Interactive Reconstruction of Monte Carlo Sampled Images with Depth of Field

Master's thesis in Computer science and engineering

Simon Petersson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Interactive Reconstruction of Monte Carlo Sampled Images with Depth of Field

Simon Petersson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Interactive Reconstruction of Monte Carlo Sampled Images with Depth of Field
Simon Petersson

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Interactive Reconstruction of Monte Carlo Sampled Images with Depth of Field
Simon Petersson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The likelihood of deploying Monte Carlo path tracing as a real-time rendering technique for global illumination in production systems is ever-increasing. In recent years, developments in both software and hardware, have taken us much closer to a first version of such systems. Fast reconstruction techniques for approximating higher quality images from low sample count Monte Carlo renders, without adding additional samples, has been particularly influential.

We develop a convolution neural network for reconstructing Monte Carlo rendered images with low sample counts at interactive speeds. In particular, we focus on extending an already developed neural network to support depth of field effects. Our network is a deep autoencoder that utilizes a set of auxiliary buffers, containing additional information about each pixel. We propose a novel auxiliary buffer based on the circle of confusion size in each pixel. We show that by allowing the network to access this buffer during reconstruction, it learns to distinguish between points in and out of focus. Our network reconstructs images at highly interactive frame rates but does not meet the reconstruction quality of many other approaches. We discuss potential reasons behind these performance limitations and suggest a few next steps to improve reconstruction.

Keywords: Computer Graphics, Path Tracing, Real-Time Ray Tracing, Depth of Field, DOF, Machine Learning, Deep Learning, Autoencoder, Computer Science, Thesis

Acknowledgements

I would like to thank my supervisor Erik Sintorn for all his help. His feedback, guidance and motivation have been crucial to the success of this thesis. I would also like to extend my thanks to examinator Ulf Assarsson for his time, feedback, and availability. Finally, thank you to family and friends for expressing genuine interest and providing crucial motivation in times of need.

Simon Petersson, Gothenburg, Aug 2019

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

The strive towards greater realism is prevalent in many real-time rendering applications. In recent years, rasterization with physically-based rendering has been applied in many such applications to improve realism. While these techniques often produce good looking results, there are many scenarios in which they fall short. Two examples include global illumination and depth of field effects. I order to achieve better results, by producing overall higher quality renders than rasterization techniques allow, real-time rendering applications would utilize path tracing to solve the rendering equation approximately [16, 12].

However, the inherent time-complexity in accurately evaluating the rendering equation using Monte Carlo integration does not lend itself well to real-time rendering. Compared to offline rendering where often hundreds or even thousands of samples per pixel are gathered to produce quality results, even the fastest of path-tracers cannot perform much better than a few samples per pixel at 1080p 30Hz [5]. Even with new top-of-the-line hardware such as NVIDIAs RTX-series, that dedicate specialized hardware to perform ray-tracing, fully path traced scenes cannot be realistically rendered at real-time speeds. As a result of such a low sample count, any image sampled using Monte Carlo integration contains a significant amount of noise.

The development of techniques that can produce good looking imagery from such noisy input is thus critical. Even for offline rendered images, where noise is not as prevalent, similar techniques are used to denoise images and improve quality [2, 34]. Over the last few years, many techniques for reconstruction and denoising of images with low sample counts at interactive speeds have been developed [5, 29, 20, 38]. Most of these produce good looking results but are limited in the types of images they can process. In particular, deep neural networks have been shown to produce particularly good results for denoising both in real-time scenarios [5, 30] and in offline rendering [2, 34].

Chaitanya et al. present a deep autoencoder capable of reconstructing images with very low sample counts at interactive speeds [5]. While their approach indeed pro-

duces good looking results, it fails to incorporate stochastic primary ray effects such as motion blur and depth of field. Such effects are important aspects of producing realistic looking visuals in some applications. Consider a game with fast-moving objects, where motion blur is a vital part in ensuring visual quality, or a set of development tools used during movie production, where being able to preview the effects of depth of field could be crucial in ensuring a productive workflow for the artist.

1.2 Problem definition

Deep neural networks have been shown to work particularly well for real-time denoising of Monte Carlo sampled images [5, 30]. Many of the recent and most popular techniques produce good looking results, but do not handle stochastic primary ray effects such as depth of field and motion blur [5]. As mentioned in Section 1.1, these effects can be an integral part of producing visuals of sufficient quality. It is thus interesting to consider ways in which deep neural networks can be used to denoise these effects at interactive speeds.

This thesis aims to develop a denoising technique based on a deep neural network, capable of denoising images with stochastic primary ray effects at interactive speeds. Focus is put towards the support of stochastic primary ray effects by extending an already developed network to support these features. To solve this problem, we have to provide an answer to the following statement; How do we have to extend the network with additional inputs and layers, for it to recognize these stochastic primary ray effects?

We define interactive speeds to be a maximum of 75ms per frame, which means that to keep the denoising process interactive, the time per frame should not exceed 75ms or ca 13 fps. This time limit was chosen with the reasoning that going above 75ms would significantly impact the interactivity of an application.

1.2.1 Limitations

To further narrow the scope of the thesis, we limit the development of a deep neural network by extending an already developed network. In particular, we extend the network presented by Chaitanya et al. [5]. Because of this, we do not consider how the technique we develop could be generalized or adapted to other networks. Focus is instead put towards trying to show that deep neural networks can be used to denoise such phenomena at interactive speeds.

We have chosen only to consider the depth of field effect. Other stochastic primary ray effects are certainly interesting, but given the limited amount of time we have

available, we cannot successfully consider them all. Depth of field was chosen based on our interests, its simplicity over motion blur [25], and plausible ideas on how to achieve successful results at initial stages of the thesis.

Also, due to restrictions in time, we do not develop an interactive application that utilizes the network for denoising. Instead, the network is given input from images that have been pre-rendered, and evaluation is performed on the output. We consider the network successful if it shows clear differentiation in the denoising of points in focus and out of focus, and is capable of doing so in under 75ms.

Furthermore, we only consider images that have been sampled using path tracing. No time was spent on denoising images generated using other ways of approximating the rendering equation.

1.3 Ethical Considerations

There are important ethical aspects to consider when dealing with software capable of advanced image processing. For the software developed during this thesis, one particular thing that comes to mind is denoising and reconstruction of images that were intentionally blurred or otherwise distorted by its creator. While this is not the intended use case of the developed software, given that the software is learning-based, it might be possible to train it at such tasks.

We argue in a few different ways that this work does not contribute to such unethical use cases. First, the extension to support depth of field does not make any significant contribution in this regard. Second, other already developed offline approaches would cover mostly the same use-cases. Given that these offline based approaches do not have restrictions on processing time per frame, they will always perform better or at least on par at such denoising or reconstruction tasks. Third, the network requires a set of additional input buffers be provided for each frame to produce good results. Such additional buffers would be almost impossible to obtain unless one is rendering the scene themselves, in which case the creator mentioned above would be oneself, and the problem would be obsolete.

2

Previous Work

Zwicker et al. [38] provide a good overview of the area before 2015. They divide the denoising techniques into two separate types. Techniques that, based on analytical analysis of the light transport equation, enhances the Monte Carlo samples and reconstruct images this way [38], as well as techniques that statistically analyzes Monte Carlo samples and reconstructs images using this information [38]. We note that none of the techniques summarized by Zwicker et al. [38] use a data or learning-based approach to reconstruction, in fact, Zwicker et al. end their overview by stating that these approaches are still largely unexplored. Even so, we provide brief descriptions about the papers that are most relatable to our thesis.

From the techniques that perform analysis of the light transport equation, quite a few include support for depth of field effects [4, 24, 25]. Mehta et al. use axis-aligned filters to successfully reconstruct and adaptively sample Monte Carlo sampled images including soft-shadows, diffuse and moderately glossy indirect illumination, and depth of field [22, 23, 24]. Munkberg et al. [25] build on the work from Belcour et al. [4], developing sheared filters that are capable of reconstructing depth of field and motion blur. Both Mehta et al. and Munkberg et al. reconstruct frames at interactive speeds, but do not take a learning-based approach to reconstruction, however.

The approaches mentioned were developed by explicitly analyzing the light transport equation with respect to depth of field. The statistically-based approaches summarized by Zwicker et al. are generic with respect to stochastic primary ray effects, as they reconstruct pixels by building weighted averages of Monte Carlo sampled pixels — even if some techniques might produce worse results with them present [38]. A common approach among such statistical methods is to utilize auxiliary buffers to improve reconstruction results. Dammertz et al. use auxiliary buffers to construct an edge-avoiding à-trous wavelet filter, capable of reconstructing frames at highly interactive frame rates [8]. Li et al. [15] and Rouselle et al. [28] both use auxiliary buffers, as well as SURE [32] error estimation and multiple cross-bilateral filters to perform reconstruction. Both Li et al. and Rouselle et al. fail to reconstruct frames at interactive frame rates, spending seconds or minutes reconstructing each frame.

Bauszat et al. [3] extend and generalize the work done on adaptive manifold filtering

by Gastal and Oliviera [9]. They generalize the adaptive manifold filtering to support an arbitrary amount of samples per pixel and improve the efficiency of the sweep-blur filtering developed by Shirley et al. [31]. Their approach supports the reconstruction of frames with global illumination and depth of field at real-time reconstruction speeds but does not apply a learning-based method.

Building on the work by Dammertz et al. [8], Schied et al. [29] has recently developed a successful real-time reconstruction filter. They use temporal filtering and project previous frames forward to effectively increase the number of sample count and improve temporal stability. Additionally, they split the reconstruction into direct and indirect lighting to improve reconstructed results in poorly sampled shadow edges. They reconstruct frames at approximately 10 ms in 1080p resolution, but specifically mention stochastic primary ray effects, such as depth of field, as a limitation in their method [29].

Mara et al. developed a similar approach to Schied et al. [29] Mara et al. use a cross-bilateral filter, temporal filtering and forward projection of previous frames, and split the reconstruction into multiple parts. In contrast to Schied et al., they split the indirect lighting into two parts of diffuse and glossy lighting. This improves reconstruction results in reflective surfaces [20].

Bako et al. [2] at Disney have recently developed a learning-based denoiser that uses convolutional neural networks to perform denoising of Monte Carlo sampled images. Their network takes low sample count images and auxiliary buffers as input, and predicts a kernel of weights. The predicted kernel is applied across the image to perform denoising. They report that predicting kernels improves convergence significantly compared to training the network to produce a pixel color directly [2]. Their network reconstructs a single frame at 1080p in about 12 seconds.

Vogels et al. [34], also at Disney, use a similar idea of kernel predicting convolutional neural networks. They provide theoretical analysis on why kernel predicting networks perform better than the direct prediction of pixel colors, as reported by Bako et al. [2]. They develop a convolutional neural network that utilizes residual blocks [11], skip connections, and temporal information from multiple frames to improve reconstruction results [34]. They report a reconstruction speed of 10.2 seconds at 1920x804.

To our knowledge, there are no learning-based denoisers capable of handling distribution effects such as depth of field at interactive frame rates.

3

Theory

This chapter gives explanations to some of the theory used during the thesis. A certain level of basic understanding is presumed, particularly in machine learning areas. We start by explaining path tracing and how to approximately solve the rendering equation. We then move on to image denoising and its applications for Monte Carlo rendered images. Finally, we describe the core machine learning concepts that were used to develop the model explained in Chapter 4.

