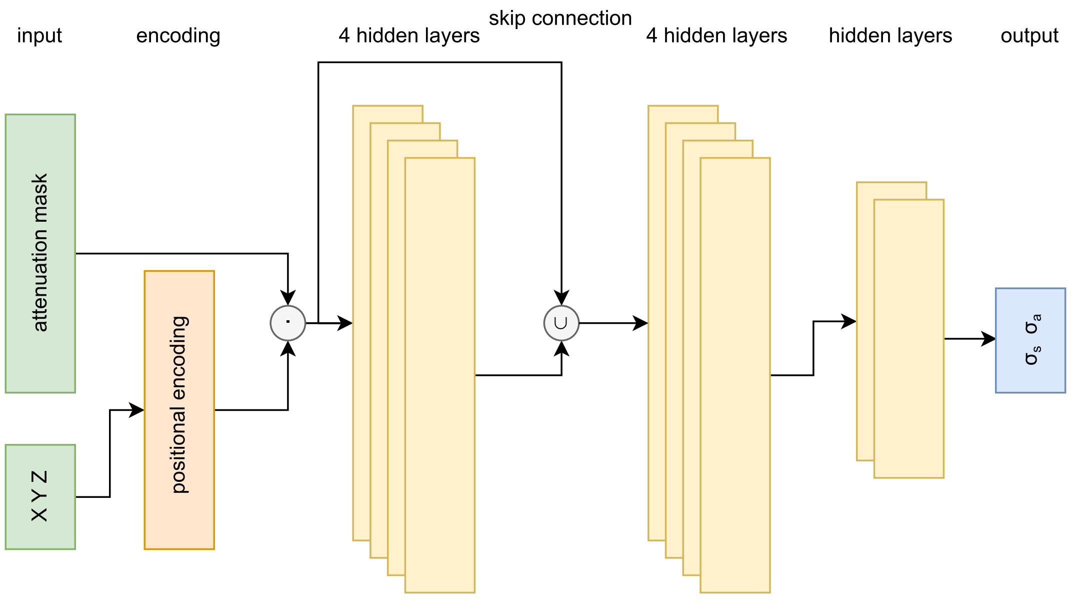


Figure 3.4: Architecture of volumetric representation network. Note that the group of 4 hidden layers might repeat based on the hyperparameters.

The final step was to prepare a volumetric representation network. Our volumetric representation network has a similar architecture to the network presented in Barron et al. [2021]. However, the input is a three-dimensional position normalized between 0 and 1. As with the 2D images, the normalization helps with containing the gradients to a sensible level. The overall architecture is very similar to the previous case of representing sets of blurred images.

Architecture The input is three spatial coordinates that are positionally encoded via Equation 2.1. Along with these three input coordinates, the network is also fed with an attenuation mask. This mask is computed as the attenuation part of Equation 2.2 in a comparable way as in the previous case. Because we use the Gaussian kernels with known standard deviations to blur the input, we can easily evaluate the equation for each blur level.

The output of positional encoding is inputted to an MLP with skip connections. The skip connections are implemented in the same way as described in the previous two networks. Every fourth layer has the original MLP input concatenated with its inputs.

Before the final output layer, there are hidden layers with different widths. This width is smaller than the width of the layers in the preceding MLP. There are no skip connections in this final hidden layers segment. The output of the final hidden layer is connected to the output layer, which outputs 2 values, the scattering and absorption coefficients that are also normalized. We have found out that keeping the output of the network between 0 and 1 is beneficial for the overall performance.

To facilitate this output normalization we simply found the highest value for each channel and then used a mapping function that transforms the full range into the smaller range. The mapping function is a simple linear rescaling of the original values. However, instead of using a range of 0 to 1, we have decided to use an even smaller range, this was inspired by the label smoothing technique described in Goodfellow et al. [2016]. In our implementation, we are using an interval $[0.05, 0.95]$. During the interpretation of the output of the representation network, we do the same process but reversed.

Overall, the hyperparameters are the same as in previous cases with the addition of configurable depth of the final hidden layers segment: The dimension of the positional encoding and also the scaling of frequencies, the depth of the first block of hidden layers, as well as the size of each layer, and the count of final hidden layers, as well as their width.

Figure 3.4 shows our architecture. It is the same architecture as the one described in subsection 3.1.2 with the inputs being three-dimensional and the outputs being two-dimensional. Another difference is the configurable depth of the final hidden layer segment. Compare it to Figure 3.3.

Training The data we are trying to represent is the output of a color 3D printing device. These devices can deposit small droplets of different materials to form the print as opposed to regular 3D printers, which deposit only a single material. Also, the droplets are tiny to enable color mixing. The properties of materials used in color 3D printing are well-defined. Table 3.1 contains these materials and their scattering and absorption coefficients. The table also contains an entry for

Table 3.1: Materials used in 3D printers and their coefficients.

	Red		Green		Blue	
Material	σ_s	σ_a	σ_s	σ_a	σ_s	σ_a
Cyan	0.45	8.55	3.15	1.35	7.35	0.15
Magenta	2.45	0.05	0.30	2.70	9.00	1.00
Yellow	2.24	0.01	3.73	0.02	2.85	16.15
Black	1.75	3.25	1.93	3.58	2.28	4.23
White	5.99	0.01	9.00	0.003	23.98	0.02
Air	0.00	0.00	0.00	0.00	0.00	0.00

an air material, which is used in the data when there is no material deposited by the 3D printer. Note that the air material has its scattering and absorption coefficients set to zero. That means the air material doesn't interact with light, which is a reasonable assumption for the scales used in 3D printing.

The values in Table 3.1 and the phase function used are the same as the values measured by Elek et al. [2017][1]. Crucially, they are the same values as Rittig et al. [2021] used for their prediction network. The phase function of the materials might be different for each material. However, we can assume an average phase function that would be the same for every material. For that reason, we approximate the phase function as the Henyey-Greenstein function with an anisotropy coefficient $g = 0.4$, see Equation 1.18. That is the same phase function as Elek et al. [2017] used in their work.

The challenge in representing this kind of data is its noisiness. As with traditional printing, to create a desired color a mixture of materials must be used and the small droplets of different materials are deposited next to each other. This creates high-frequency features that are very difficult to represent with the neural network.

Each volume was represented by three different representation networks. One for red, one for green, and one for blue channel. The training data was preprocessed by blurring into nine different levels. These were required by the appearance prediction network. We used Gaussian blur as opposed to Rittig et al. [2021], who used a box-shaped kernel. However, the Gaussian kernel enables us to use the attenuated positional encoding from Equation 2.2 as the attenuation directly follows from the convolution in the frequency domain. Using a different kernel for blurring would require a different function to be used as a factor. The kernel size of each blur level was chosen to match the required blur level of the appearance prediction network. We achieved this by selecting the standard deviation of the Gaussian to be half of the box kernel radius. The relation between these two kernels with this ratio of sizes is illustrated in Figure 1.1.

The sampling of the voxel grid was done in such a way that for each voxel in each blur level, we produced a single sample that corresponds to the middle of that voxel. In addition to these "center" samples we added extra samples across a voxel for the first blur level (without blur). This super-sampling option is a hyperparameter of our representation network. In theory, multiple samples should introduce more detail and higher frequencies into the training data for the

[1]Elek et al. [2017] state the values in terms of albedo and extinction coefficients. We have recomputed their values to scattering and absorption coefficients.

finest level. However, it doubles the training time as there are twice as many samples to train on. The increased precision on the finest level is required by the fact, that the representation network might get evaluated at different resolutions than it was trained on and these positions might not end up in the middle of a voxel.

3.2 Appearance prediction network

As stated in section 2.1, we use the prediction network proposed by Rittig et al. [2021]. We swap the summed-area table data structure for the representation network mentioned above.

The first step in connecting the two networks is to compute the visible outer voxels of the volume. For this purpose, we use two approaches depending on the volume at hand. The first one utilizes an algorithm used by Rittig et al. [2021], that iteratively ray-marches from outside of the volume towards the center and seeks material that is different from the air material. We use this technique when working with an unknown volume. The other approach relies on the fact that we know the positions where the volume is occupied by the material, which is the case for simple, generated volumes. Once these positions are found, we use them to generate stencils for the prediction network.

Next, attenuation masks for each stencil blur level are computed. Then for each position found by the previously described search algorithm, we generate positions according to stencils used by the prediction network. These positions are spaced out for each blur level by a factor of 2. For positions that lie outside of the trained area and thus their value would be outside of the normalized range of 0 to 1, we decided to clamp them to the said range. All data that were used contained a small buffer of air between the actual voxels with materials and the edge of the data array. Also, for each blur level, we override voxels on the edge with values for air material (zeros). For that reason, the clamping of positions shouldn't introduce much distortion as the position outside of the data array is assumed to be air.

Once we have all positions, we group them with the attenuation mask and pass them into the representation network. The representation network outputs stencils with scattering and absorption coefficients that we pass to the appearance prediction network. The prediction network outputs a single number that corresponds to the pixel intensity at that position.

We repeat said process for each color channel to get the final result. Theoretically, there can be as many color channels as needed, for example, if we were to use spectral data then we would be able to predict the appearance based on spectral rendering. However, we used a trichromatic approach with red, green, and blue channels.

The major difference between our representation and the original data structure used by Rittig et al. [2021] was that we used the Gaussian kernel to average the data for coarser stencil levels, while they used the box kernel. The standard deviation of the Gaussian kernel was half of the box kernel radius. The relation between these two kernels is illustrated in Figure 1.1. Due to this difference, we decided to first test our approach by evaluating the prediction network on the same data that was used to train the representation network. By querying the processed data array at appropriate positions we constructed stencils for the appearance prediction network. If the requested position was outside of the data array, we clamped the position to the nearest edge to preserve the behavior of the representation network query.

4. Implementation

We have implemented the discussed solution from chapter 3 using the Tensorflow 2 framework. To visualize the results we used Jupyter notebooks. We have utilized Docker to bundle the whole software solution together into a container. To prepare the setup of the container we used Docker Compose. The sources for our solutions were coded in Python programming language.

The original appearance prediction network by Rittig et al. [2021] was implemented in Tensorflow 1. Even though the conversion between Tensorflow versions might be possible, we didn't succeed. For that reason, we distribute the original docker container of Rittig et al. [2021] complemented with our script that reads the outputs of the representation network and feeds the appearance prediction network. To obtain the appearance prediction, a set of stencils has to be generated using our Docker container. Once these stencils are generated, they can be loaded into the appearance prediction network and the surface appearance is computed.

Most of our results are presented in Jupyter notebooks. For instance, the implementation of the image representing networks (presented in sections 3.1.1 and 3.1.2) is done in Jupyter notebooks as well as simple volumetric representation. While the volumetric representation of complex volumes (presented in section 3.1.3) is handled in regular Python scripts.

The Tensorflow framework provides an implementation of various machine learning algorithms. For instance, the Adam optimizer, the cosine learning rate decay, the dense layer, the mean squared error loss function, and many more. It also includes a framework that manages the training process with automatic differentiation and application of the optimizer. Another useful interface included in Tensorflow is `tf.data`, which can be used to generate, shuffle, and batch data. The API of the Tensorflow package is provided in the Python programming language, hence it was also the language of choice for our implementation.

As stated in the previous paragraph, for the random sampling of the generated data we used the `tf.data` interface. This interface has a function to randomize the order of the data. The major bottleneck we encountered was the fact that the shuffling was happening on the CPU. This in theory should enable better performance as the main training is done on the GPU and the otherwise idle CPU would be used to prepare the data for the next batch. However, we have found out that for the best results, we have to use very large shuffle buffers, ideally as large as the whole dataset. This prevents any parallel operation whatsoever, hurting the performance massively.

We used different hardware configurations to run our experiments. For small-scale training and 2D image representation a computer with AMD Ryzen 5 3600 CPU and NVIDIA RTX 2060 GPU was used. Larger experiments, especially volumetric representation network training ran on a compute server. We used a server with Intel Xeon E5-2680 v3 CPU with 256 GB of RAM, and NVIDIA Titan RTX GPU with 24 GB of video memory.

Our implementation is provided as an electronic attachment to this thesis, see Attachment A.1. The documentation for the package is also attached to the thesis, for that see Attachment A.2.