3.1 Path Tracing

We provide a general overview of the rendering equation and how it can be calculated using ray tracing. A thorough walkthrough of the rendering equation, path tracing, and many more related topics can be found in the excellent *Physically based rendering: From theory to implementation* [27], which has recently been made freely available online.

Path tracing is a way of rendering images by simulating how light behaves as it bounces around in a 3-dimensional space. Kajiya introduced the rendering equation and path tracing as a method for solving said rendering equation in 1986 [12]. The equation, shown in 3.1, describes the outgoing radiance from a point x in the direction w_o. For some readers, at first sight, the equation might look intimidating. We split the equation down into two parts to understand it better. The first part $L_e(x, w_o)$ describes the amount of radiance that is being emitted from a point x towards the direction w_o. The second part of the equation, $\int_\Omega f_r(x, w_i, w_o)L_i(x, w_i)(w_i \cdot n)dw_i$, describes the amount of incident radiance towards the point x that is being reflected again in direction w_o. In other words, the rendering equation at some point x equals the amount of light that the point emits in a given direction w_o, plus the amount of light that bounces off the point x towards the same direction w_o.

$$L_o(x, w_o) = L_e(x, w_o) + \int_\Omega f_r(x, w_i, w_o)L_i(x, w_i)(w_i \cdot n)dw_i \tag{3.1}$$

To render images using the rendering equation, we need to understand the integral in further detail. Consider that the irradiance towards a point x can be expressed as the integral of incident radiance from all directions in x's hemisphere Ω. Thus to evaluate the irradiance towards x, we want to find the incident radiance from all individual directions into x. Simply evaluating all directions into x would be unreasonable, so we have to approximate the integral in some way. For now, assume that we can iterate over all incoming directions, and focus instead on how to calculate the incident radiance from one direction. We split the integral into three parts; First, the bidirectional reflectance distribution function (BRDF) $f_r(x, w_i, w_o)$. The BRDF describes the fraction of light incident from direction w_i that is also reflected in direction w_o. Second, the dot product $w_i \cdot n$ between the surface normal n and the incoming direction w_i. This dot product describes how the irradiance from point x, coming from direction w_i, is weakened by the angle between n and w_i. Finally $L_i(x, w_i)$ is the incident radiance towards the point x from direction w_i. As mentioned, together, these three parts describe the light that hits a point x from direction w_i and is reflected again in direction w_o.

To reach this understanding, we assumed that we could iterate over all incoming directions in x's hemisphere. This assumption does not hold. Iterating all incoming directions would take an infinite amount of time and resources; thus, we need to find a way to evaluate the integral without calculating all possible values. Kajiya proposes to approximate the integral using Monte Carlo integration [12]. In doing this, we trade processing time for potential noise in the rendered image, i.e., the more samples we take of the integral, the less noise we have in the image.

Note that the rendering equation is recursive since the irradiance in one point is dependent on the incoming radiance towards that point. Thus to evaluate the rendering equation at one position, we have to evaluate the rendering equation at all positions were the outgoing radiance of that point contributes to the incoming radiance of the current point, and so forth. To provide a method of evaluating this recursive structure, we utilize ray tracing within the scene.

While it might seem logical to start tracing rays from the position of each light within a scene, this would be extremely inefficient. Consider all the light rays emitted from each light that never ends up at the viewer's eye. There is no way of knowing which rays these are when starting ray tracing at the light position, and so we would waste resources tracing rays that never contribute to the final image. Instead, we start tracing light rays at the viewer's eye and into the scene, effectively tracing light rays backward. This is often referred to as backward ray tracing.

To achieve this, we send one ray into the scene for every pixel on the image being rendered. Once the ray intersects an object within the scene, we evaluate the rendering equation at this position. Evaluating the rendering equation requires us to send a new ray into the hemisphere Ω of that point, and evaluate the rendering equation at the new intersection position, and so forth. This recursive process is stopped once the next ray either does not hit any object within the scene or when

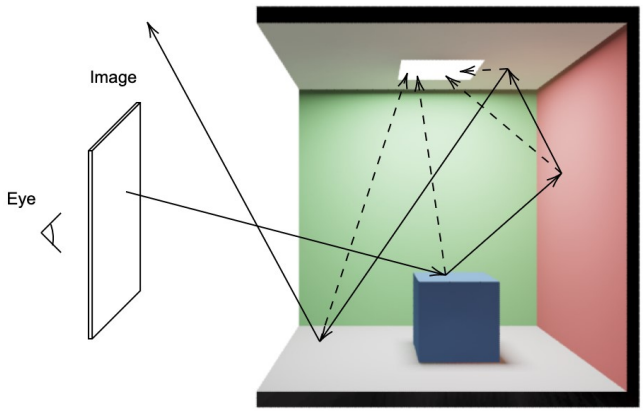


Figure 3.1: The recursive process of path tracing performed on a simple scene. The solid arrows represent the light ray originating in the pixel. The dashed arrows represent direct sampling of light at each position where the solid arrow ray path intersects an object.

it hits a light. See Figure 3.1 for a visualization of this recursive process.

In some scenes, the probability of a ray hitting a light might be very small or none at all, causing light rays to be potentially unbounded in length. To avoid wasting resources, we want to avoid light rays becoming too long. We achieve this by bounding the light rays using Russian Roulette [27]. Russian Roulette randomly stops recursion, cutting the light rays short with a random probability every time we trace another ray recursively. Additionally, with such a low probability of hitting a light, only those few ray paths that hit a light eventually contribute to the final color of a pixel. To find those paths more often, it turns out we can split the integral in the rendering equation into two separate parts, one part representing the *direct light* and one part representing *indirect light*. In other words, the light that comes directly from a light source, and the light that has bounced off other points within the scene, before hitting the current point. To account for this, we trace multiple rays recursively at every intersection point, one ray towards every light, and one ray towards a random direction in the hemisphere. Note that the ray being sent directly towards each light only contributes to the points illumination if it reaches the light and is not blocked by some object within the scene. Tracing separate rays for direct and indirect light at every intersection this way is often referred to as next event estimation. See Figure 3.1 for a visualization on how this works.

To generate images using this method, we trace rays from each pixel in the image multiple times. We combine the results for all rays originating at a pixel by averaging the results for that pixel. The number of times we trace a ray from a pixel is often referred to as the number of samples per pixel (spp). See Figure 3.2 for an example of how the different number of samples affect the final result. Note that taking additional samples for all pixels corresponds to taking more samples of the Monte Carlo integration of the rendering equation, and thus it reduces noise in the final render.

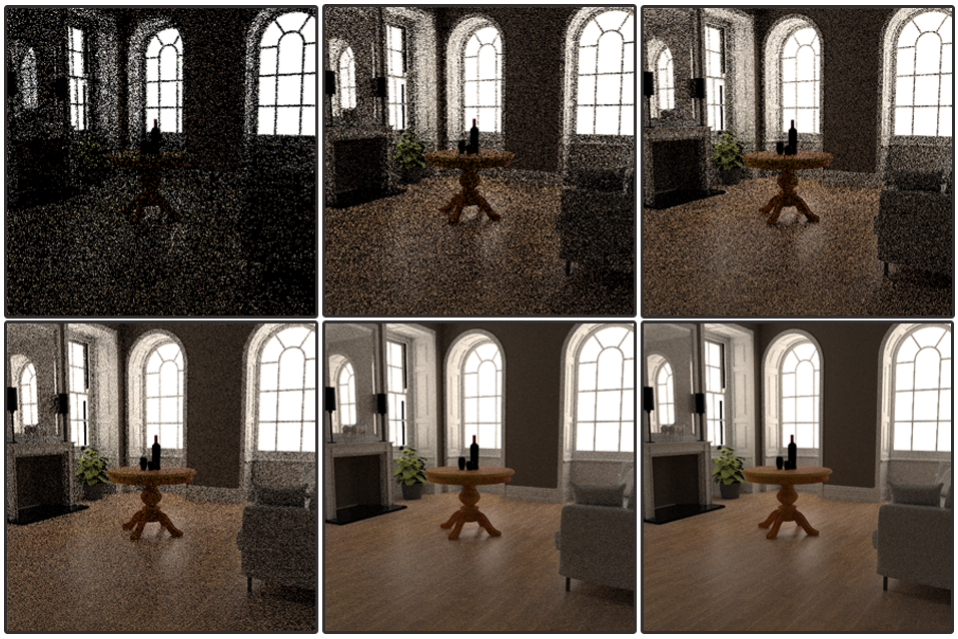


Figure 3.2: Showing different stages of the path tracing process, having traced different amount of samples per pixel. From top to bottom, left to right: 1spp, 8spp, 16spp, 256spp and 512spp

3.1.1 Depth of Field

Depth of field (DOF) is the distance between the nearest and farthest point in an image where objects are considered to be in acceptably sharp focus. All objects outside this area are thus considered out of focus. DOF is a widely used effect by artists and photographers. It is commonly used to draw attention to specific areas in a frame or to improve the visual appeal in the final image. An example of depth of field is shown in Figure 3.3.

Whether or not an object is considered to be acceptably sharp is determined by the circle of confusion. The circle of confusion is a measurement of the area in which light rays from a single point collapse and form a circle on the camera sensor. The circle of confusion size is determined by the camera's aperture and focal length, as well as the size of the medium displaying the image. In general, a point is considered acceptably sharp when the circle of confusion is smaller than a single point on the medium were the image is displayed. In our application, a point is thus considered acceptably sharp when the circle of confusion is smaller than the area covered by a single pixel.

Rendering DOF using ray tracing was introduced by Robert L. Cook et al. in 1984 [7]. By sending rays through a virtual lens before they enter the scene, we can simulate how light refracts on hitting the lens of a real camera. In a real camera, it is the bending of the light as it passes through the lens that causes DOF effects. Accurately simulating how light rays bend through a glass lens before they pass into the scene would undoubtedly produce DOF effects in our final render. However, by utilizing our virtual lens properly, we can significantly simplify the process, avoiding many calculations needed to simulate the bending of light rays as they pass through the glass of the lens.

To render DOF using our virtual lens, we need to determine a focal distance f_D as well as an aperture A of the lens. Note that in a regular camera, one cannot change the focal distance directly. Instead, the focal distance is a result of changing the aperture as well as the focal length.

Having values for both f_D and A, rendering images with DOF effects is reasonably straight forward. For every primary ray R that was being sent into the scene from a pixel, we instead generate a secondary ray S. The secondary ray S has its origin at the lens and travels into the scene similar to how R did. We use this secondary ray as the light ray that contributes to the color of the original pixel where R originated. As a result, we effectively have a light ray that originates in the pixel and bends as it goes through the lens.

To generate S, we need to find an origin position S_O as well as a direction S_D. To find the origin S_O, we sample a position on the lens using the radius A and set S_O to that sampled position. To find the direction S_D, we use the focal point f_P in the scene and get the direction between the f_P and S_O. We get that $S_D = \frac{f_P - S_O}{|f_P - S_O|}$. To calculate the focal point f_P within the scene, we use the focal distance we defined and the direction of the primary ray R_D; we get that $f_P = f_D * R_D$.

Figure 3.3: A comparsion between the same image rendered with and without depth of field.

3.2 Image Denoising & Reconstruction

As mentioned in Section 3.1, the amount of noise in an image is dependent on how many samples per pixel we take for that image. The amount of noise in an image is therefore highly dependent on the amount of time we can spend rendering that image. For offline rendering applications, this might be several hours. In interactive applications, we do not have the privilege of such large amounts of time per frame. Instead, for an interactive application at 30fps, we could only spend around 33 milliseconds rendering per frame.