5. Results

This chapter reviews the results of our representation network. First, we look at representing 2D images, where we evaluate the training progression and the impact of the network capacity on the representation quality. In the next section, we analyze the representation of a set of blurred images. This case should take advantage of our Mip-NeRF-inspired architecture. The most important results are related to the representation of volumetric data. We inspect the representation of simple synthetic volumes as well as one more complex volume. The last section presents the results of the appearance prediction network and compares the outputs that use the representation network with outputs that don't use it.

5.1 Representation of 2D images

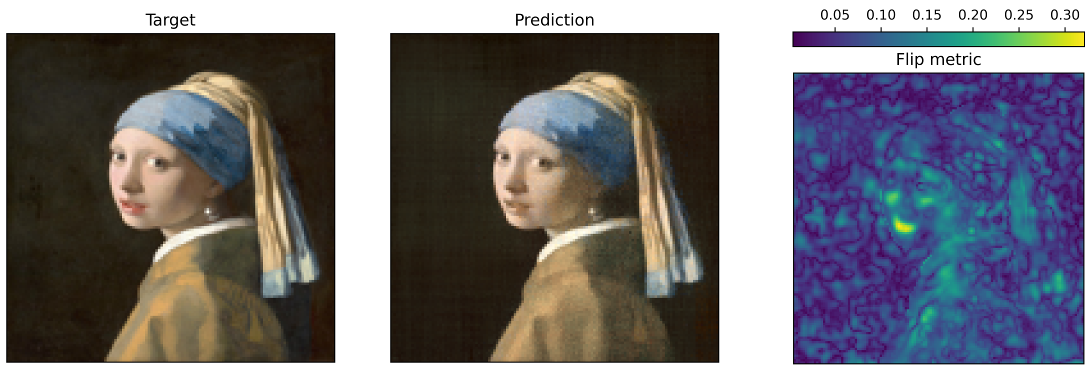


Figure 5.1: Result of 2D image representation network. The left image is the target, the middle image is the representation network output and the image on the right is the flip metric.

Table 5.1: Parameters of representation network used to represent a single image.

Hidden layers	8+1
Width of hidden layers	48
Width of final hidden layer	16
Positional encoding scale	2.0
Positional encoding size	16
Epochs	500
Initial learning rate	$5 \cdot 10^{-5}$
Batch size	64
Trainable parameters	23 683

For the representation of 2D images we used the network described in subsection 3.1.1. In this section, we will review the results of this representation for one particular image.

Figure 5.1 shows the result of the representation network. The first image from the left is the desired target image, the second image shows the representation network prediction and the final image on the right is the flip metric. The

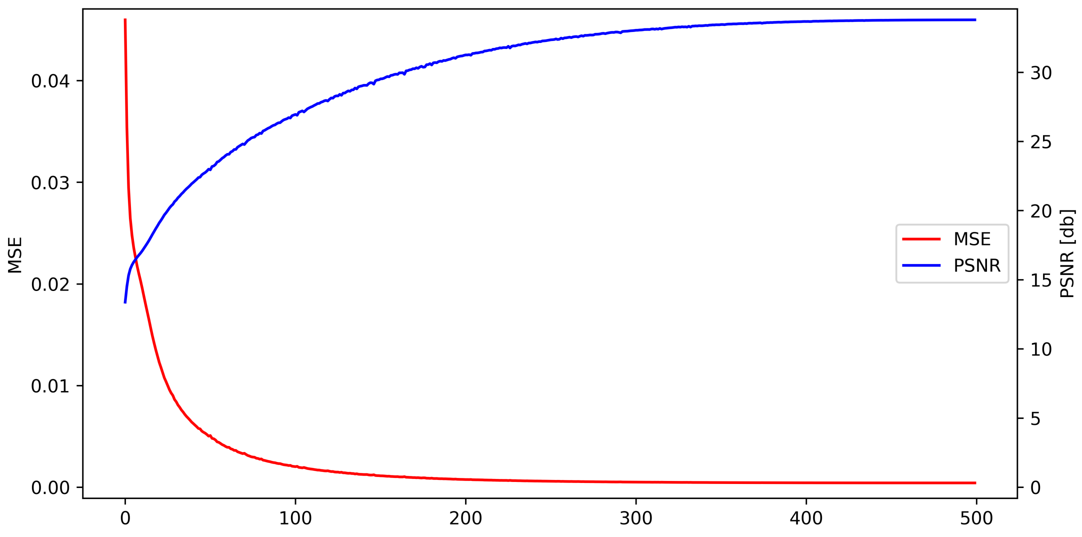


Figure 5.2: Training progress of a representation network of a single image. The blue line shows how the PSNR evolved over the training period, the red red line shows the training loss.

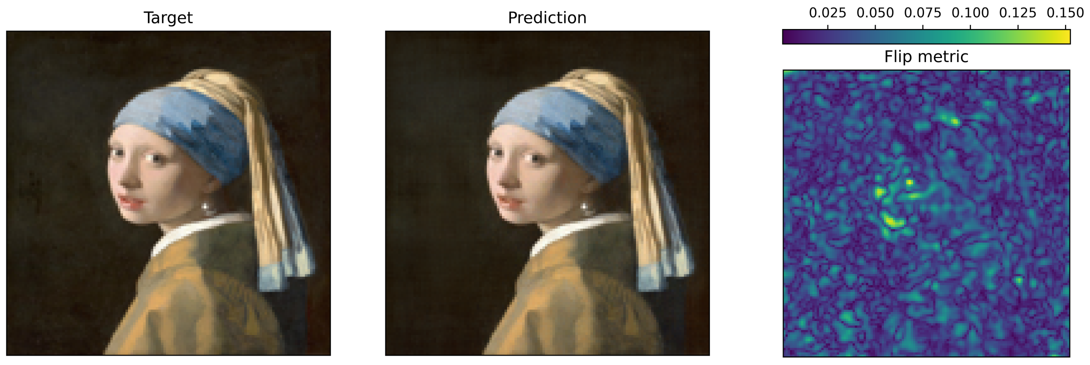


Figure 5.3: Result of the larger 2D image representation network. The left image is the target, the middle image is the representation network output and the image on the right is the flip metric.

Table 5.2: Parameters of the larger representation network used to represent a single image.

Hidden layers	12+1
Width of hidden layers	64
Width of final hidden layer	48
Positional encoding scale	1.75
Positional encoding size	16
Epochs	500
Initial learning rate	$5 \cdot 10^{-5}$
Batch size	64
Trainable parameters	61 763

flip metric was introduced by Andersson et al. [2020] and it helps to highlight perceivable differences between the two images.

The overall representation of an image is reasonably good. High-frequency details, like edges, are well preserved. However, the cloudiness in the dark portion of the original image is missing in the representation. Probably the biggest artifact is that the red color of the lips is missing and generally the predicted image seems to be less saturated in color.

Figure 5.2 illustrates the training process, the two curves are the evolution of loss and PSNR over training epochs. We can see that during training there weren't any anomalies that would suggest an incorrectly chosen learning rate. The final PSNR after 500 epochs was approximately 33.8 db. The time required for each epoch with NVIDIA RTX 2060 GPU and AMD Ryzen 5 3600 CPU was approximately 5 s, and overall the training took about 44 minutes.

Table 5.1 contains the values for hyperparameters of the representation network and parameters used for training. As stated in the previous chapter, the target image had a resolution of 128 by 128 pixels with 3 color channels. We were using a 32-bit floating point per channel, which means that in total we would need 49 152 floating point numbers. However, the representation network is comprised of 23 683 32-bit floating point parameters, thus providing a certain compression effect.

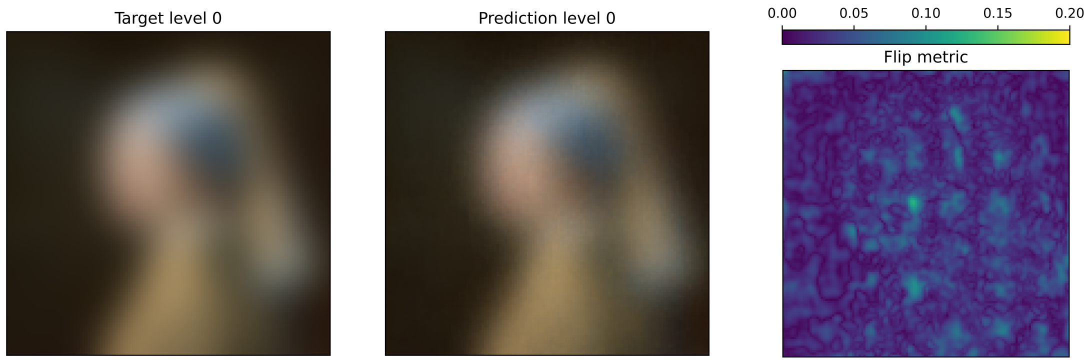
By sacrificing the compression effect, we can obtain better performance by increasing the size of the representation network. Figure 5.3 shows the results of the representation network with higher capacity, while Table 5.2 contains its parameters. As you can see, the overall quality of the representation is much better than in the case of the smaller network. The most noticeable artifacts disappeared and the discrepancies were reduced mostly to the facial area. Additionally, the saturation of the image is improved over the prediction of the smaller network even though the redness of the lips is still missing but to a lesser extent. The flip metric also peaks at about half of the original maximum. The PSNR for this representation network was about 41.0 db and the time it took to train the network was approximately 55 minutes on the same machine as in the previous case.

Our package that is provided with this thesis, Attachment A.1, contains more examples of single-image representation networks. Those are available in a Jupyter notebook called `2D Images.ipynb` in the notebooks directory.

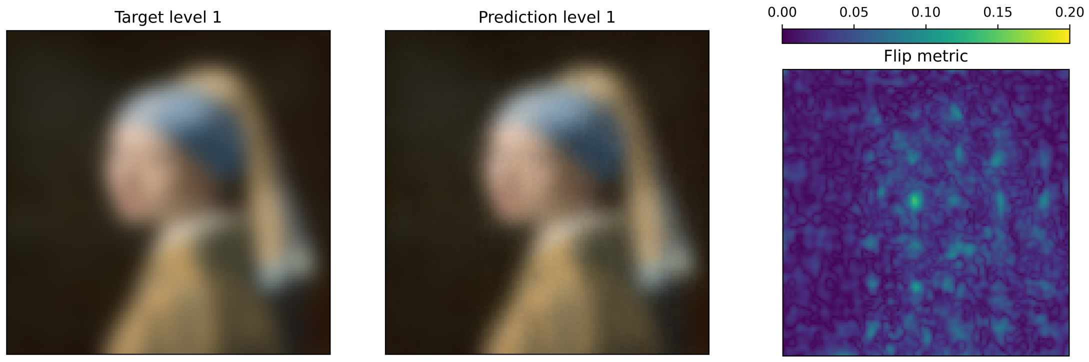
5.2 Representation of a set of blurred image

In this section we will take a look at representing multiple levels of detail with a single network. The image, that we are representing, was blurred by five different Gaussian kernels (the first kernel had a standard deviation of zero, so the blurred image is the same as the source), resulting in five levels of detail. The sizes of these kernels are given in Table 5.3. Figure 5.4 shows the results of a representation network trained on this set of blurred images with frequency attenuation. Each level of detail has its image triplet, where the leftmost image corresponds to the output of the network, the middle image is the target and the rightmost is the flip metric. The flip metric images are normalized to have the same range.

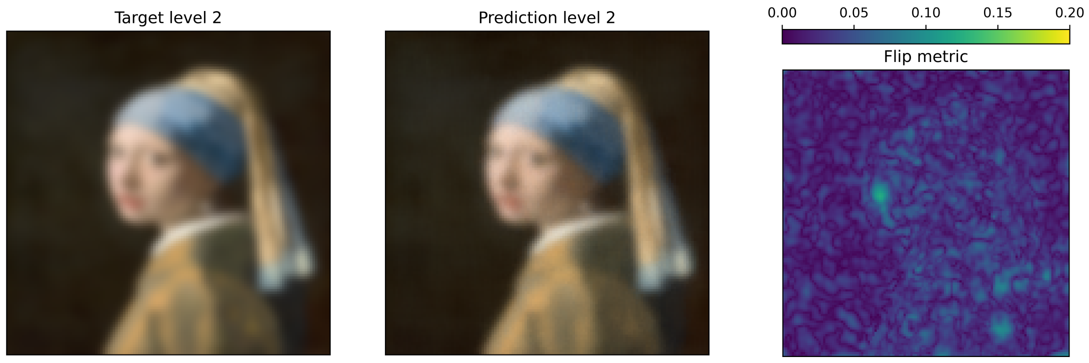
The images without fine details were represented quite well and even finely



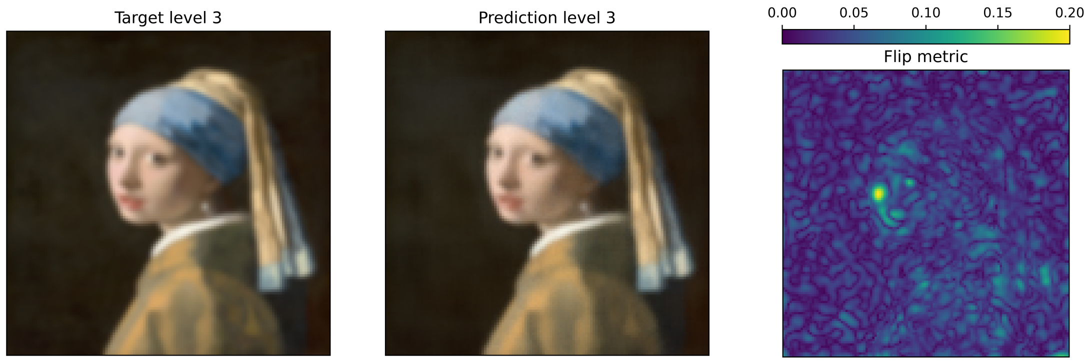
(a)



(b)

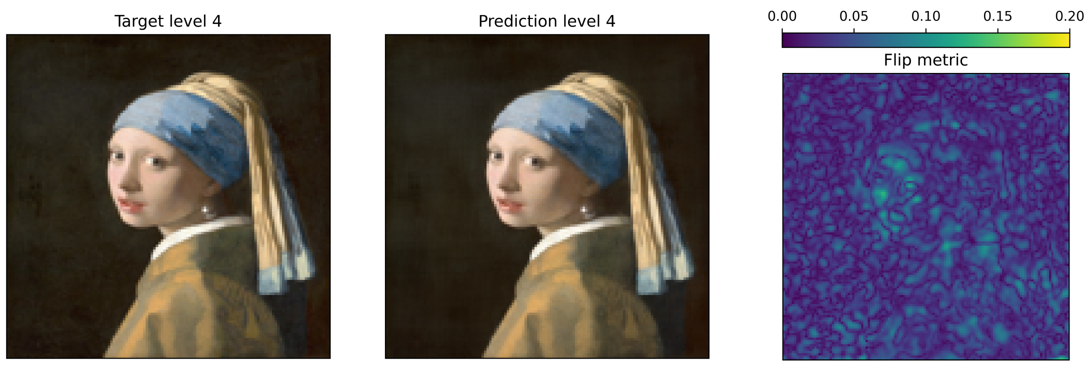


(c)



(d)

Figure 5.4: Result of a representation network for a set of blurred 2D images.



(e)

Figure 5.4: Result of a representation network for a set of blurred 2D images.

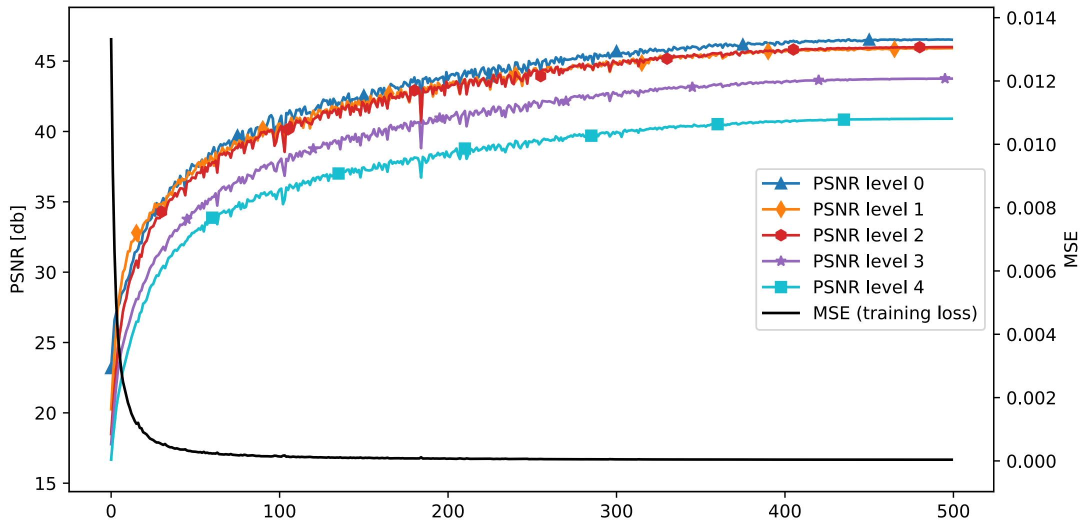


Figure 5.5: The evolution of PSNR of individual levels and training loss over the training period.

detailed images were represented well with all high-frequency details present. Even though we used a network with the same number of trainable parameters as in the case of the bigger model of single-image representation, the results for the most detailed level are almost the same. But this time the representation network can correctly represent even the blurred versions of the target image. The difference in the PSNR between the finest detailed level and the representation from the previous case is only 0.07 db as evidenced by Table 5.4 which contains PSNR values for individual levels. Also, the flip metric looks qualitatively similar.

Figure 5.5 shows the evolution of the PSNR metric over the training period as well as the loss function. Comparing this figure with Figure 5.2, the PSNR values seem to be more noisy, so perhaps the learning rate was set a little bit too high.

The network architecture is described in subsection 3.1.2 and Table 5.4 contains the values of hyperparameters used to obtain these results. The results shown in Figure 5.4 were obtained after training for 500 epochs. As there was five times more data than in the case of a single image representation, each epoch took more time. However, the additional time needed wasn't five times longer.

The time required for each epoch with NVIDIA RTX 2060 GPU and AMD Ryzen 5 3600 CPU was approximately 20 s, so overall the training took about 173 minutes. Note that for each epoch the representation network was evaluated to compute the PSNRs of each level, which negatively impacted the performance.

Our package that is provided with this thesis, Attachment A.1, contains more examples of this type of representation network. Those are available in a Jupyter notebook called `2D Blurred Images.ipynb` in the notebooks directory.

Table 5.3: Standard deviations (SD) of Gaussian kernels used to obtain target images in Figure 5.4.

Level	SD
4 (finest)	0
3	1
2	2
1	4
0 (coarsest)	8

Table 5.4: Values of PSNR for individual blur level. Ordered from finest details to coarsest.

Level	PSNR [db]
4 (finest)	40.9
3	43.8
2	46.0
1	45.9
0 (coarsest)	46.5

Table 5.5: Parameters of representation network used to represent multiple level of details of an image.

Hidden layers	12+1
Width of hidden layers	64
Width of final hidden layer	48
Positional encoding scale	1.75
Positional encoding size	16
Epochs	500
Initial learning rate	$5 \cdot 10^{-5}$
Batch size	64
Trainable parameters	61 763

5.3 Representing volume

We have chosen to represent volumes with varying complexity and we can divide them into two groups. The first group consists of simple volumes, which we generated ourselves. Another group is volumes used by Rittig et al. [2021] to train their appearance prediction network. These volumes are more complex than the simple ones and contain a lot of small features which proved difficult to represent to our representation network. From the second group, we selected one volume to examine.

5.3.1 Simple volumes

The simple volumes were generated as a slab with a pattern in two dimensions that is extruded into the vertical dimension. These patterns were chosen to give us hints about the representation network performance in different conditions and they can be seen in Figures 5.6 to 5.9. The image on the left represents a top-down view and the image on the right is the vertical cut through the volume, indicated by the red line. The printer materials are represented with their respective colors and the air material is represented by gray color. As mentioned before, all volumes have a small area on the outside filled with air material.

Table 5.6: Parameters of representation network used to represent simple volumes.

Hidden layers	8+1
Width of hidden layers	128
Width of final hidden layer	64
Positional encoding scale	1.20
Positional encoding size	24
Epochs	15
Initial learning rate	$5 \cdot 10^{-4}$
Batch size	64
Trainable parameters	161 730

Table 5.7: Standard deviations (SD) of Gaussian kernels used to create training data for volumetric representation network for simple volumes. The size is proportional to the size of a single voxel.

Level	SD
0 (finest)	0
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8 (coarsest)	128

The volumetric representation network architecture is described in Chapter 3.1.3. The parameters of this architecture used for simple volume are in Table 5.6. The volumes were blurred to nine different levels, which is the number of levels that the appearance prediction network expects. The sizes of Gaussian kernels are in Table 5.3. As it is sufficient to show only one of the three color channels to understand the quality of representation, we have trained representation networks only for the red color channel.

Super-sampling The effect of super-sampling can be illustrated on black and white edge volume. We can observe how the network responds to the changes in the target data. To compare the difference, we have trained two networks, one that samples just one sample per voxel for the first (finest) level and one that samples eight additional samples located near the corners of the voxel.

Figures 5.10 and 5.11 show the difference between super-sampling and no super-sampling. Figure 5.10 is a scatter plot of all values for scattering and absorption coefficients of the first level of the volume. The black dots correspond to the target values, all of which are located at three distinct positions, which stand for three materials (black, white, and air). The red dots correspond to the prediction by our representation networks, these aren't located at a single point but are spread around the target points. The size of this spread is larger for the network trained without super-sampling, this indicates that the super-sampling increases the accuracy of the representation network.

Figure 5.11 is a one-dimensional cut through the volume, focusing on the area of the edge where the properties change. The cut consists of four voxels

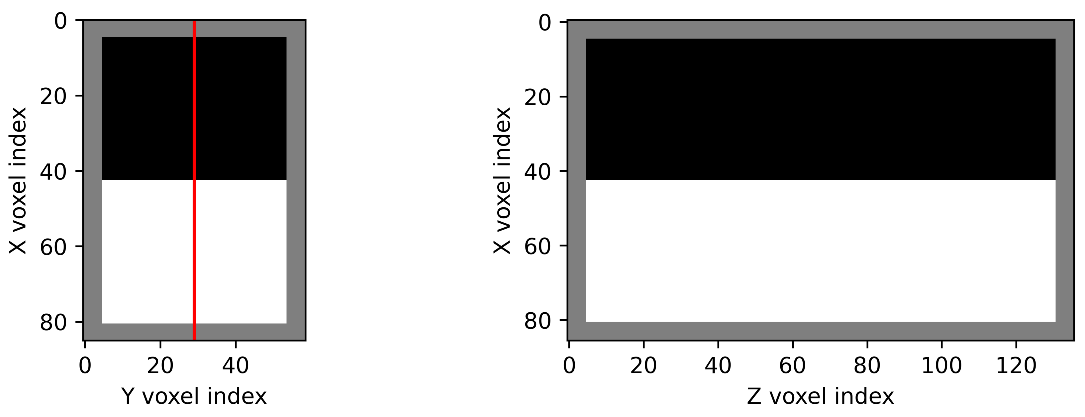


Figure 5.6: Black and white edge volume. The size of this volume is 86 by 59 by 136 voxels.

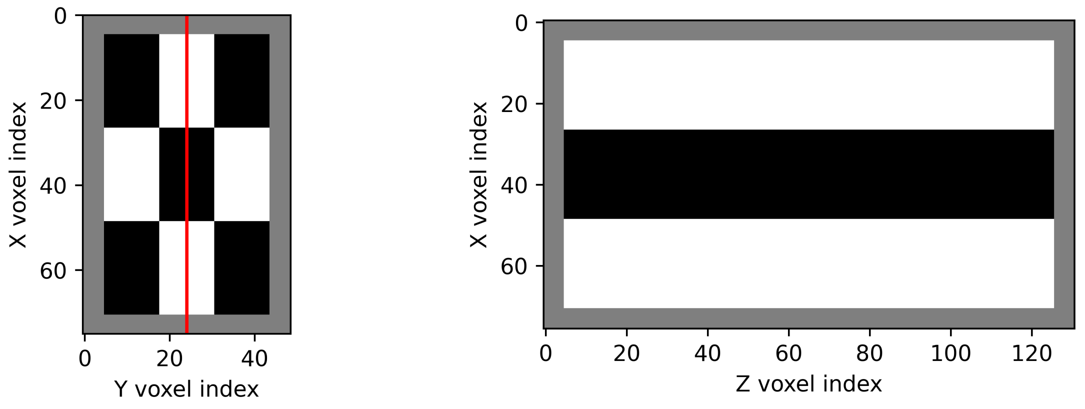


Figure 5.7: Black and white checkerboard volume. The size of this volume is 76 by 49 by 131 voxels.

exactly in the middle of the volume. The representation network was queried multiple times per voxel to visualize its interpolation capabilities. Again, the network trained with super-sampling follows the target values more closely while the other network smoothly interpolates between the values.