In practice, noise is unavoidable in most scenes. Even for offline rendering, any reasonably complex scene with a reasonable amount of rendering time produces some noise. It is thus a vital part of the rendering process to apply some denoiser as a post-processing effect after Monte Carlo rendering has been completed. There are many examples of production level renderers doing so [2, 6].

Applying a denoiser as a post-process is not the same as an approximation of the rendering equation. A side effect of this is that it is not as accurate, and can introduce artifacts into the final render. The benefit of the denoiser is that it is significantly faster than collecting more samples per pixel. In an optimal scenario, we would be able to build a denoiser that produces similar enough results that the difference would be barely noticeable.

For offline rendering this is certainly possible [2, 6]. We can spend a significant amount of time running a denoiser for each frame since producing noise-free images by adding additional samples would take large amounts of time. Additionally, given a reasonable amount of time spent rendering, there is a lot of information available for a denoiser to work with. We can thus apply sophisticated filters and produce high-quality results.

For interactive rendering applications, we are not so lucky as to have such favorable conditions under which to apply a denoiser. The amount of information available to an interactive denoiser is in the approximate range of a handful of samples per pixel, compared to the hundreds or even thousands of samples for an offline denoiser. Additionally, we cannot spend as long performing the denoising operation. Again, take an interactive application at 30fps where rendering and denoising combined would now have to be performed in around 33 milliseconds. Considering these factors, the complexity of interactive denoising of Monte Carlo sampled images becomes immediately apparent.

3.3 Deep Learning

We assume that the reader is knowledgeable in most basic machine learning concepts. However, since large parts of the thesis consist of work on a deep learning model, we spend this section explaining some of the core concepts we use to develop our model. In particular, we describe the basics on convolutional neural networks, explain what an autoencoder is and how they can be used to reconstruct data, as well as talk briefly about max pooling. More in-depth descriptions can be found in the book *Deep Learning* [10].

3.3.1 Convolutional Networks

A convolutional neural network (CNN) is a type of neural network introduced by LeCun et al. in 1989 [14], which uses convolutions instead of matrix multiplication in at least one of its layers [10]. CNN's can be used for processing data in any form of a grid, but are mainly used for analyzing visual media.

In general, convolution is a mathematical operation that calculates the integral of the product between two functions $F(x)$ and $W(x)$ [10]. When working with convolutions in CNN's we often refer to $F(x)$ as the input and $W(x)$ as the kernel of the convolution.

The mathematical definition of a convolution applies to any two functions. In the context of CNNs, we can redefine and simplify the convolution slightly to better fit our purpose. Working with different types of media on the computer, we often represent the media using a discrete set of values. Consider, for example, an image represented as a 3-dimensional matrix. Representing an image this way, we can read the input $F(x)$ as a matrix of values. We assume that similarly, the kernel W can be represented using a matrix. The product between these two functions $R = F * W$ is then equal to the element-wise matrix multiplication between their matrices. It follows that the convolution is calculated as the integral over R. Since the possible values of R are entirely represented by the output matrix, calculating the integral corresponds to summing all elements in matrix R.

While this represents a valid convolution, it is uncommon that the size of the kernel exactly matches the size of the input. In theory, given a few restrictions, we can choose any kernel size. The number of dimensions in the kernel must remain the same as in the input. Additionally, it is important that the depth, or the number of channels, in the kernel remains the same as the channel count of the input. Keeping these restrictions in mind, we can change the size of the kernel as we see fit. In practice, it is common that the kernel size remains rather small. Keeping the kernel size small keeps the number of trainable parameters down, as it is the values within the kernel that change during training.

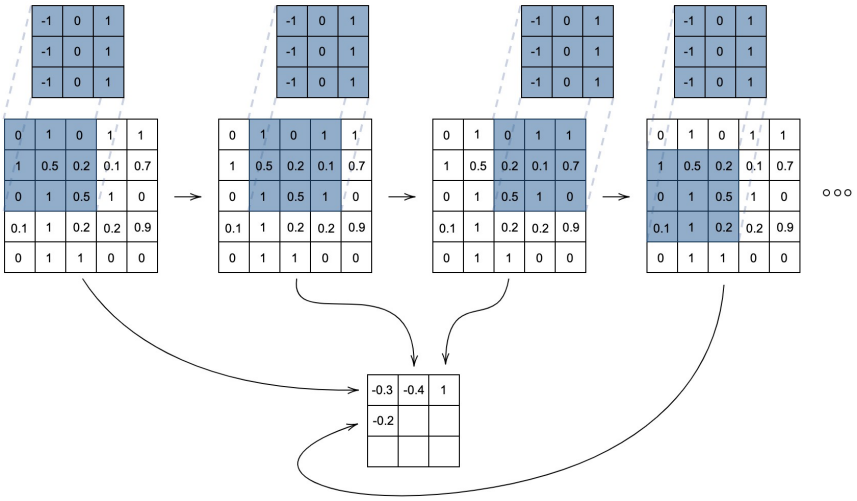
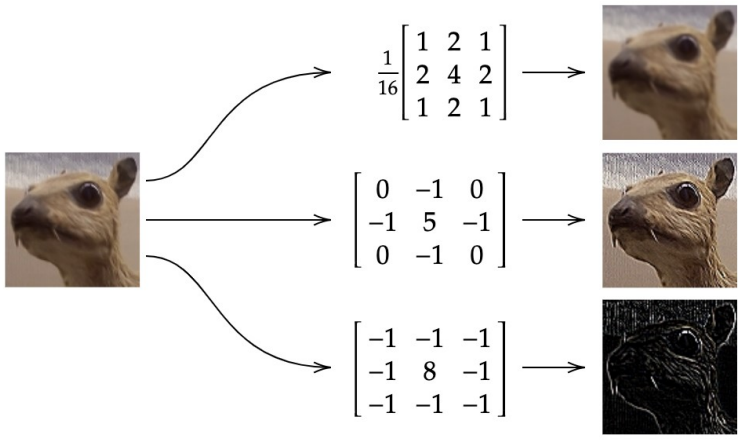


Figure 3.4: An example of applying a 3x3 kernel to a 2-dimensional image. We start the sliding position with the kernel in the top left most corner of the input, each sliding step the kernel is moved one step towards the right. Once the kernels right edge reaches the right edge of the input, the next step the kernel is wrapped around to the left side of the input and moved down one step. We keep sliding the kernel until the bottom right corner of the kernel hits the input's bottom right corner.

Changing the kernel size, even with these restrictions in mind, breaks the element-wise multiplication between the input and the kernel. To correct for this, we add an extra step to the convolution process. With a smaller kernel size, there are not enough values in the kernel to perform element-wise multiplication with the input. Instead, we slide the kernel along the dimensions of the input. We skip sliding along the depth or channel dimension since its size remains the same. While sliding the kernel along the input dimensions, we perform a convolution at each position, resulting in the application of the same small kernel across the whole image. The output matrix is then the derived scalar for each convolution placed at the kernels center position. See Figure 3.4 for an example of this process.

The attentive reader might have noticed that sliding the kernel along the input this way can potentially reduce the dimensions of the final matrix compared to the input. This happens because the center of the kernel never reaches the edges of the input. We can solve this problem by applying padding to the edges of the input. Essentially we pad the input with zeros in all dimensions but the channels, allowing the center of the kernel to reach the edges of the input. As a result, the dimensions of the output matrix matches that of the input.

During training, the values in the kernels are changed to detect different features in the input. To illustrate the usefulness of such kernels, we provide a few example kernels and the result of applying them to images. See Figure 3.5.



Original images provided by Michael Plotke. Licensed under CC BY-SA 3.0. The above image has been modified and is licensed under the same license. Original content accessed from `https://en.wikipedia.org/wiki/Kernel_(image_processing)`

Figure 3.5: The results of applying a few different types of sliding convolution kernel across an image. The first kernel produces an approximation to the 3x3 Gaussian blur, the second produces a sharpness effect, and the final kernel is an edge detector.

3.3.2 Pooling

Pooling is a type of layer often used in convolutional neural networks. It takes some input and reduces the dimension of that input, in turn reducing the number of trainable parameters and computations needed to evaluate the network. By reducing the number of trainable parameters, it also helps avoid overfitting the network.

Reducing the input's dimensions is done by grouping the values in the input, and deriving a new value for each group. The size of output from the pooling layer depends on the stride and size of the pooling layer. Derivation of a new value for each group can be performed in multiple ways, the most common of which is using the *max* operator.

Performing the pooling operation is similar to how sliding the kernel across the input works in CNN's. We explain the process using the max operator and thus explain the process of a max-pooling layer. A filter, similar in concept to the kernel in convolutions, is slid along the dimensions of the input. Each time it moves some number of steps over the input corresponding to the stride of the layer. Each filter evaluation, the maximum value in the input that overlaps the filter, is taken as the value of the output matrix. See Figure 3.6 for an illustration of the process.

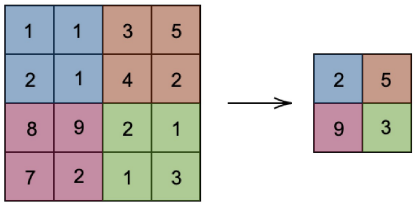


Figure 3.6: Illustrates the process of a max pooling layer. The pooling layer has a filter size of 2x2 and a stride of 2. Each color represents the area covered by one filter application.

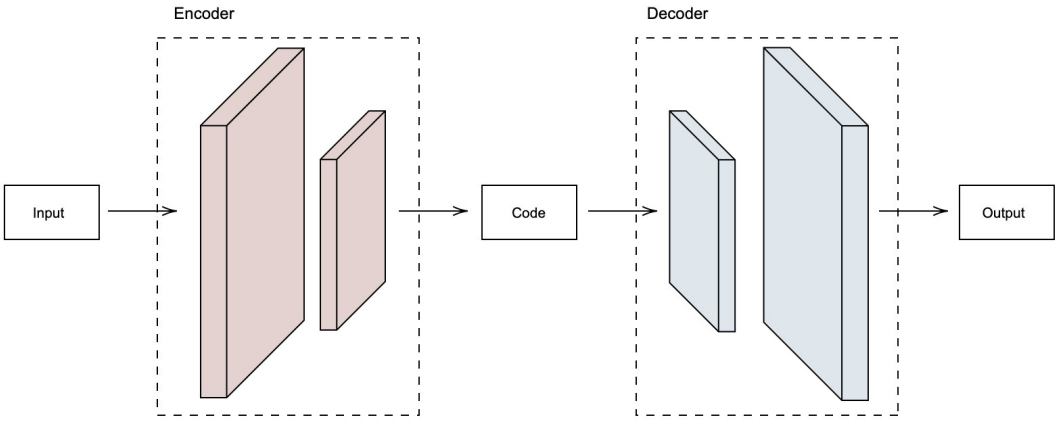


Figure 3.7: An autoencoder with a deep neural network on each side of the code. The encoder converts data into the internal *code*. The decoder takes the code and tries to reconstruct the original input provided to the encoder.

3.3.3 Autoencoders

Autoencoders have been part of machine learning research and applications for many decades [10]. In recent years, they have been one of the most prominent methods in generative modeling [10]. The applications of autoencoders are many, both in image processing and other areas, but some relevant works include lossy image compression [33] and anomaly detection [1].

In its most simple form, an autoencoder is a neural network that attempts to learn to copy its input to its output. Autoencoders consists of two separate operations as well as an internal data representation often referred to as the *code*. The two operations lie on each side of the code, effectively converting data to and from it. The encoder takes some input $\mathbf{x}$ and applies a function $E(\mathbf{x}) = h$, converting an input $\mathbf{x}$ to code h. The second operation, known as the decoder, takes input in the form of a code h and tries to reconstruct the original input $\mathbf{x}$ again by applying another function $D(h)$. Figure 3.7 shows a basic autoencoder setup that uses this structure.

A neural network that copies its input over to its output would not be particularly useful. Instead, autoencoders are built only to be able to copy their input approximately. It is essential that the autoencoder cannot merely learn the identity function to perfect its training. There are multiple ways to ensure this does not happen, and different solutions result in different types of autoencoders. In turn, different autoencoders are used to learn different properties.

Restricting the dimensions of the code h to be lower than that of the input $\mathbf{x}$ is one way to ensure the autoencoder does not learn the identity function. If the code has a lower dimension than the input, it is impossible for $E(\mathbf{x}) = h$ to be true if E is simply the identity function, since the encoding function E is required to encode the input in some other way. Autoencoders that follow this principle are known as undercomplete autoencoders. Given this restriction, undercomplete autoencoders try to minimize a loss function $L(\mathbf{x}, D(E(\mathbf{x})))$, where L is a metric that penalizes the network for the reconstruction result $D(E(\mathbf{x}))$ being different from the original input $\mathbf{x}$.

An autoencoder type that is of particular interest in this thesis is the denoising autoencoder (DAE). Instead of restricting the dimension of h to avoid learning the identity function, a DAE is trained to reconstruct a corrupted version $\mathbf{x'}$ of the input into the uncorrupted version $\mathbf{x}$. The loss calculation is given by $L(\mathbf{x}, D(E(\mathbf{x'})))$. By carefully inspecting the loss function, it is clear that learning the identity function would not be optimal, since $\mathbf{x'} \neq \mathbf{x}$.

We can indeed utilize DAEs when trying to denoise Monte Carlo renderings, in fact, the concept is quite straight forward. Take the original input $\mathbf{x}$ as a render of reference quality, and the corrupted input $\mathbf{x'}$ as a render of the same frame, but with a significantly lower sample count. We can then train the DAE to reconstruct the reference quality frame from the lower sample count version. While it is straight forward to apply the concept of autoencoders to denoising Monte Carlo renderings, it is far from trivial to have the autoencoder learn the proper encoding and decoding functions. We walk through our implementation of these functions in Section 4.3.1.

When reducing the dimension of the input significantly in some manner, e.g., using convolutions or max pooling, we may lose essential information to the reconstruction process [19]. Mao et al. show that we can retain detail in the reconstructed image by adding skip connections between convolution and deconvolution stages. Clean inputs pass through skip connections, allowing the deconvolution step to combine convoluted and clean inputs, retaining detail in the final output. Chaitanya et al. show that a similar approach can be taken with an autoencoder [5]. In an autoencoder, we add skip connections between encoder and decoder stages, effectively achieving the same result as Mao et al.

4

Methods

In this chapter, we provide an in-depth description on the implementation of the applications developed during our thesis. In Section 4.1, we give an overview of all systems and how they work together. In Section 4.2 we detail the implementation of a path tracer. Finally, in Section 4.3, we describe the development of a denoiser that can process images with DOF effects, produced by the path tracer described in the previous section.

4.1 Overview of System Setup

The system can be split up into two separate subsystems, a path tracer, and a learning-based denoiser. These combine into a system that tries to train a learning-based denoiser to reconstruct Monte Carlo sampled images with low sample count into images of comparable quality to one with a higher sample count.

To train the denoiser, we need access to a training and evaluation dataset. We must be in full control of generating the dataset, to allow any form of customized generation of data points, potentially improving convergence significantly.

To generate the dataset, we developed a path tracer. The implementation is exclusively in C++ and is entirely CPU based. The path tracer renders a provided configuration, specifying the scene, render quality, animation frames, lights, and output folders. For each configuration, the path tracer produces all necessary data points for the denoiser to train and evaluate its performance. Images are rendered as 512 by 512 pixels and are being saved to disk continuously as they finish rendering. The denoiser then reads the files from disk during initialization of the datasets.

The denoiser is a neural network based on the DAE concept. The network is configured to train on data generated by the path tracer. For each training session, the network tries to learn the appropriate behavior of reconstructing images from a single scene. Depending on the configuration, the path tracer renders a few hundred or a few thousand frames. The network is then configured to train on a subset of

all rendered frames, and evaluate its performance on the remaining frames. Generalization of the network is thus in the context of a single scene.

During a training session, the best weights and losses for the network are continuously saved to disk, allowing for evaluation and visualization of the network's performance throughout the training session. Additionally, we can later use each saved model in a separate application by loading the network back into memory and use it solely for denoising purposes.

4.2 Path Tracer

While the aim of the path tracer is not to be a production-ready image renderer, we have to produce images of sufficient quality to evaluate the performance of the denoiser. To properly assess the performance of the denoiser, it is important that our path tracer is capable of rendering images of comparable quality to the ones that would typically be used in a real-time rendering application. To produce such renders, a set of key features were identified as essential parts of the path tracer. The following sections walk through each feature in detail. Here we provide a general summary of the path tracer's execution life cycle. We suggest that readers refer back to this summary as they read through the upcoming sections.

The execution process of the path tracer follows the description given in Chapter 3 rather closely. Originating at the camera position, we trace a ray towards the scene in a direction determined by considering the position of the pixel, and the lens' focal length and aperture. We use Intel Embree [35] to trace the ray throughout the scene. If Embree reports a hit within the scene, we evaluate the rendering equation at that position. If not, we determine the result of that ray from the scene background. Evaluating the scene background consists of determining what position the ray hit our environment map and use that point as the final result of the ray traversal.

Evaluating the rendering equation is done in three steps. **First** we perform direct light sampling by picking a random light in the scene and trace a shadow ray from our point towards the light. If the shadow ray intersects another object within the scene before hitting our light, we determine the point where the ray originated to be in shadow. In this case, the light does not contribute to the illumination of the originating point. If the shadow ray reaches the light, we evaluate the contribution of illumination to the originating point provided by the light. **Second**, we importance sample the BRDF for the next ray direction to continue tracing the ray. The BRDF is evaluated using the newly sampled direction. **Third**, we recursively trace the sampled path from the BRDF within the scene using Embree and evaluate the rendering equation at each point of intersection. We stop recursion once the ray does not hit an object in the scene, or the ray is canceled from Russian Roulette.

4.2.1 Lighting

To properly light the scenes used during the thesis, we deemed it necessary to implement two types of light sources. First, an environmental light source that could be used to increase realism. Second, a positional based light that could be used to light specific areas of a scene with higher precision. These two types of light certainly do not cover all lighting situations but are enough to produce sufficient training material.

We implement support for environmental lighting by loading and sampling HDRI images. Images are loaded into a simple data structure that can be sampled using UV-coordinates over the image. We also construct a 2D-Distribution over the intensity along the texels for each HDRI map. We use this distribution to importance sample the environment light, allowing us to improve convergence significantly while using environment lights [27].

For positional lights, we implemented one-directional area lights. These lights are importance sampled with the reciprocal to their area, significantly improving convergence.

To improve convergence and overall speed further, we implement next-event estimation, multiple importance sampling [27] when sampling all types of lights, and single light sampling [27] to improve speed in scenes with multiple light sources.

4.2.2 Materials & Shading

An essential part of having the path tracer generate realistic visuals are good simulations of different types of materials. Different types of material models allow for the simulation of different materials. We must choose a model capable of rendering all the materials we need. The subset of materials that we require is reasonably small, and so a relatively simple model is appropriate.

Our path tracer implements support for diffuse and specularly reflective materials. We implement diffuse materials through the Lambertian BRDF, and specularly reflective materials through a partly implemented GGX[36] BRDF. When defining each material, we have to specify the base color, emission color, metallic ratio, index of refraction, and roughness parameters. We read these parameters through textures that are mapped onto each object, allowing parameters to vary across the surface of an object, improving detail further.

We decided not to implement support for refractive materials. Some of the scenes rendered using the path tracer would undoubtedly have looked better with support for refraction, but it is not a requirement to evaluate the performance of the denoiser. Implementing support for these types of materials would have been too

time-consuming. However, the GGX BRDF is fully extendable to support such materials as well [36].

In some shading models, roughness is referred to as shininess; these two parameters describe the same phenomena. In our model, we use the roughness parameter, while in the scene archive were we retrieved our scenes [21], the material files use a shininess value. To correct for this, we convert the shininess value from the scene files into roughness while loading the scene. We apply a simple formula given in equation 4.1, where R is the roughness and S the shininess of the material. This formula converts shininess values in the approximate range $0 - 5000$ into roughness values in the range $1.0 - 0.02$.

$$R = \max(\sqrt{2/(S+2)}), 0.02) \tag{4.1}$$

4.2.3 Depth of Field

When adding support for DOF in a path tracer, there are multiple levels of complexity one can choose. We implement a relatively simple model by replacing the usual pinhole camera model with a thin lens approximation [27]. Our shutter is a simple circular shape, and we do not support changing between different types of camera models. Our implementation closely follows the description given in Section 3.1.1.

Given the circumstance in which we are rendering, DOF is perhaps one of the more essential features in our path tracer. It is important to consider how different types of implementations might impact the results. It turns out that the simple thin lens approximation serves our purposes quite well. Adding support for additional features such as different types of shutter shapes would significantly increase the complexity of the denoiser's task. It seems appropriate that one would try to achieve good results with a simple DOF model, and then add additional complexity later.

4.2.4 Auxiliary Features

To improve the convergence of the denoiser, we provide a set of auxiliary feature frames as input to the network. We get back to this and provide further detail in Section 4.3.2. It is thus a requirement for our path tracer to produce these auxiliary feature frames. In particular, the path tracer needs to support rendering frames with albedo, normal, roughness, depth, and circle of confusion maps.

Rendering most of these frames is relatively straight forward. When determining the direction of the primary ray, before sampling a position over the aperture of the lens, we check if the current path tracer configuration tells us to render an auxiliary

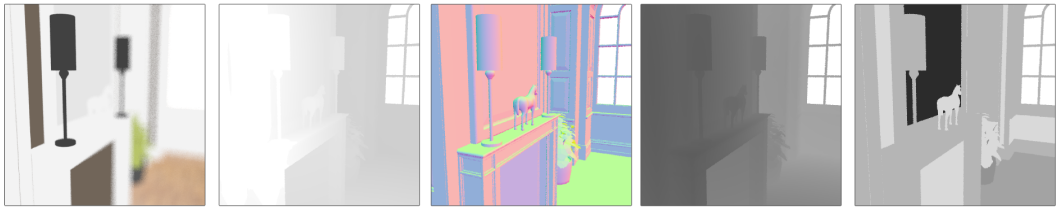


Figure 4.1: Shows all auxiliary buffers rendered by the path tracer. From left to right: Albedo, Circle of Confusion, Normal, Depth, Roughness.

frame. If we are rendering any auxiliary frame except for the albedo map, we skip sampling the lens. Effectively resulting in all auxiliary frames except the albedo map having no DOF effects applied, we provide further motivation on this decision in Section 4.3.2. We trace the primary ray in the scene using Embree, and if we hit an object, we evaluate the value for the auxiliary frame. For the albedo and roughness maps, this evaluation is straightforward. We read a value from the material that the primary ray hit and return it as the result. For normal, depth and circle of confusion maps, things are slightly more involved. Note, however, that none of the auxiliary frames are recursively traced during rendering. We stop tracing paths within the scene after the primary ray has hit a surface.