The super-sampled results clearly show the benefits of such an approach. For this reason, we trained all other networks with super-sampling. However, it is important to note that super-sampling introduces eight additional samples per voxel, which nearly doubles the size of the training dataset. This leads to longer training times. The super-sampled network took 1964 s to train with NVIDIA RTX 2060 GPU and AMD Ryzen 5 3600 CPU. While the network without super-sampling took 1006 s with the same hardware. Both networks were trained for 15 epochs as the rest of the other simple volumes.

Scattering and absorption coefficients representation We can examine the representation of other simple volumes by studying the aggregate values of scattering and absorption coefficients. Figures 5.12–5.14 show the values of said coefficients by aggregating them for each volume and level separately. The black dots represent values from the target volumes, while the red area around them is the prediction of our representation network. The larger this area is the less

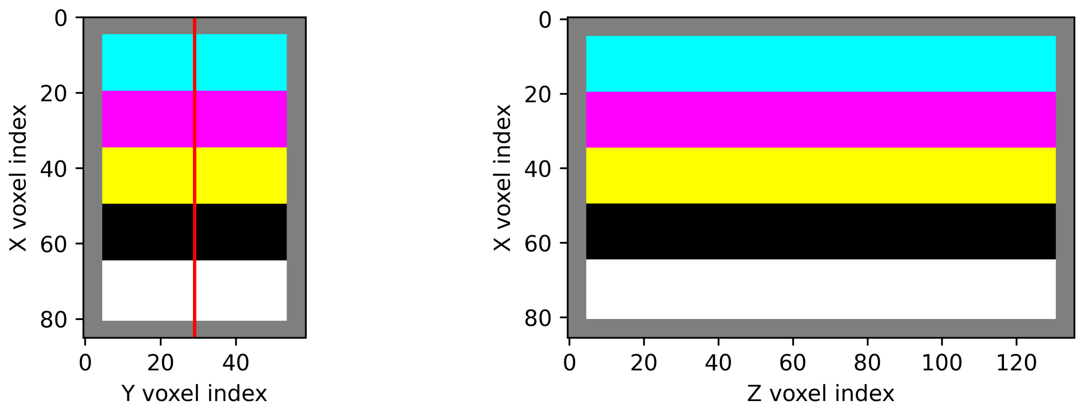


Figure 5.8: Colorful stripes consisting of all possible materials The size of this volume is 86 by 59 by 136 voxels.

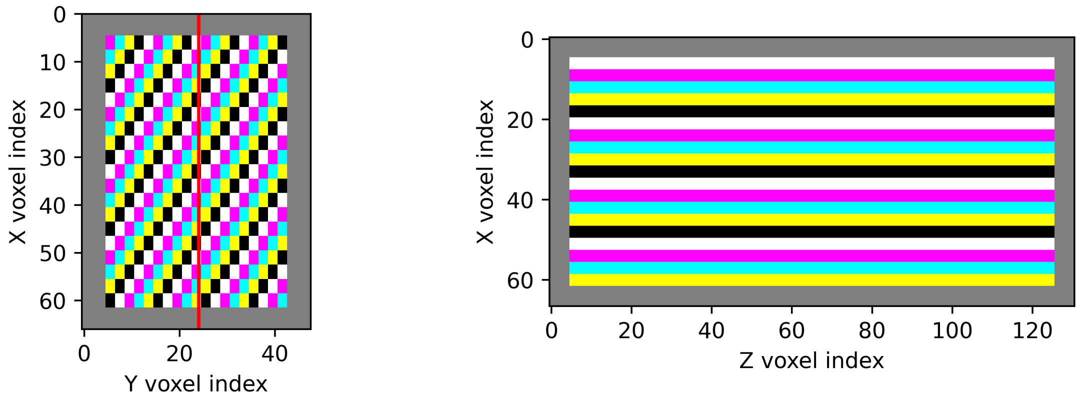


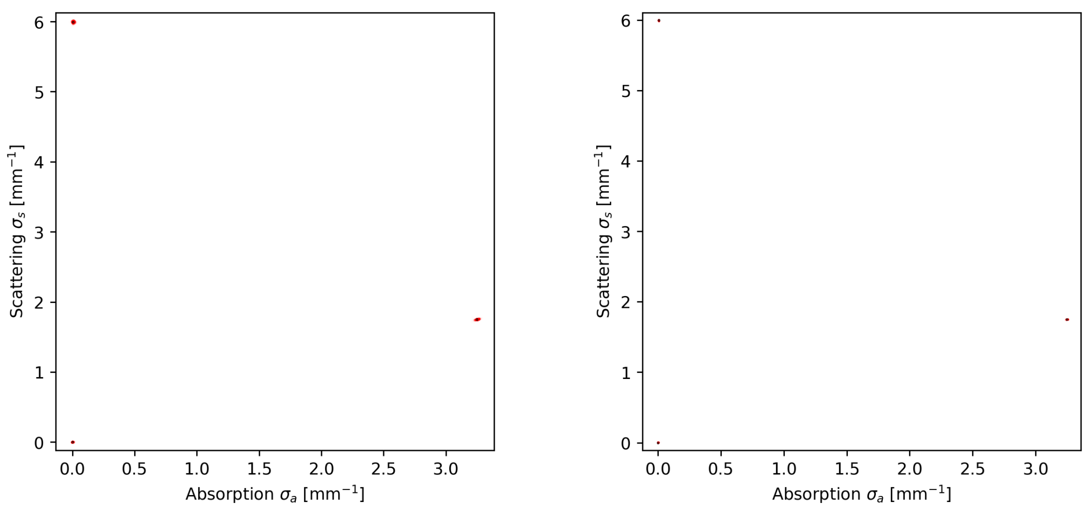
Figure 5.9: Colorful checkerboard volume. The size of this volume is 67 by 48 by 131 voxels.

precise the representation network is.

Figure 5.12 shows aggregate scattering and absorption coefficients of different blur levels for black and white checker volume. Black dots denote the target values and the red area around them shows the output of the representation network. The sharpest level is shown in Figure 5.12a. The three distinct dots that can be seen correspond to three materials: black, white, and air. Higher levels of blur mix these three materials together, so there are more black dots in Figures 5.12b–5.12d. The size of the red area is small so the quality of the representation is still very good.

Because the whole volume is covered in a thin layer of air material on the boundary, the scattering and absorption coefficients of higher levels tend to zero as the blurring kernel size increases and we use the value on the boundary whenever the blurring kernel needs value outside of the volume. Due to this phenomenon, we omit the figures for higher levels of blur as the values are very close to zero but they are available in the appropriate Jupyter notebook.

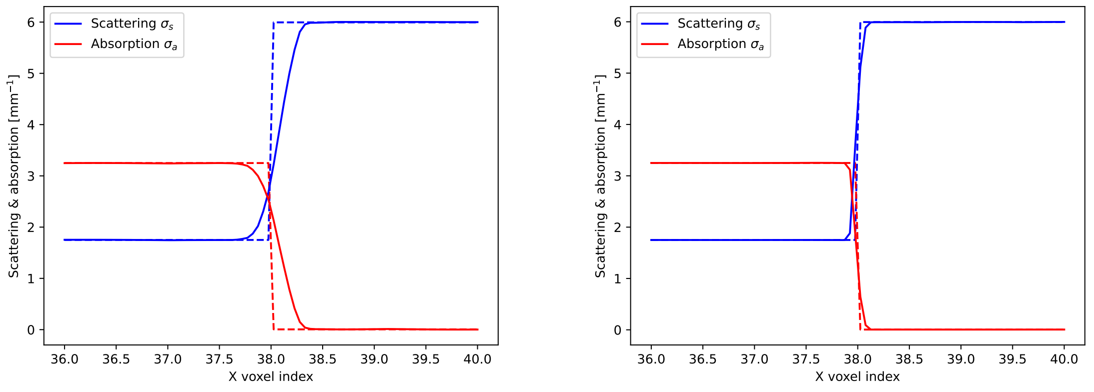
Training this network took 2046 s which is a little longer than in the case of super-sampled black and white edge. Given the sizes of the volumes are the same, the time should also be the same. We can use this difference to establish the minimal uncertainty of time measurement.



(a) No super-sampling.

(b) Super-sampling.

Figure 5.10: Scattering and absorption coefficients of the first level of black and white edge volume. The black points correspond to the target values across the whole volume. The red points correspond to the prediction of our representation. Notice the larger spread around the black points in the case without super-sampling.



(a) No super-sampling.

(b) Super-sampling.

Figure 5.11: Scattering and absorption coefficients of the first level of black and white edge volume. The dashed lines correspond to the target values. The solid lines correspond to our representation network. The representation network was queried at multiple locations per voxel to give a smooth line. Notice that, in the case with super-sampling enabled, the line is much steeper.

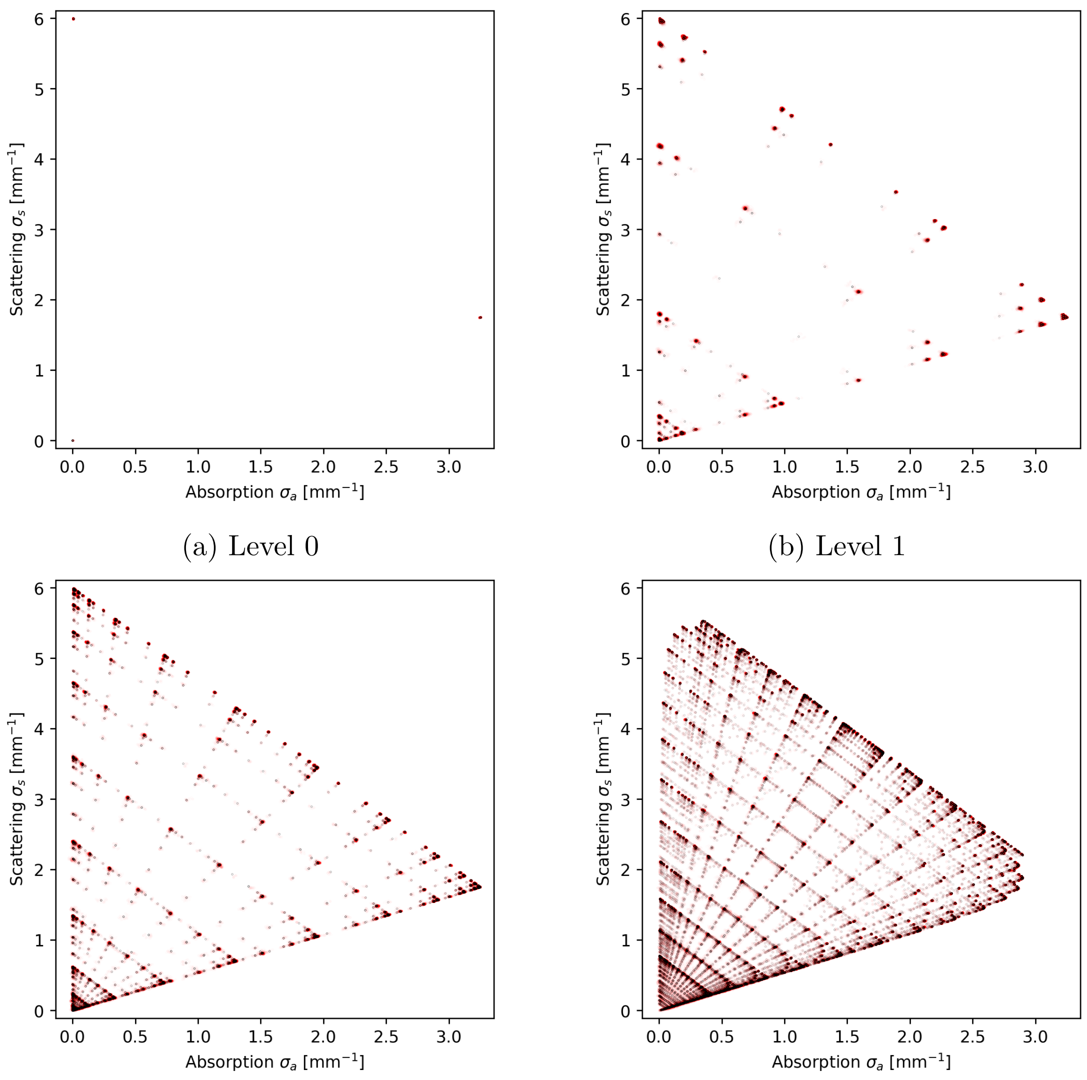
Figure 5.13 illustrates the aggregate values for the colorful stripes volume. This volume has five different materials and air, this manifests as six different black dots in Figure 5.13a. The values are again interpolated on higher levels, resulting in more dots.

The representation of this volume is also good as red areas are still small. This is as expected as the volume is similar in complexity to the previous one. The time to obtain these results was 1981 s on the same hardware as the volumes before. The shorter time than in the previous case can be explained by the volume being about 2 % smaller.

Finally, Figure 5.14 depicts the results of the colorful checkerboard volume. This is the most complex volume of all the simple volumes. The representation isn't as precise as it was in previous cases, which is evident from larger red areas around black dots. For example, on the first level, the difference between this volume and the previous one is about the same as the difference between enabled or disabled super-sampling (compare Figures 5.11a and 5.14a).

The training time for this volume was 1721 s. The shorter time can be explained by the volume being smaller than previous ones by about 12 %. The difference in times roughly matches the difference in size.

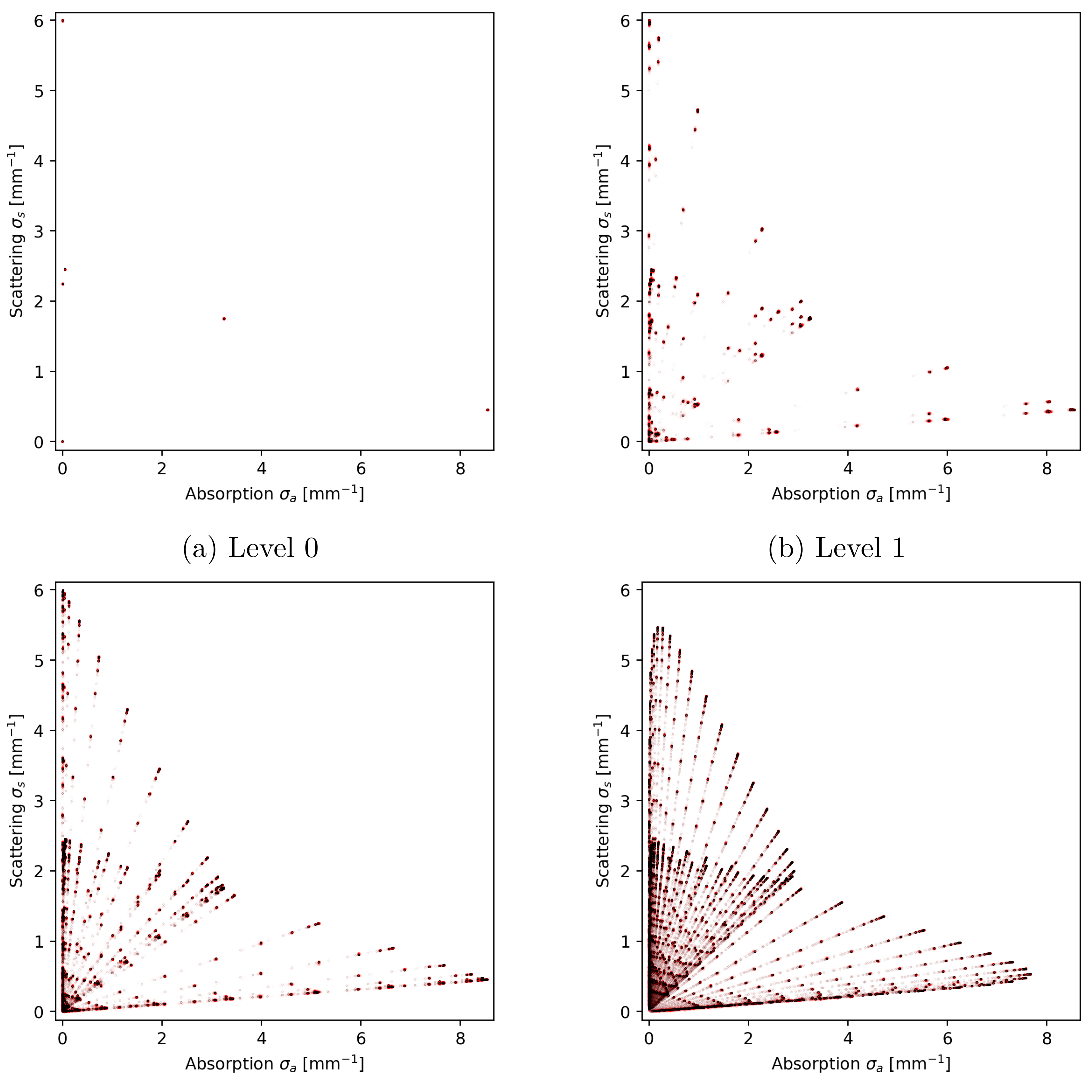
To create meaningful conclusions in the later chapters, we also trained the



(a) Level 0 (b) Level 1

(c) Level 2 (d) Level 3

Figure 5.12: Scattering and absorption coefficients of the red channel of black and white checker volume, various blur levels. The black points correspond to the target values across the whole volume. The red points correspond to the prediction of the representation network.



(a) Level 0 (b) Level 1

(c) Level 2 (d) Level 3

Figure 5.13: Scattering and absorption coefficients of the red channel of colorful stripes volume, various blur levels. The black points correspond to the target values across the whole volume. The red points correspond to the prediction of the representation network.

representation network on the colorful checkerboard volume for 35 epochs. We chose the number of epochs so that the loss and MAPE metric would match the values reached by previous volumes. This longer training took 4038 s which is proportional to the increase in the number of epochs.

All results presented in this section can be viewed in Attachment A.1 inside a Jupyter notebook called `3D volumes.ipynb`. There are scattering and absorption figures for levels of blur omitted from this section as well.

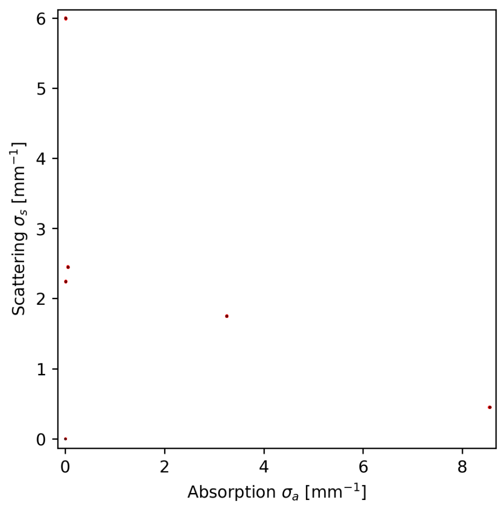
5.3.2 Complex volume

As stated above, complex volumes were taken from the training dataset of the appearance prediction network presented by Rittig et al. [2021]. We will present one such volume that will help us illustrate challenges presented by these more complex volumes[1]. Figure 5.15 illustrates the overall composition of the volume that we are representing. The first two figures from the left show cuts through the middle of the volume in XY and XZ planes. The figure on the right shows the outermost voxels of the volume. We obtained these voxels by ray-marching through the volume in the XZ plane in the direction of the Y axis and when a non-empty voxel was encountered it was displayed in this image.

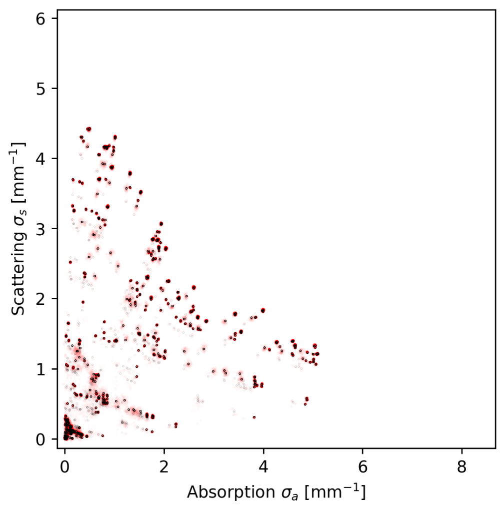
We can see that the volume consists mostly of white material that is broken by occasional magenta, yellow, black, or cyan material deposited near the surface. These materials are mixed with dithering to create different colors, as in real-world applications, which poses a real challenge to the representation network.

Table 5.8 shows the configuration of the representation network. The size of

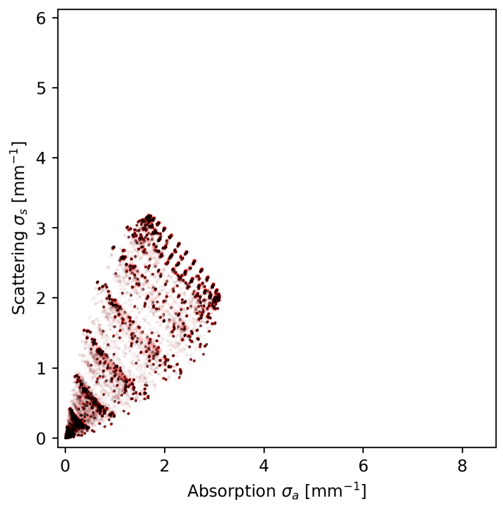
[1]We used the volume: "Sphere 5 mm 9" from the training dataset of Rittig et al. [2021]



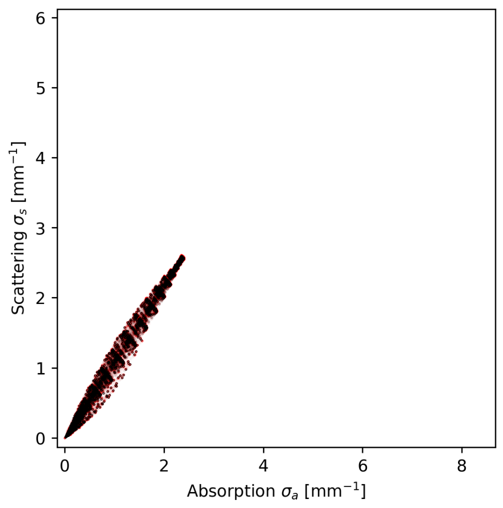
(a) Level 0



(b) Level 1



(c) Level 2



(d) Level 3

Figure 5.14: Scattering and absorption coefficients of the red channel of colorful checkerboard volume, various blur levels. The black points correspond to the target values across the whole volume. The red points correspond to the prediction of the representation network.

Table 5.8: Parameters of representation network used to represent complex volumes.

Hidden layers	8+1
Width of hidden layers	512
Width of final hidden layer	256
Positional encoding scale	1.15
Positional encoding size	64
Epochs	100
Initial learning rate	$5 \cdot 10^{-4}$
Batch size	2048
Trainable parameters	2 367 234

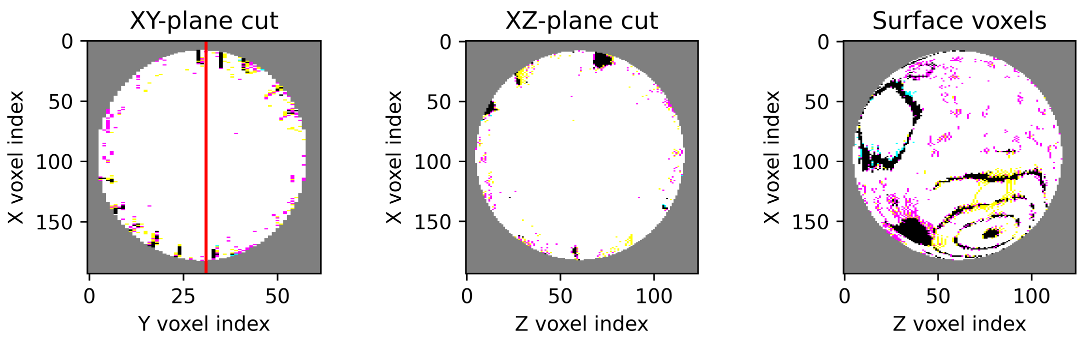
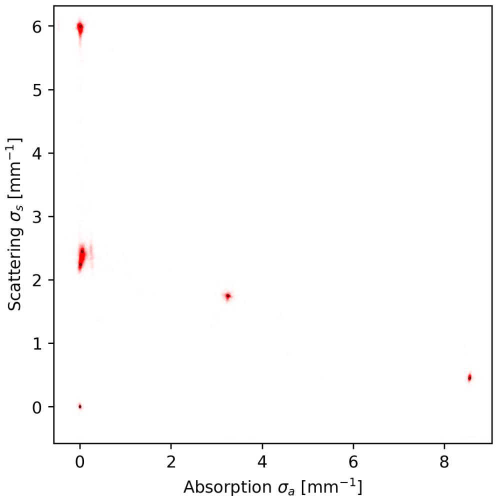


Figure 5.15: Complex volume. The images on the left and in the middle show the cut through the volume. The image on the right shows the outermost voxels of the volume. The size of this volume is 194 by 62 by 124 voxels. Notice the non-uniform size of voxels.