When reading the normal from the position in the scene hit by the primary ray, we get a normal in world space. One could argue that our denoiser works in view space, always looking at the scene from the camera's point of view. It thus seems appropriate that we should convert the normal to view space before giving it to the denoiser. Additionally, to map each direction onto a color channel, we apply a mapping to the direction of the vector in each dimension as given in Listing 1.

```
auto remappedNormal = vector3(
    (normal.x + 1.0f) * 0.5f,
    (normal.y + 1.0f) * 0.5f,
    (-normal.z + 0.5f) * 0.5f,
);
```

Listing 1: Shows how viewspace normals are converted into appropriate RGB channels being saved to disk and eventually sent as input to the denoiser.

We evaluate the depth of a point within the scene by reading data from Embree. Once Embree returns that the primary ray hit an object in the scene, we query for the length of the ray. Since this primary ray vector originates in the camera, the length of the traced ray equals the depth within the scene relative to the camera. However, a depth buffer is commonly represented with values in the range $0 - 1$ while the depth value provided by Embree can be any positive number. Rendering this buffer to an image thus requires a remapping of the depth value. The mapping is highly dependent on the scene being rendered. We remap the value by dividing the depth value provided from Embree by an approximate total depth of the scene, which we specify manually for each scene.

Evaluating the circle of confusion map at a point within the scene is the most complicated auxiliary feature. We refer the reader to Section 4.3.2 for the reasoning behind the design of this auxiliary feature. In the following paragraph, we describe the value we seek and how we calculate that value.

For each point in the scene where the primary ray hit, we want to determine whether or not we are within the acceptable area of depth of field for the observer to determine the object in focus. If we are within this area, we want to express that. If not, we want to express how far away from this acceptable area we are. First, we have to determine two values that apply for the entire scene; An acceptable and a maximum circle of confusion size we expect to see within our scene. These values are not the same for every scene but depend on the size of the scene. As such, they have to be determined on a scene by scene basis. Second, we calculate the circle of confusion at the point where the primary ray hit the scene. We use this value to calculate the difference between the determined maximum circle of confusion size and the actual size. We then divide this difference by the scene maximum to get a value between $0 - 1$. The closer this value gets to 0, the less in focus the point is. See Listing 2 for the actual code used during the rendering of Fireplace.

```
const auto distanceHitToLens = distance(
    primaryRayHit.position,
    camera.getPositionVector()
);
const auto distanceBetweenCameraAndImgPlane = 1.f;
const auto circleOfConfusionInMM = camera.aperture * (
    abs(distanceHitToLens - camera.focalDistance) /
    distanceHitToLens
) * (
    distanceBetweenCameraAndImgPlane / camera.focalDistance
) * 100.f;

// Example values used during training for our scene "fireplace".
const auto acceptableCoCSizeInMM = .65f;
const auto sceneMaxCoCSizeInMM = 15.f;

auto pixelColor = 1.f;
if (circleOfConfusionInMM > acceptableCoCSizeInMM) {
    pixelColor = (
        sceneMaxCoCSizeInMM - circleOfConfusionInMM
    ) / sceneMaxCoCSizeInMM;
}

return vector3(pixelColor);
```

Listing 2: Shows the code used for rendering the circle of confusion map during the fireplace scene

4.2.5 Importing and Animating Scenes

Being able to iterate on scene setups quickly is an integral part of producing high-quality training data. Support for importing scenes produced in a modeling software, where we can quickly perform changes to the scene setup, is thus a crucial part of our application. We implemented a system capable of importing scenes from the FBX file format, in combination with loading a custom configuration file. By utilizing both of these files, we support importing scenes with models, materials, textures, area lights, environmental lights, cameras, and animations.

When training neural networks, the training data must be diverse enough that the network performs well on the validation set. To generate data points with enough diversity, we need a mechanism to produce multiple different frames within a scene. The path tracer supports animating the position, rotation, focal length, and aperture of the camera. We use this to perform scene fly-throughs, where we progressively change the focal length and aperture, resulting in the animation of DOF effects throughout the fly-through.

Animations work by reading and interpolating keyframe data from the imported FBX file and the custom configuration file. For each frame, we find the two closest keyframes and read their values. We linearly interpolate between these two values with our current frame as the time parameter and update the appropriate value on the camera before rendering the next frame.

4.2.6 Performance & Efficient Rendering

Rendering the approximately 3500 frames used during training in reference quality was going to be unreasonable unless a few performance improvements were made. Since the path tracer is entirely CPU based, rendering is relatively slow compared to many production level renderers. While production-level performance is not a requirement, the path tracer needs to be fast enough that rendering a significant amount of training data can be done without limiting the progress of the thesis. We managed to secure resources on SNIC, a computing cluster available to researchers and students at Chalmers, reducing the need for performance improvements slightly, but not eliminating it. Since there is no graphical context available on the SNIC cluster, we had to make sure that the path tracer could run entirely without it by eliminating OpenGL from the implementation.

Evaluating the performance of the path tracer, it was immediately clear that thread utilization was far from optimal — a crucial issue when running computations on a computing cluster. Multithreading had been naively implemented using OpenMP by specifying that the loop over each pixel should be performed in parallel, resulting in massive CPU spiking with CPU utilization jumping between $70 - 90\%$. We resolved this by dividing each frame into 16 by 16 pixel rendering jobs. We initialize a thread

pool that picks from these rendering jobs and reports back when a block has finished rendering, resulting in the path tracer utilizing between $90-100\%$ CPU at all times, a significant improvement from before.

We also found that the path tracer was spending a significant amount of cycles sampling the BRDF. To mitigate this, we combine the BRDF sampling of the next bounce within the scene and the BRDF sampling for multiple importance sampling. Effectively halving the number of times we sample the BRDF. Additionally, we introduced a cache where we store values that are used multiple times during the evaluation of the material. We also introduced fast inverse square root [17] in areas where accurate normalization is not a crucial part of the calculation.

4.3 Denoiser Implementation

The development of the denoiser is based on the work presented by Chaitanya et al. [5]. Here we present our approach to reproducing their results, as well as extending their work to support images with DOF effects. The network we develop is not identical — even without support for DOF effects — to the one developed by Chaitanya et al., but has been modified to improve convergence during our testing.

Our implementation uses Python and PyTorch [26] to define, train, and evaluate the network. We perform training and evaluation on the GPU using PyTorchs' built-in GPU support. Due to not having an appropriate GPU available, we initially tried training and evaluating the network on the CPU. Our CPU implementation worked well for testing and developing the network, but resulted in unreasonable training times once we run training sessions with more comprehensive data sets.

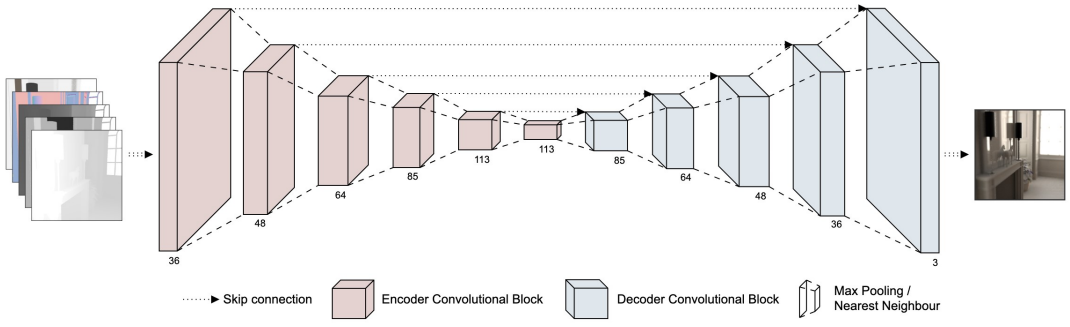


Figure 4.2: Shows the structure of the denoiser.

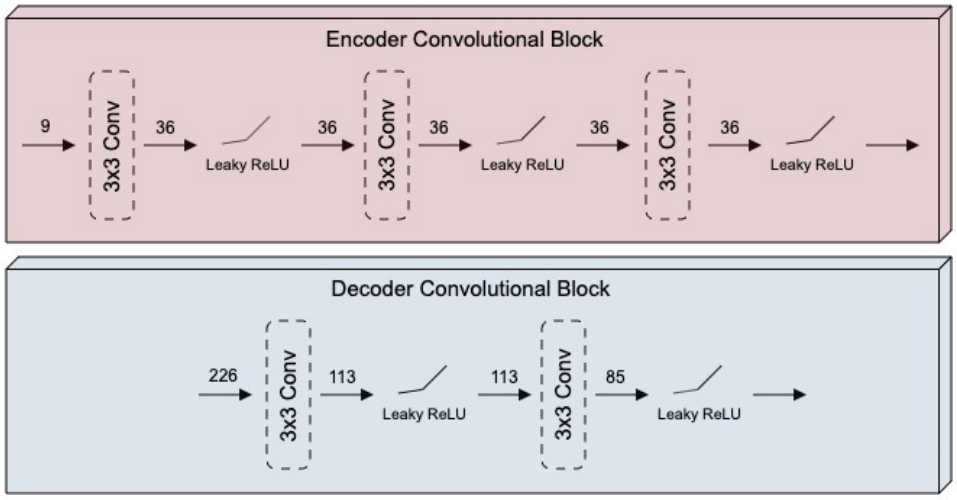


Figure 4.3: Shows the way in which each convolutional block is constructed. The encoder block shown is the first block in the full network, and the decoder block is the first decoder block in the full network. All blocks follow the same structure, except the last decoder block in which the Leaky ReLU activations are gone.

4.3.1 Denoiser Network Setup

The denoiser is a DAE where the encoder and decoder are built up of what we refer to as blocks of convolutions. We also utilize skip connections between corresponding encoder and decoder stages like explained in Section 3.3.3. The input to the network is a single deep image, and the output is a 3-channel image representing the illumination at each pixel. Both the input and the output are covered in further detail in Section 4.3.2.

In the encoder, each block represents three 3 by 3 convolutions. Each convolution has a padding of one and is activated using Leaky ReLU [18], with parameter $\alpha = 0.1$. Each encoder block increases the channel count of the deep image as it passes through the block. The increase in channel count happens in the first convolution, and the two following convolutions then operate on the increased number of channels. Figure 4.3 illustrates the channels changing over the convolutions inside each block. Between each encoder block is a 2 by 2 max-pooling layer, effectively halving the size of the image every time it passes through a block.

The decoder blocks are represented by two 3 by 3 convolutions and a concatenation of the input from the encoder-decoder skip connection. Each convolution has a padding of one and is again activated using Leaky ReLU [18], the exception being the last decoder block which skips activation entirely for all convolutions. Concatenation of the skip connection is performed before any convolutions are applied, which means that the channel count in the first convolution is twice that of the output from the last block. The first convolution reduces the channel count to the same as the one provided by the output of the previous block, before the next convolution increases

it again. See figures 4.2 and 4.3 for illustrations on how this works. Between each decoder stage, we perform a 2 by 2 nearest neighbor upsampling.