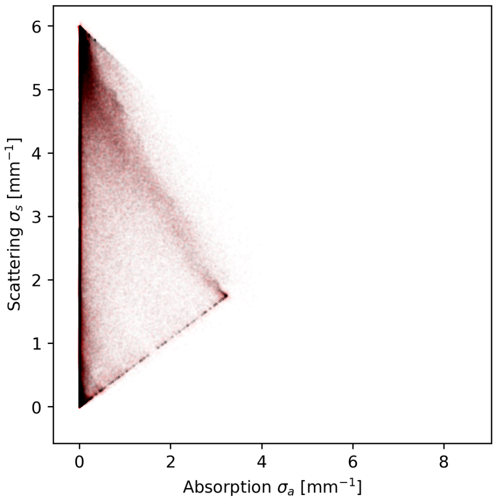
blurring kernels was the same as in the case of simple volumes (see Table 5.7) as these sizes are dictated by the appearance prediction network. For the training of this volume, we used the super-sampling for the most detailed level. The representation network was trained on the server with Intel Xeon E5-2680 v3 CPU, and NVIDIA Titan RTX GPU. The training time for all three channels was 1 575 minutes which averages to roughly 525 minutes per channel and 315 s per epoch. Even though the training took place on more powerful hardware than the training of the representation of simple volumes, the time per epoch is longer. We attribute this to the volume being larger and to the representation network containing more trainable parameters.

As with simple volumes, we will take a look only at a single channel, because results for all three color channels are qualitatively the same. Figure 5.16 shows the target and represented values of the scattering and absorption coefficients in the same way as in the case of simple volumes (Figures 5.12–5.14). We can see that the quality of the representation is much worse. On the first level, the spread around the target values is much bigger than in any of the simple volumes. The spread is also quite large on higher blur levels. However, it seems that there are no outliers and while the predicted values are reasonably close to the target values, the representation is certainly suboptimal.

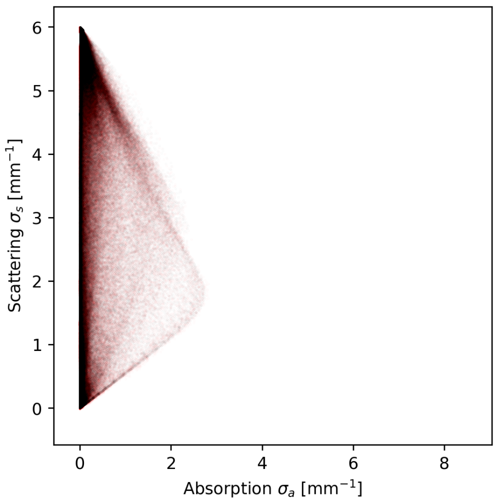
Values in Table 5.9 were computed based on stencils that were later used to generate the appearance predictions and as a base we used the training data. Before calculating the RMSE, we normalized the values to the largest value in the training data. This table includes all color channels. We can see that the value for the complex volume is substantially larger than for simple volumes, despite the fact that the representation network was trained for more epochs. The best result is recorded for the colorful checkerboard which is the most complex of the simple volume, we attribute this to more training epochs being used. The rest of the simple volumes have similar quality of representation.



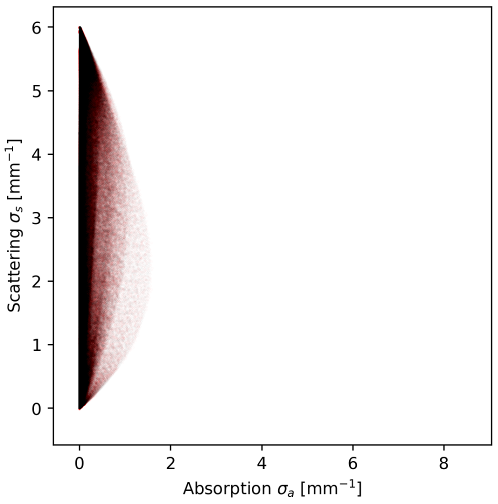
(a) Level 0



(b) Level 1



(c) Level 2



(d) Level 3

Figure 5.16: Scattering and absorption coefficients of the red channel of the complex volume, various blur levels. The black points correspond to the target values across the whole volume. The red points correspond to the prediction of the representation network. Notice the large spread of predicted values, indicative of suboptimal representation.

5.4 Appearance prediction network

Let's investigate how well the representation network works with the appearance prediction network. For this case, we ran two experiments, one with the aforementioned simple volumes and one with the complex volume. Simple volumes allowed us to examine individual artifacts more closely, while the complex volume gave us overall insight into the usefulness of our approach.

Figure 5.17 explains different experiment configurations that we have used. Config A corresponds to Rittig et al. [2021], Config B is modified to work with Gaussian blurring instead of box averaging, Config C represents our solution, and Config D utilizes Monte Carlo rendering and can be considered a ground truth. Simple volumes were used in Config A, Config B, and Config C. While complex volume was used in Config A, Config C, and Config D.

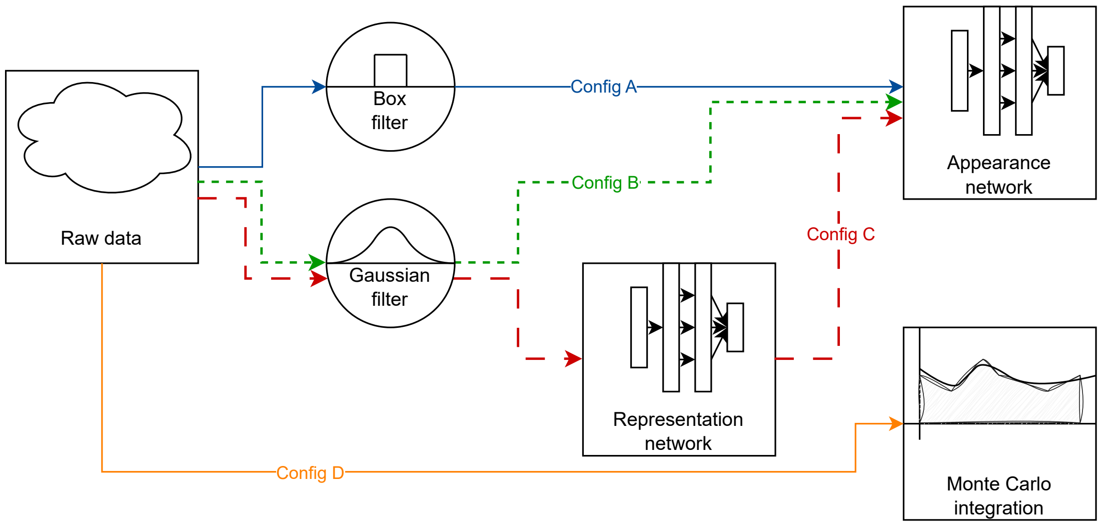


Figure 5.17: Diagram explaining different rendering configurations. Config A uses the grid with box kernel and feeds directly into the appearance prediction network, this corresponds to Rittig et al. [2021]. Config B uses the grid with Gaussian blurring that also feeds into the appearance prediction network. Config C uses the representation network with Gaussian blurring. Config D takes raw volumetric data and renders it with a Monte Carlo renderer, this can be considered as ground truth.

Table 5.9: RMSE between representation network output and the Gaussian blurred data for all values in all stencils used to create the appearance prediction. The values were normalized before computing the RMSE.

Volume	Epochs	RMSE
Black and white edge	15	0.141
Black and white checkerboard	15	0.148
Colorful stripes	15	0.131
Colorful checkerboard	35	0.114
Complex volume	100	0.205

5.4.1 Simple volumes

For each of the simple volumes, we ran the appearance network with three different input sources. At first, we used the representation network as the source of scattering and absorption coefficients. To evaluate the impact of the representation network, we used two variants of the voxel grid directly, without our representation network. By using two different kernels to blur the data, we obtained two versions of the grid representation. Firstly, we used the Gaussian kernel, which is the same kernel we used for our representation network. Secondly, we used the box kernel, which was originally used by the appearance prediction network.

The positions at which we evaluated the appearance prediction network coincide with the voxel grid, which means that there is no transformation applied. Chosen positions were located on the top layer of the volume. We didn't render sides without the patterns that extend to the Z axis.

The output of the appearance prediction network is saved into a vdb file, see Museth [2013] for details about the format. Final images are produced by rendering vdb files in VoxelViewer application by Kužel [2021].

Figures 5.18–5.21 show the results of the appearance prediction network in the described setting. The leftmost figure represents the prediction with the box kernel applied to the source data, the middle figure depicts the prediction with the Gaussian kernel applied to the source, and the rightmost figure shows the prediction using the representation network. We can see that overall, there isn't a large difference between the predictions using the box and Gaussian kernels. However, there is quite a large disparity between these images and the predictions using the representation network.

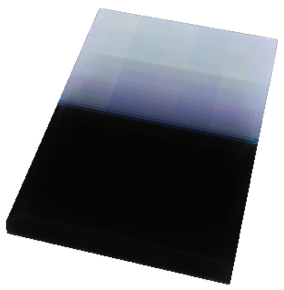
Looking at the black and white edge volume in Figure 5.18, we can see that the image produced with the representation network has a blue tint and the white part of the volume has a significant amount of color noise in it. Additionally, there are noticeable blocky artifacts visible. On the other hand, the transition of colors on the edge is decent and sharp enough.

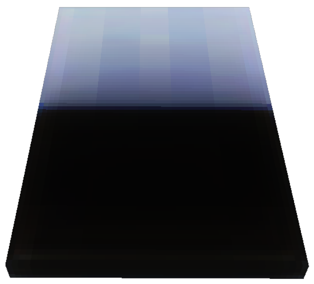
The issues with transitioning over an edge are visible on the checkerboard pattern in Figure 5.19. There is a considerable amount of green voxels around the edges. However, in this case, the overall tint of the volume isn't as off as in the previous case.

Subjectively, the best result is probably the prediction of the colorful stripes volume as evident in Figure 5.20. The transitions between individual colors are fine, although there are still some noisy patches present. Another artifact that can be observed on this particular volume is the overall brightness difference.

The greatest brightness difference can be observed in the case of the colorful checkerboard volume in Figure 5.21. Even though the overall pattern can be seen, the individual rectangles are deformed by noise.

Stencils for the appearance prediction network were generated in Jupyter notebooks contained in Attachment A.1. Notebook `3D Volumes Mocked.ipynb` contains the code to generate stencils from the training data directly and notebook `3D volumes appearance.ipynb` contains the code to generate stencils using the representation network. Due to space limitations, we have elected not to include the generated stencils with our work as they can be easily reproduced using the appropriate notebooks.


(a) Config A
Box kernel


(b) Config B
Gaussian kernel

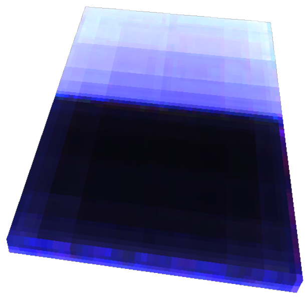
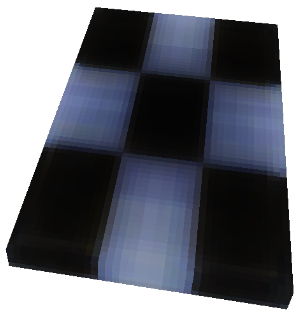
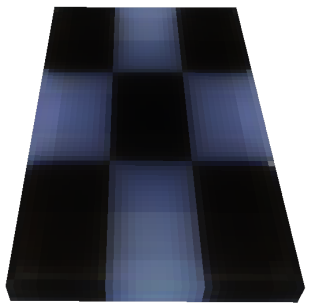

(c) Config C
Representation network

Figure 5.18: Appearance prediction of the black and white edge volume.


(a) Config A
Box kernel


(b) Config B
Gaussian kernel

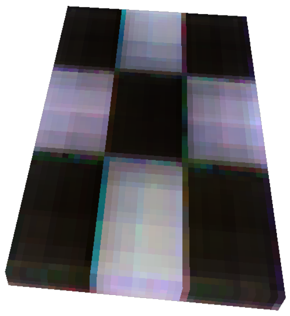
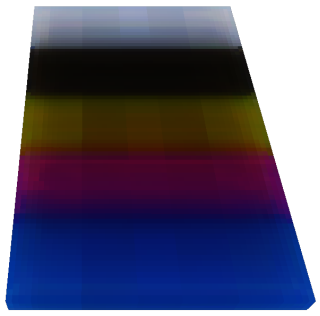
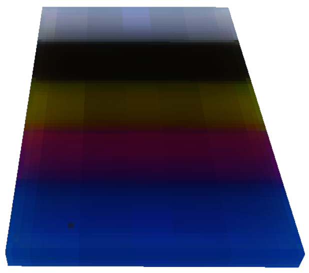

(c) Config C
Representation network

Figure 5.19: Appearance prediction of the checker volume.


(a) Config A
Box kernel


(b) Config B
Gaussian kernel

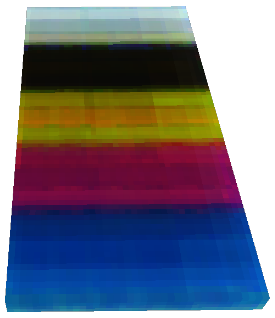
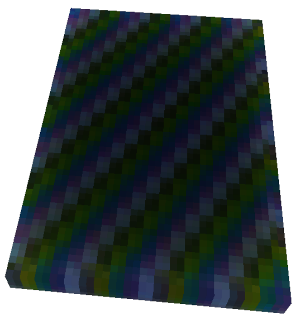
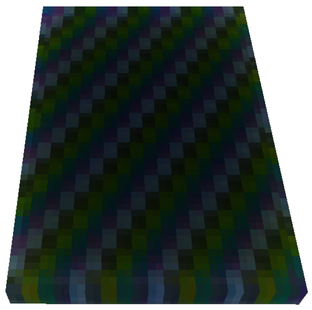

(c) Config C
Representation network

Figure 5.20: Appearance prediction of the colorful stripes volume.


(a) Config A
Box kernel


(b) Config B
Gaussian kernel

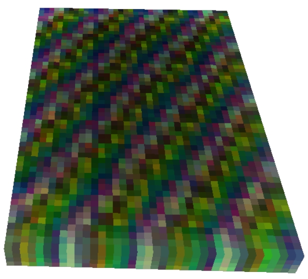
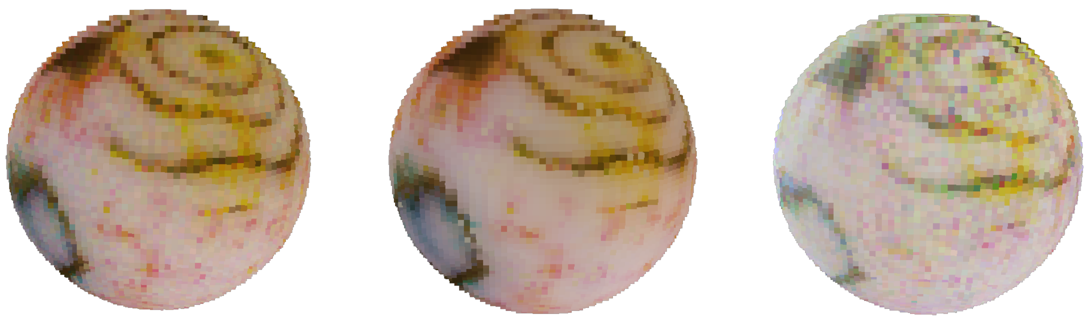

(c) Config C
Representation network

Figure 5.21: Appearance prediction of the colorful checkerboard volume.



(a) Config D
Monte Carlo renderer

(b) Config A
Box kernel

(c) Config C
Representation network

Figure 5.22: Appearance prediction of the complex volume.

Table 5.10: RMSE of appearance prediction using representation network.

Volume	Configurations used for comparison	RMSE
Black and white edge	Config C to Config A	0.190
Black and white checkerboard	Config C to Config A	0.145
Colorful stripes	Config C to Config A	0.154
Colorful checkerboard	Config C to Config A	0.171
Complex volume	Config C to Config D	0.356

5.4.2 Complex volume

Given that the complex volume is from the training set of the appearance prediction network we have also the physically accurate prediction available, which was obtained using Monte Carlo integration of the rendering equation (Equation 1.17). We can compare our results with this integration result as well as with the output of the appearance prediction network.

In contrast to the previous case of simple volumes, there is a transformation applied between the original voxel grid and the output voxel grid, resulting in queries into the voxel grid that aren't centered on the original voxel grid. We speculate that this might bring in more inaccuracies in our representation network even though we used super-sampling during the training.

Figure 5.22 depicts the complex volume in question. The previously mentioned ground truth prediction image is portrayed in Figure 5.22a and even though there is some noise, typical for Monte Carlo integration, all shapes and features are clearly visible. Figure 5.22b displays the result of the original appearance prediction network as presented by Rittig et al. [2021]. This result lacks the noise but some features aren't as clearly visible as in the ground truth case. Both of these results are taken from Rittig et al. [2021] data.

Finally, Figure 5.22c shows the result using the representation network. A lot of the artifacts present in the case of simple volumes can be collectively observed in this image. For example, the image is brighter than the target. Also, there is a lot of color noise and some features can't be distinguished because of it. It seems, that the sphere lacks the proper color and appears much whiter than the target images. To give credit to our representation network, there are hints of the surface structures apparent.

Table 5.10 describes the RMSE of different appearance predictions. Interestingly, among simple volumes, there seems to be no correlation to the values presented in Table 5.9 as the colorful checkerboard has the highest RMSE of appearance prediction while having the lowest RMSE of representation. To calculate the RMSE for the complex volume, we excluded all empty voxels to obtain a more accurate result. We can see that there is a big difference between simple and complex volumes, the difference in RMSE is almost twofold.

6. Discussion

2D images The results of representing single two-dimensional images show that the representation networks greatly benefit from positional encoding as was shown by Mildenhall et al. [2020]. Even a smaller network that offers data size compression performs reasonably well. However, the compression ratio can't compete with purpose-built algorithms such as jpeg.

The larger representation network achieves better results, especially in the area of color saturation, which is depicted more accurately. Nevertheless, when comparing the number of trainable parameters with the number of floating point values needed to represent the original image, this larger network requires more space and is thus inefficient for encoding.

Interestingly, when using a representation network of the same size for representing a series of blurred images, it handles the task well even though it can represent more levels of detail. The image with the finest details is represented in a comparatively same quality as it was in the case of representing a single image. The difference in PSNR or maximum flip metric value is minimal and can be explained as an error of measurement.

As expected, the coarser levels of detail are represented even better as evidenced by larger PSNR values as well as by flip metric. We should point out the fact that even though there is less detail, the inputs to the network are also limited because of the attenuation term.

Volumes The results of representing volumes are mixed. On the one hand, the representation of simple synthetic volumes was done reasonably well. We were able to achieve good results even with small networks in a very short time with a minimal number of epochs. Also, we observed the precision of the representation decreases as the complexity of a target volume increases.

On the other hand, the representation of more complex volumes with real-world features wasn't as promising. Even with a much larger capacity and much longer training times, we couldn't represent the fine details as well, as in the case of simple volumes. The representation of coarser levels was better but still not as good as with simple volumes.

We could also see the positive result of super-sampling. Providing extra sampling points per voxel yielded crisper transitions on the first level as evidenced by Figure 5.11. This is important for accurately representing the dithering nature of the mixture space on the first level. Super-sampling isn't as important on higher levels as the transitions aren't expected to be as sharp, because of the applied blurring. Using super-sampling only on the first level looks like a good compromise, as more samples prolong the training times significantly, hindering the training performance.

In our method, we opted to use individual representation networks for each of the three color channels. The upside of this approach is that it can be easily extended into spectral rendering applications, where more channels are needed. Another advantage might be the ability to tweak the representation network parameters for each channel individually. However, we decided to use the same values for each color. The disadvantage of this approach might be the fact that the representation network might not be able to extract extra information from the training data as it is unaware of the correlation between the values of different channels.

Appearance prediction We were able to feed the appearance prediction network with our representation network as evidenced by Figures 5.18–5.22. Unfortunately, the results are disappointing and with several issues. Even the simple volumes, where the representation was very accurate, displayed defects. It seems, that the appearance prediction network is very sensitive to the precision of its inputs as only a small deviation from the original values leads to artifacts in the output as indicated by simple volumes. The complex volume example shows many of the artifacts that can be seen with individual simple volumes. These include brightness increase, tint shift, lower saturation, and color noise. Despite all these defects, the surface structure of the complex volume is somewhat apparent.

Limitations Our solution for a representing data structure has a significant amount of limitations that prevent it from being a viable solution. One such issue is the long training time. Each volume needs to be learned from scratch, which takes so much time that it defeats the purpose of the appearance prediction network as a faster way of rendering participating media.

Another limitation is the fact that the representation network parameters have to be tweaked for each volume to obtain the best time-to-quality ratio. This severely limits the generalization of our method to any volume.

In addition, the memory requirements are astronomical. We were able to represent only very small volumes on a commonly available desktop computer. Given the three-dimensional nature of the problem at hand, the scaling of the memory requirements is cubic with respect to the length of the edge of a volume.

The biggest limitation is the sub-optimal results. Sadly, the amount of noise and other artifacts in the output make our representation network unusable in its intended application.

Future work Our work leaves significant space for improvements, which unfortunately lies outside of the scope of this thesis. Firstly, to address the slow training times, a meta-learning approach might be used. This would train the representation network on a vast set of volumes, creating an initial state that should be hopefully optimized faster than a randomly initialized network. Because of the correlation between values of individual color channels, one might also use the weights of the first channel as a starting point of training for the subsequent channels.

Better results of the appearance prediction might be achieved by fine-tuning the appearance prediction network on the data that was blurred with the Gaussian kernel instead of the box kernel. Even though the results in subsection 5.4.1 indicate that the difference between box and Gaussian kernels isn't as significant as one might have thought. Another way to improve the appearance prediction network might be fine-tuning the network directly on the volumes represented by the representation network. This might help the appearance prediction network to be more tolerant of its inputs and produce less noisy images.

Conclusion

In this thesis, the main aim was to study the representation networks. We have evaluated their performance in various settings and investigated the possibility of replacing the SAT data structure in the appearance prediction network by Rittig et al. [2021].

We have successfully shown the benefits of positional encoding in representation networks for two-dimensional images. We have shown the implementation of attenuated positional encoding to sets of blurred two-dimensional images and that by masking the positional encoding in this manner, the representation network can effectively represent multiple levels of details of the same image.

We attempted to represent the volumetric data with our implementation of the representation network. We have shown that representing simple data is possible with very few errors. However, more complex volumes, which corresponded to the real-world application more closely, were missing some of the finer details and were lacking the required accuracy. We have also shown the benefits of supersampling the training data to get better fine details in the results.

The intended application of our representation network as a data structure in the appearance prediction network didn't yield the expected results. Even for the simple volumes, which were reasonably well represented, the outputs of the appearance prediction model exhibited a significant amount of color noise. The appearance prediction network produced significantly lighter output when it was using the representation network as its input.

Overall we have found out that representation networks as presented in our thesis aren't suitable as a replacement for summed-area tables in appearance prediction networks proposed by Rittig et al. [2021]. Additional work has to be done to improve the representation of finer details. Also, the appearance prediction network should be fine-tuned to reflect the change in the smoothing kernel in the data structure.