The smallest block, or the bottleneck, is very similar to the encoder blocks. The difference is that none of the convolutions change the channel count. Before the bottleneck, max-pooling is applied to fit the data into the bottleneck. After the bottleneck, we apply upsampling to allow the data to pass into the next decoder stage.

Chaitanya et al. make heavy use of recurrent connections to improve their results. We tried implemented recurrent connections similar to Chaitanya et al. in our encoder blocks, but saw no significant benefits in performance. It is likely that the missing performance gain was due to improper implementation. However, due to time restrictions, we moved on with the thesis. As such, we do not have recurrent connections as part of our final implementation.

The network has five encoder stages, five decoder stages, and one bottleneck. Chaitanya et al. propose that the channel count for each block-pair be $32, 43, 57, 76$, and 101. These channel counts worked well before incorporating DOF effects. However, when adding support for DOF effects, we provide additional inputs to the network as detailed in Section 4.3.2. It seems appropriate to allow the network a corresponding amount of additional channels to learn the new input. We found that increasing the channel count for each block-pair worked well. The new channel counts for each block-pair are $36, 48, 64, 85$, and 113.

4.3.2 Denoiser Input & Output

The DAE takes input in the form of a deep image. A deep image is a collection of images where the channels have been concatenated, forming a deeper image with more channels. Inspired by Chaitanya et al. we provide the network with albedo, normal, roughness, and depth maps to perform denoising on images without DOF. To try and add support for DOF, we concatenate an additional auxiliary feature on the deep input image.

We want to provide the denoiser with information on whether a specific point on the image is in focus or not. Additionally, since points outside of the acceptably sharp area can vary significantly in their sharpness, the network must be able to determine how out of focus a point is. To achieve this, as explained in Section 3.1.1, we evaluate if the circle of confusion is small enough to evaluate each pixel as acceptably sharp. If it is not, we calculate how out of focus this pixel is relative to other pixels in the frame. We have developed a method for rendering a frame that represents both of these aspects. See Section 4.2.4 for more details on its implementation. See Figure 4.4 for a visualization of how the circle of confusion frame might look in a few different scenarios.

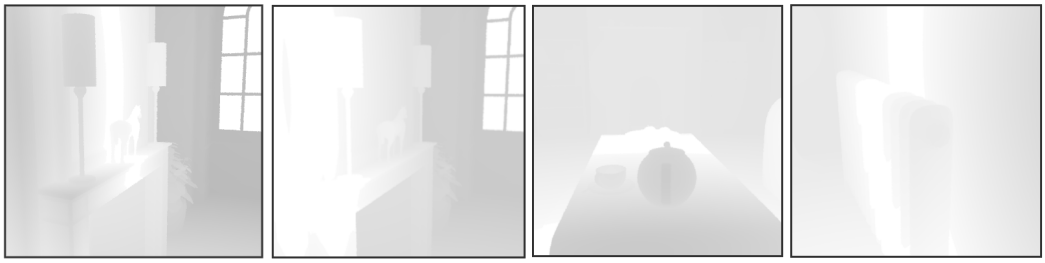


Figure 4.4: Shows the circle of confusion buffer for different times in the scenes fireplace and living room. The fully white area represents the area in the scene where the frame is acceptably sharp. For the rest of the frame, the brighter an area is, the more in focus that area is considered.

In providing the denoiser with an auxiliary frame representing this information, the hope is that it manages to reconstruct areas in and out of focus differently. In areas where the focus is, or moves towards, acceptable, having a clear differentiation between edges in geometry is important. In areas where the focus is not as acceptable, it is most likely more useful to find the average color over a specific area in the scene.

The output from the network is an image with 3 channels representing the illumination at each pixel on the input frame. We follow the procedure described by Chaitanya et al. to avoid having the network reconstruct fully textured scenes. We apply demodulation of the noisy single sample input and the reference output by the albedo map before training the network on that input/output pair. A re-modulation by the albedo map is then made once the network has finished the reconstruction of the illumination to get the final result. Specifically, demodulating refers to dividing the input and reference frames by the albedo map. Re-modulating then refers to multiplying the albedo map back into the frame.

An important part of the work by Chaitanya et al. is that the auxiliary buffers can be rendered using only one sample per pixel. If we start increasing the number of samples while rendering the auxiliary buffers, we effectively increase the number of samples per pixel required as input to the denoiser, which, of course, defeats the purpose of a denoiser which aims is to reconstruct images with low sample counts.

Not being able to take more than 1spp for each auxiliary buffer becomes a problem when we add support for DOF in the path tracer, as the primary ray becomes non-deterministic. Through experimentation, we ended up ignoring the DOF effects for all auxiliary features except albedo. It turns out that we can get the network to recognize the effects of DOF by doing this. For the albedo buffer, we cannot simply ignore the DOF effects, however. Because we perform the demodulation of the noisy input and reference output using the albedo buffer, without DOF effects, the demodulation result becomes incorrect in areas out of focus. Instead, we sample the albedo buffer using DOF effects and perform training at 1spp and 24spp for the albedo buffer. We present the results of these different trainings in Chapter 5 and discuss the side effects and possible improvements to this approach in Section 6.1.

4.3.3 Generating the dataset

While generating the training and validation datasets, we perform several steps of data pre-processing. Before any processing is performed, the data consists of an ordered set of frames from a single scene in size 512 by 512 pixels.

The frames are manually split up into sets of training and validation frames. The split between training vs. validation is approximately $70 - 80\%$ training data and $30 - 20\%$ validation data. We try to separate the two sets such that the validation set is not identical to the training set, but still resembles it, which indeed is not too difficult since most scenes tend to have a common theme throughout.

For each frame in a scene, we let the path tracer render 10 different versions of the noisy single sample input. We can utilize the same output reference and auxiliary buffers across all noisy versions, which allows us to increase the amount of training data tenfold without any significant increase in rendering time.

Chaitanya et al. describe a pre-processing step where they generate sequences of 7 consecutive frames, which seems mostly to be a side-effect of the recurrent nature of their DAE. Even though we do not implement the recurrent connections in our denoiser, we utilize this split into sequences to perform data augmentation in a similar way that Chaitanya et al. do.

Creating the sequences consists of a couple of rather simple steps. First, we chunk the total number of frames into parts of 7 consecutive frames. For each such chunk, we iterate over the frames starting at the first frame. Given a 25% probability, we stop at a specific frame and change the noisy sample input to one of the other nine available versions, keeping the target output and all other auxiliary features the same. Effectively this results in stopping the camera, allowing the network to train on the same frame twice for two different noisy inputs. Second, once a sequence has been finished, we reverse the order of the frames with a random probability of 50%, which results in training data where the camera moves in the reversed direction 50% of the time.

Once the frames in all sequences have been set, before it is ready to be input into the denoiser, we perform additional pre-processing of each frame. For each sequence, we pick a random 64 by 64 crop position across the image and rotation by either 90, 180, or 280 degrees. We apply these crops and rotations to each frame within the sequence. Finally, we perform albedo demodulation of both the noisy input and reference output at this stage as well.

After all pre-processing has been completed, we store the processed frames in a NumPy memmap structure. This allows us to fetch the images back into memory easily during training, allowing us to separate the time spent pre-processing the images from the actual training of the network and increase overall training time as a result.

4.3.4 Loss Calculation

To appropriately penalize all parts of the reconstruction, we construct the loss function from two separate loss functions by a weighted sum. We have a spatial component L_s, penalizing overall loss of quality in the reconstructed image, and a gradient-based loss function L_g penalizing missing edges and features in the reconstructed image. We weight the two components by 0.8 and 0.2 respectively:

$$L = 0.8 * L_s + 0.2 * L_g \tag{4.2}$$

The spatial component calculates the mean of the absolute difference between each pixel in the reconstructed frame:

$$L_s = \frac{1}{N} \sum_{i}^{N} |P_i - T_i| \tag{4.3}$$

where P is the output from the network, T is the target output of the network, and i is each pixel in the frame.

The gradient component is calculated using the Laplacian of Gaussian (LoG). LoG is a combination between the Laplacian and the Gaussian operators. Laplacian is used to detect edges in images but is very sensitive to noise. It is therefore common to apply a Gaussian operator before the Laplacian to lower the amount of noise. We calculate the gradient loss component as follows:

$$L_g = \frac{\sum (LoG(P) - LoG(T))^2}{\sum LoG(T)^2} \tag{4.4}$$

where again P is the output from the network, T is the target output of the network, and each sum is performed over all elements in the resulting image.

4.3.5 Training the Denoiser

We train our network until we notice a significant and consistent increase in validation loss. We train using the Adam optimizer with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.99$. We use a minibatch size of 4 sequences or 28 frames. During each epoch, we compare the performance against previous epochs and continuously store the best loss values. If the current epoch performs better than all previous versions, we save the network weights to disk and continue with the next epoch. For each epoch, we also store the training set loss and validation set loss, regardless of whether they improve or not.

The learning rate is adjusted throughout the training by changing it slightly each epoch. For the first epoch, the learning rate starts at 0.0001. We increase the learning rate tenfold during the first 10 epochs using a geometric progression resulting

in a learning rate of 0.001. After the first 10 epochs the learning rate is slowly decreased every epoch. The exact learning rate modification is shown in equation 4.5, where E_i is the epoch index starting at 0.

$$LR = \begin{cases} 0.0001 * 1.25893^{E_i}, & \text{if } E_i <= 9. \\ \frac{1}{\sqrt{E_i}} 0.0001 * 10, & \text{otherwise.} \end{cases} \tag{4.5}$$

5

Results

In this chapter, we present the results from various training sessions using the model detailed in Chapter 4. We start by explaining how the evaluation and measurements have been set up in Section 5.1. In Section 5.2, we present the results from denoising with a model that has not been extended with DOF support, and as such tries to replicate the results from Chaitanya et al. [5]. In Section 5.3 we present the results of reconstruction having introduced DOF support into the model.

All training sessions, evaluations, and measurements were performed on a machine running Ubuntu 16.04.6 LTS. The machine uses an Intel Xeon E5-2690 v3 processor, has 56GBs of RAM, and uses one of the GPUs on a K80 Tesla graphics card.

5.1 Evaluation Setup

To evaluate the performance of each trained network, we have set up a simple Python script. When evaluating a training session, we load the best available weights for the network based on the measured training loss. Using the loaded network, we perform denoising on the complete collection of frames for that particular scene. For each frame, we measure the time it took to denoise that frame and the structured similarity (SSIM) [37] between the reconstructed and reference frames. When all frames have been denoised, we measure the average processing time and SSIM across all images and report back.

Measuring the time to denoise each frame, we include the actual denoising as well as the re-modulation of the albedo. We do not, however, include the time it takes to load the image from disk, perform any metric calculations such as SSIM, or any other tasks that are irrelevant to the actual denoising process.

Since training sessions are always performed on a single scene, generalization is always measured on different frames within the same scene. We do not measure how a network that was trained on one scene performs while processing frames from another scene. While this type of cross-scene generalization is undoubtedly interesting, this was not the aim of the network developed for this thesis.

Scene	Average Denoising Time	Average SSIM	# Frames
Cornell Box	5.410 ms	0.904	940
Fireplace	5.250 ms	0.795	998
Living Room	5.373 ms	0.792	910
Sponza *(No DOF)*	5.103 ms	0.819	778

Table 5.1: Denoising metrics for each scene trained and evaluated during the thesis. Shows the average time spent denoising per frame, the average SSIM [37] per frame, and the number of frames used to calculate the average values. All measurements were taken using a network trained with albedo 24spp.

5.2 Denoising without DOF

We evaluate the quality of the reconstruction before adding DOF support to set a baseline for comparison. This baseline can then be used when evaluating the network that has been extended to support DOF. Without knowing how our network performs before adding DOF, we cannot evaluate how the changes have affected the reconstruction quality. In this section, we provide results of reconstructing the Sponza scene using a network that has not yet been extended to support DOF.

Looking at the convergence plot in Figure 5.1, we see that the validation loss reaches a stable point fairly quickly. The training loss keeps progressing, indicating that the network lightly overfits towards the training data, which is interesting since most of the frames in Sponza are relatively similar. We want to note here that the validation data is slightly darker overall than the training data, which might be the reasoning behind this behavior. See Figure 5.2, where the first row of images is from the training set, and the second row is from the validation set.

Overall our results are significantly worse than the ones presented by Chaitanya et al. We reconstruct the Sponza scene with an average SSIM of 0.819, while Chaitanya et al. manage to achieve an SSIM of 0.953 for Sponza [5]. In Figure 5.2 we see a significant loss of quality in darker regions. See, for example, the missing shadows behind the red flag, and the area behind the arch in the first row of images. Note, however, that some details are still present. In particular, all arches are well defined, and some details are present in the pillars underneath the arches. In general, we believe that being able to penalize darker regions further would have improved the results quite significantly.

While hard to notice in the frames presented in this chapter, the network produces significant flimmering between consecutive frames in animated sequences. Chaitanya et al. try to resolve this with their utilization of recurrent connections [5].

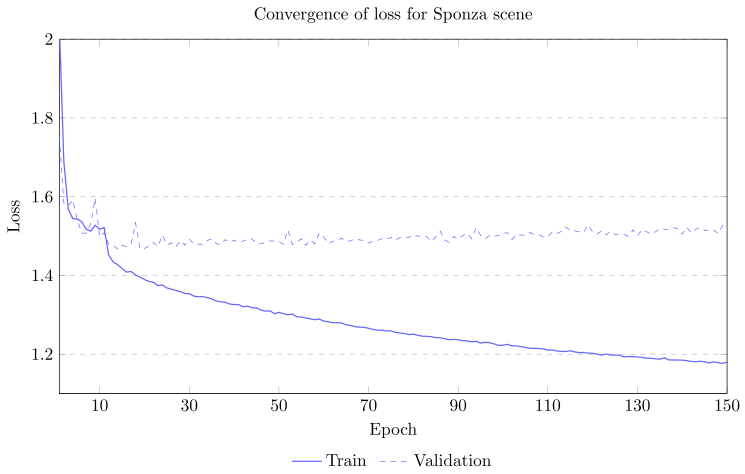


Figure 5.1: The loss during training of the Sponza scene.

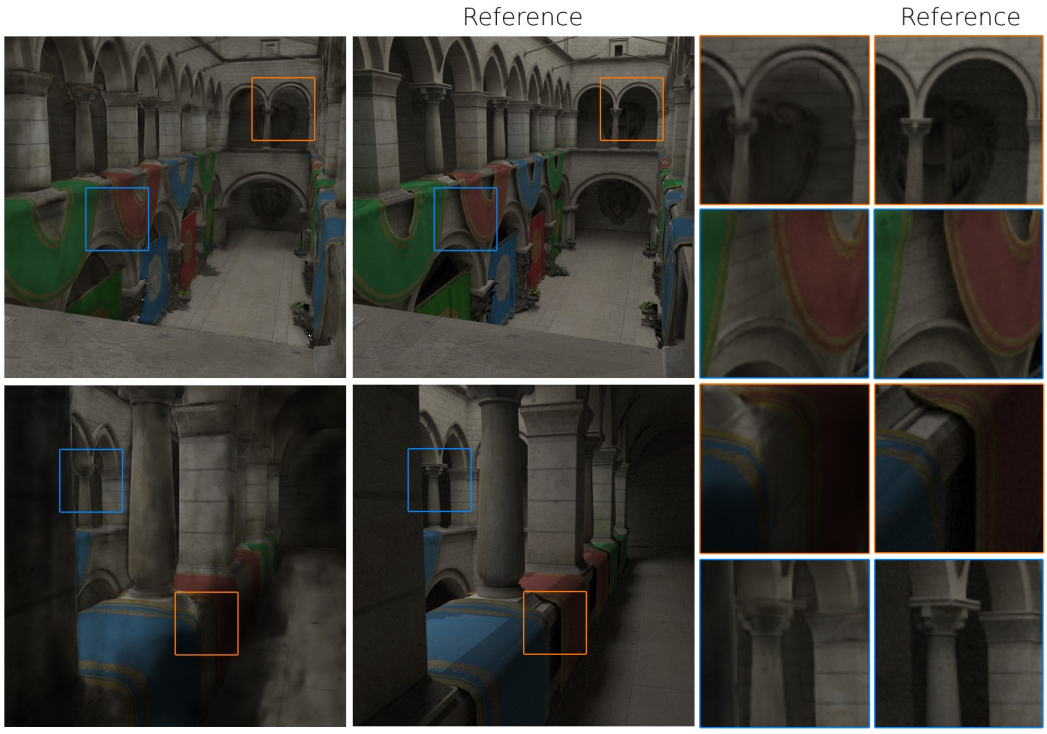


Figure 5.2: Denoised frames of the Sponza scene. One column represents the reconstructed frames and the other one the rendered reference frames. The zoomed in crops showcase specific areas where we see significant details reconstructed or shadows not reconstructed properly.

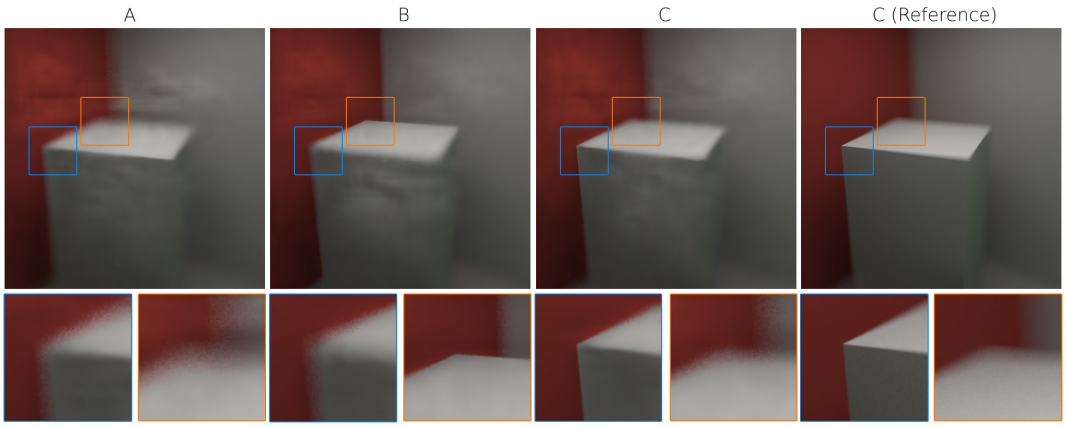


Figure 5.3: Denoised frames from an animated sequence in Cornell Box. The animation only changes the cameras aperture and focal distance. The crops show specific areas where we see significant change in focus. The last image is the reference image of frame C. All images were denoised using the albedo 24spp network.

5.3 Denoising with DOF

The goal of adding DOF support in the denoiser is that it learns to distinguish between points in and out of focus, and handles them appropriately. In practice, this means points in focus keep the same amount of sharpness that they had before incorporating DOF in the model, and points out of focus are blurred appropriately to match the reference images.

For each scene, two sets of training sessions and evaluations have been run, one with an auxiliary albedo frame sampled at 24spp, and one with the albedo frame sampled at 1spp. It is also worth mentioning that we have run experiments with all auxiliary frames being non-rasterized. According to our tests, this does not seem to improve convergence significantly. As such, we do not report any results on different combinations of auxiliary frames being rendered as rasterized vs. non-rasterized.

Looking at figures 5.3, 5.4, and 5.5 we can see a clear difference between points in and out of focus, indicating that the network is capable of distinguishing between such points. While not shown in any figure, we would also like to note that in frames where the aperture is 0, we can see the network trying to denoise the whole frame as if it was in focus — indicating that the circle of confusion auxiliary frame impacts the denoising results of the network.

We manage to get quite good results for simple scenes. In particular, for the Cornell Box scene, we reconstruct images with an SSIM of 0.904. When we increase the complexity of the scene, the SSIM falls significantly. We achieve an SSIM of 0.795 for Fireplace and 0.792 for Living Room. This reflects clearly in the images presented in figures 5.3, 5.4, and 5.5. The reconstructed images in 5.3 are significantly better

than those in 5.4, and 5.5. We also note that even for simple scenes such as Cornell Box, we see significant artifacts in large flat areas. See Figure 5.3 behind the cube for a reference point of such a large flat surface. These areas are also the ones that generate most of the significant flimmering mentioned in Section 5.2.

For more complex scenes, we see an overall decrease in quality. Specifically, we note that a significant amount of details disappear, even for points in focus. See Figure 5.4 image C, where details on both the fireplace and the horse on top of the fireplace are almost entirely gone. While there is still a difference between points in and out of focus, the network seems to overshoot quite significantly, and thus points in focus still significantly lack well-defined edges and details. We see similar behavior in the Living Room scene.

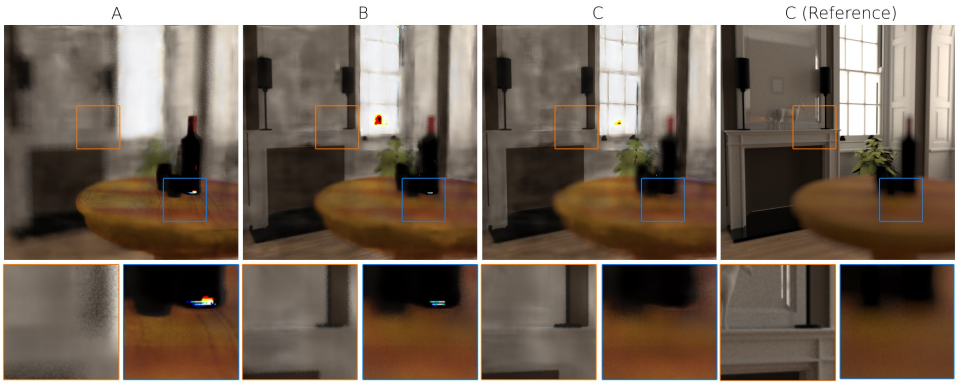


Figure 5.4: Denoised frames from an animated sequence in Fireplace. The animation only changes the cameras aperture and focal distance. The crops show specific areas where we see significant change in focus. The last image is the reference image of frame C. All images were denoised using the albedo 24spp network.