We consider the results of our novel approach to be partial success. It showed some promise, however, there are more downsides than upsides.

Bibliography

Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. FLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020. doi: 10.1145/3406183.

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

Crow, Franklin C. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84. ACM, January 1984. doi: 10.1145/800031.808600. URL `http://dx.doi.org/10.1145/800031.808600`.

Oskar Elek, Denis Sumin, Ran Zhang, Tim Weyrich, Karol Myszkowski, Bernd Bickel, Alexander Wilkie, and Jaroslav Křivánek. Scattering-aware texture reproduction for 3D printing. *ACM Transactions on Graphics*, 36(6):1–15, November 2017. doi: 10.1145/3130800.3130890. URL `https://doi.org/10.1145/3130800.3130890`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, La Jolla, CA, USA, September 2008.

James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. ACM, August 1986. doi: 10.1145/15922.15902. URL `https://doi.org/10.1145/15922.15902`.

James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques - SIGGRAPH '84*. ACM Press, 1984. doi: 10.1145/800031.808594. URL `https://doi.org/10.1145/800031.808594`.

Simon Kallweit, Thomas Müller, Brian Mcwilliams, Markus Gross, and Jan Novák. Deep scattering. *ACM Transactions on Graphics*, 36(6):1–11, November 2017. doi: 10.1145/3130800.3130880. URL `https://doi.org/10.1145/3130800.3130880`.

Animesh Karnewar, Tobias Ritschel, Oliver Wang, and Niloy Mitra. ReLU Fields: The Little Non-Linearity That Could. In *ACM SIGGRAPH 2022 Conference Proceedings*, SIGGRAPH '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393379. doi: 10.1145/3528233.3530707. URL `https://doi.org/10.1145/3528233.3530707`.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL `https://arxiv.org/abs/1412.6980`.

Vojtěch Kužel. Real-time voxel visualization and editing for 3D printing. Master's thesis, Charles University, Faculty of Mathematics and Physics, Department of Software and Computer Science Education, Prague, September 2021. URL `http://hdl.handle.net/20.500.11956/148776`.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017.

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020.

Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *ACM Transactions on Graphics*, 40 (4):1–16, July 2021. doi: 10.1145/3450626.3459812. URL `https://doi.org/10.1145/3450626.3459812`.

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022. doi: 10.1145/3528223.3530127. URL `https://doi.org/10.1145/3528223.3530127`.

Ken Museth. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics*, 32(3):1–22, June 2013. ISSN 1557-7368. doi: 10.1145/2487228.2487235. URL `http://dx.doi.org/10.1145/2487228.2487235`.

Thomas Klaus Nindel, Tomáš Iser, Tobias Rittig, Alexander Wilkie, and Jaroslav Křivánek. A gradient-based framework for 3D print appearance optimization. *ACM Transactions on Graphics*, 40(4):1–15, July 2021. ISSN 1557-7368. doi: 10.1145/3450626.3459844. URL `http://dx.doi.org/10.1145/3450626.3459844`.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering.* Elsevier, 2016. ISBN 978-0-12-800645-0. doi: 10.1016/C2013-0-15557-2. URL `https://doi.org/10.1016/C2013-0-15557-2`.

Tobias Rittig, Denis Sumin, Vahid Babaei, Piotr Didyk, Alexey Voloboy, Alexander Wilkie, Bernd Bickel, Karol Myszkowski, Tim Weyrich, and Jaroslav Křivánek. Neural acceleration of scattering-aware color 3d printing. *Computer Graphics Forum*, 40(2):205–219, May 2021. doi: 10.1111/cgf.142626. URL `https://doi.org/10.1111/cgf.142626`.

Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 7537–7547. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper_files/paper/2020/file/55053683268957697aa39fba6f231c68-Paper.pdf`.

Matthew Tancik, Ben Mildenhall, Terrance Wang, Divi Schmidt, Pratul P. Srinivasan, Jonathan T. Barron, and Ren Ng. Learned initializations for optimizing coordinate-based neural representations, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

D.Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, December 2003. doi: 10.1016/s0893-6080(03)00138-2. URL `https://doi.org/10.1016/s0893-6080(03)00138-2`.

List of Figures

List of Tables

List of Abbreviations

DFT	–	Discrete Fourier transform
SD	–	Standard deviation
MSE	–	Mean squared error
RMSE	–	Root-mean-square error
PSNR	–	Peak signal-to-noise ratio
MAPE	–	Mean absolute percentage error
MLP	–	Multilayer perceptron
SGD	–	Stochastic gradient descent
Adam	–	Optimization algorithm with adaptive moments
BRDF	–	Bidirectional reflectance distribution function
RPNN	–	Radiance prediction neural network
SAT	–	Summed-area table
NeRF	–	Neural radiance field
GPU	–	Graphics processing unit
CPU	–	Central processing unit
API	–	Application programming interface

A. Attachments

A.1 Contents of Electronic Attachment

The electronic attachment contains these items:

- AppearancePredictionNetwork – The original source code of Rittig et al. [2021].
  - fabnn/predict_stencils.py – Our script that loads stencils outputted by the representation network and makes an appearance prediction.
- RepresentationNetwork – The source code for representation network.
  - Dockerfile – The dockerfile is used to build the Docker image.
  - docker-compose.yaml – A configuration for Docker Compose, that is used to create an image and start up the container.
  - dev-env – This folder contains all scripts with our implementation.
    - data – This folder contains all data to be used for training.
    - modules – A folder with most of our scripts.
    - notebooks – Notebooks with experiments.
    - scripts – Scripts contained here are used when running the container via Docker Compose.
    - pretrained – This folder contains configurations for the representation network.
    - results – This is where the results of notebooks are saved.
  - third-party – Third-party libraries necessary for our package: OpenVDB and Blosc implementation.

A.2 Documentation

As stated in chapter 4, we have used Docker to package our solution together. The container was tested to run on Windows with WSL 2 as well as in the Linux environment. We also provide the implementation of the appearance prediction network as presented by Rittig et al. [2021] in a separate container complemented by our script that can render the stencils outputted by the representation network.

Requirements

In order to run our solution, the system that is intended to be used with our solution must include support for Linux-based Docker containers. The system must also have an NVIDIA GPU with driver support for CUDA 11.8. The CUDA toolkit is not required. A sufficient amount of RAM is also required. We tested our solution with 32 GB of system RAM and at least 6 GB of video memory.

Interactive mode

To view the Jupyter notebooks, the Docker container should be started without the Docker Compose command. We call this mode of operation an interactive mode.

Setup

First, an image has to be built, to do so run the following command in the `RepresentationNetwork` directory, where our Docker file is located:

```
docker build . -t <<your tag name>>
```

After the preparation of the image, the container can be created by the following command:

```
docker run -it --gpus all \
-p <<your jupyter port>>:8888 \
-v dev-env:/dev-env/  \
--name <<your container name>> \
<<your tag name from previous command>> bash
```

If everything finishes successfully, a bash environment in the newly created container should be opened.

When the container is already created, but the container was stopped or the bash environment was exited, following steps can be used to enter the container's command line. At first, the container has to be started by this command:

```
docker start <<your container name>>
```

Next, the interactive bash command should be executed:

```
docker exec -it <<your container name>> bash
```

Running a Jupyter server

When the setup is properly completed, the bash command line environment of the container should be available. To start up the Jupyter server the following command should be used:

```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so.2 &&
jupyter notebook  --ip 0.0.0.0 --port 8888 --allow-root
```

To open the provided notebooks navigate in your favorite browser to:

```
http://localhost:<<your jupyter port>>/notebooks
```

Copy the token from the container's command line and paste it into the token text field. After that, the following notebooks can be viewed:

- 2D Images.ipynb – An example of representation network presented in section 3.1.1.

- 2D Blurred Images.ipynb – An example of representation network presented in section 3.1.2.
- 3D Volumes.ipynb – An example of representation network presented in section 5.3.
- 3D Volumes Appearance.ipynb – This notebook generates stencils for the appearance network using the representation network, as presented in section 5.4.1.
- 3D Volumes Mocked.ipynb – This notebook generates stencils for the appearance network using the training data, as presented in section 5.4.1.
- Complex Volume Representation.ipynb – This notebook inspects the result of complex volume representation, which was presented in section 5.3.2.

Training mode

To run a particular workload non-interactively, a Docker Compose might be used to make the setup faster and easier. This workload is useful when the solution is being deployed on the compute server for training. We provide a simple docker compose configuration that can be edited to run the desired job.

To make use of the docker compose configuration file, copy the contents of the `RepresentationNetwork` directory of electronic attachment to the desired machine. Edit the `docker-compose.yaml` file by commenting out or modifying the command (the last entry in the configuration file). These are the available commands:

- `repNet_training.py` – This will train the representation network based on the file containing the description of the voxels, specified as a `file` argument (the file has to be located in the data folder and its name has to end with `_discrete_volume.gz`). Other available arguments are: epoch (controlling the number of epochs the training will take), model (a path to a JSON configuration of the representation network), logdir (an optional path to the folder where to store tensorboard logs), checkpoints (a boolean flag indicating whether to store checkpoints during the training), suffix (an optional string that gets appended to the trained network save name), restarts (how many times to repeat the training with learning rate reset to the starting value).
- `render_all.py` – This will render the predicted volume into the vdb file. The representation network is loaded from a standard location and is determined by the combination of file, suffix, and designation parameters. The suffix parameter is either `"final"` or the number of epochs, if a checkpoint should be used. The model configuration JSON has to be provided and it has to be the same configuration as the one used for the training.
- `create_stencil.py` – This command will produce the three files required by the appearance prediction network. These files contain scattering and absorption coefficients. Representation networks for all three channels must exist. The suffix parameter is either `"final"` or the number of epoch if a checkpoint should be used. The model configuration JSON has to be provided and it has to be the same configuration as the one used for the training.

- `main.py` – This will run all previous commands successively. It has only one parameter which is the file that will be used for training. There are the same requirements for this input file as in the case of the training command. The training runs for 100 epochs. The JSON configuration is taken from the repNet.json in the pretrained directory.

Table A.1 contains the explanation of the options that are used to construct the representation network. These options are defined in the JSON file that has to be passed to the commands. The configuration is used to construct the representation network model described in section 3.1.3.

Table A.1: Parameters of representation network JSON configuration.

JSON keyword	Explanation
`posenc_size`	The number of frequencies used for positional encoding
`posenc_scale`	This controls the spacing of the frequencies on the positional encoding
`layer_depth`	The amount of hidden layers in the first part of the network
`layer_width`	The size of hidden layers in the first part of the network
`skip_connection`	How many hidden layers to skip when using skip connections
`layer_depth_2`	The amount of hidden layers in the second part of the network
`layer_width_2`	The size of hidden layers in the first second of the network
`output_dimensions`	The number of outputs (has to be)
`batch_size`	The size of batches when training the netwok
`learning_rate`	The initial learning rate
`final_activation`	The name of the activation function used on the output layer
`log`	A path to a tensorborad log that will be created or `"No log"`

Once the `docker-compose.yaml` is edited to run the desired script, execute the following command to run the Docker container:

```
docker-compose up
```

Appearance prediction

The appearance prediction is also provided as a Docker container. It is the same code as Rittig et al. [2021] provided for their work. We added our script that can process stencils produced by our representation network.

To build a container navigate to `AppearancePredictionNetwork` directory and run following command:

```
dokcer build -t neural-scattering-prediction:latest .
```

After that, you will have to copy the stencil files to the `data/stencils` directory. Stencil files produced by the representation network consist of three different files. The first describes the shape of the output of the appearance prediction network, this file has a suffix `_out_shape.npy`. The second file describes the positions that should be populated in the appearance prediction output, this file has a suffix `_positions.npy`. The last file contains all scattering and absorption coefficients for all stencil positions and all color channels, this file has a suffix `_stencils.npy`. All three files have to be copied to the data directory of the appearance prediction network. Next, the container can be started by the following command:

```
docker run --rm \
--gpus all \
--mount type=bind,source="$(pwd)/data",target=/project/data \
neural-scattering-prediction:latest \
python fabnn/predict_stencils.py -s <<stencil_to_predict>>
```

The only parameter to change is the name of the stencil. For example, if there are files named: `cube_out_shape.npy`, `cube_positions.npy` and `cube_stencils.npy`, the parameter should be: `-s cube`. This command will produce a .vdb file in the `data/stencils` directory.