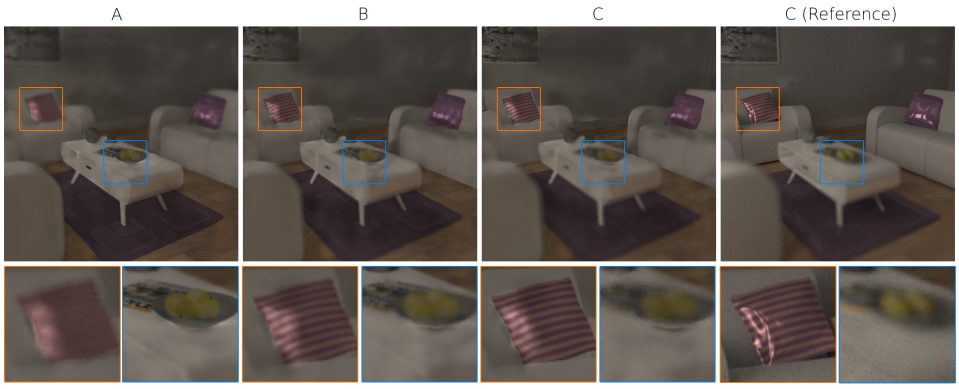


Figure 5.5: Denoised frames from an animated sequence in Living Room. The animation only changes the cameras aperture and focal distance. The crops show specific areas where we see significant change in focus. The last image is the reference image of frame C. All images were denoised using the albedo 24spp network.

Figure 5.6 shows that by utilizing the albedo map sampled at 24spp, we improve training loss significantly. Additionally, we note that the number of samples in the albedo impacts the generalization of the network onto the validation set. Surprisingly, it is the network trained with albedo sampled at 1spp that performs better on the validation set relative the training set. For Living Room, the validation loss at 1spp albedo even surpasses the 24spp albedo version around epoch 100, again indicating that the 24spp albedo version is easier to overtrain.

For the Cornell Box scene, we observe a different behavior. Figure 5.7 shows, contradictory to Figure 5.6, that the network trained with 1spp albedo performs slightly better than the network trained with 24spp albedo. Given that the only difference is the number of samples in the albedo map, we can venture as far as to guess that this behavior depends on the complexity of the albedo map. Figure 5.8 shows a comparison between the complexity of the albedo map in all three different scenes.

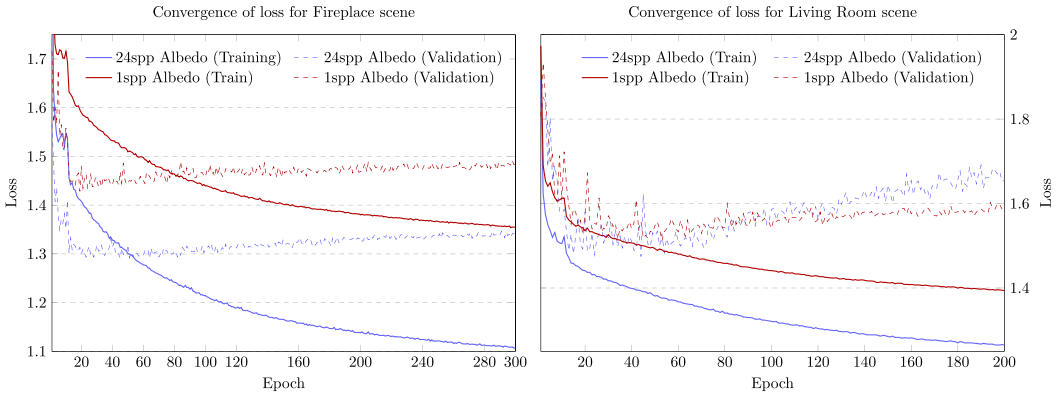


Figure 5.6: The loss during training of the Fireplace and Living Room scenes.

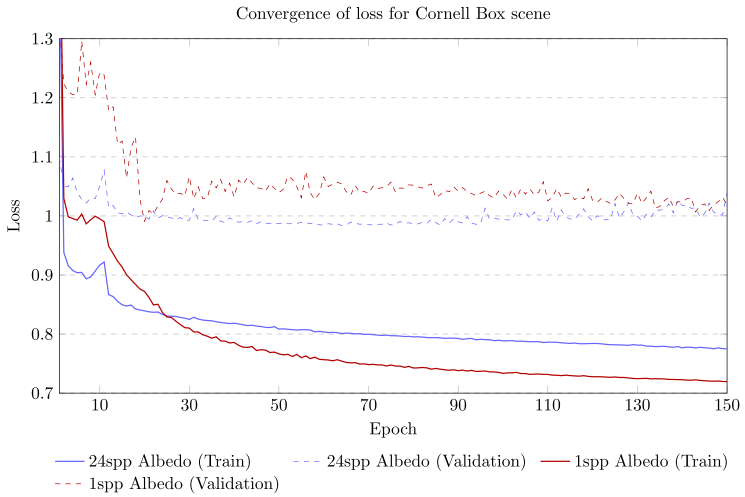


Figure 5.7: The loss during training of the Cornell Box scene.

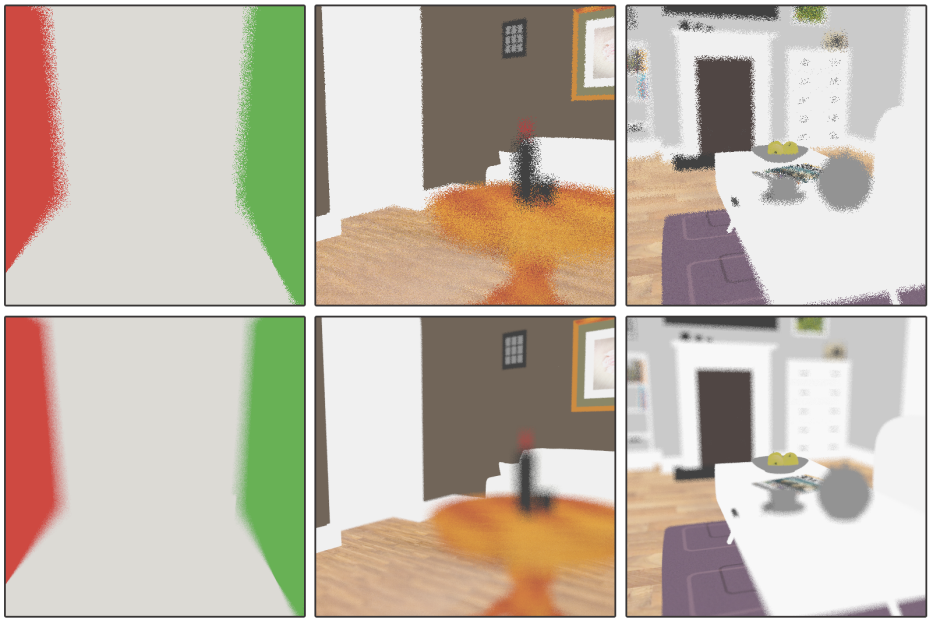


Figure 5.8: The different Albedo auxiliary buffers with both 1spp and 24spp. The top row shows the 1spp frame and the bottom row the 24spp.

It is also interesting to look at the difference in denoised results between using the albedo 24spp and 1spp frames. We show both variants in Figure 5.9. We see that the amount of noise in the final reconstructed image depends heavily on the amount of noise in the albedo map, in particular in areas of the frame that are out of focus. This behavior is to be expected. The 1spp albedo map contains significantly more noise in out of focus areas, as seen in Figure 5.8. As such, unless the network learns to counteract the noise in the albedo map, re-modulating the reconstructed result by the albedo carries along that noise to the final result.

In some frames, we see significant artifacts that impact the overall quality of the reconstruction results heavily. See Figure 5.4 image A and B for examples. These extreme values carried along from the output of the network, and do not seem to be a side-effect of the albedo modulation process. The reference image modulated by the albedo map does not have these artifacts, so they are indeed an incorrect reconstruction from the network. It is unclear what exactly is causing this.

Finally, the denoiser can process a single frame of size 512x512 in about 5ms, achieving an approximately 200fps at this resolution, which is certainly in the realm of real-time denoising. We do not measure the performance of the path tracer since the goal was not to build an interactive path tracer.

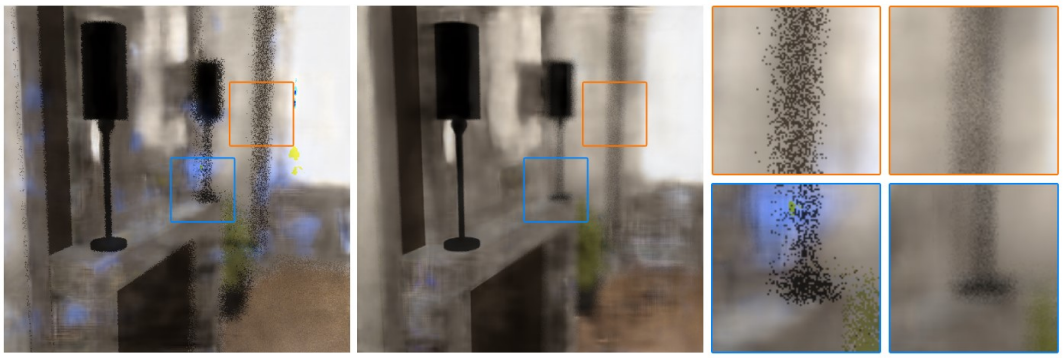


Figure 5.9: The difference between the result of denoising images with a network that has been trained with 1spp and 24spp albedo. The left, more noisy image, is the 1spp albedo reconstruction and the right image is the 24spp albedo reconstruction.

6

Conclusion

Throughout this report, we have covered many topics. We started by explaining the theory behind the core concepts used during the thesis in Chapter 3. We then moved onto Chapter 4, where we explained the methodologies used during the development of a path tracer, and the extension of a DAE to support DOF. In Chapter 5 we presented the results of our extended DAE and showed how it performed while denoising frames in several different scenes. In this chapter, we discuss the results presented in Chapter 4 and close off with some conclusions on the final results of the thesis.

6.1 Discussion

The reconstruction results before adding DOF support, presented in Section 5.2, are overall worse than the reconstruction presented by Chaitanya et al. [5]. There are many potential sources of this. First, we are missing the implementation of the recurrent connections detailed by Chaitanya et al. As a side effect, we are missing much information collected and stored in these recurrent connections. Chaitanya et al. mention that these recurrent connections store information across multiple frames and combine these to generate higher sample count images [5]. Second, our training material might not be as optimized as the ones used in the original paper. Third, there might be parts of the implementation that are missing or incorrect because of missing, or misunderstood, implementation details. Nevertheless, we can see that our network does interpret the results correctly, and produces acceptable results, even if we cannot get quite all the way.

While introducing the recurrent connections from the original paper could potentially improve the results significantly, both with and without DOF, it might introduce some ghosting artifacts into the DOF reconstruction as well. It would then be interesting to evaluate how the method developed by Schied et al. [30] could be used to try and reduce ghosting artifacts in areas that are out of focus.

We note that our execution time for denoising a single frame is significantly faster

than that of Chaitanya et al., who report a processing time of 54.9 ms for a single frame at 1280x720 [5]. We denoise a single frame at 512x512 in approximately 5 ms. Assuming execution time scales linearly with the number of pixels, as reported by Chaitanya et al. [5], we would denoise a single frame at 1280x720 in around 15 ms. Still significantly beating the original implementation. This is another indicator that we might be missing a few parts of the network, and thus, we save time not having to perform those missing operations. It is likely that a significant amount of the saved execution time is due to the missing recurrent connections in our implementation.

We see a significant loss of quality in darker regions of our reconstruction results. This might primarily be due to not having implemented the gamma correction prior to loss calculation, which Chaitanya et al. mention specifically helps with penalizing darker regions further. We did not end up implementing this gamma correction, as our network outputs negative values initially during training, and gamma correction cannot be performed on negative values when $0 < \gamma < 1$. This gamma correction missing might have played a big roll in dark regions not getting penalized enough during training. We tried including areas of Sponza that are extremely dark as part of the training data, this, however, only resulted in black splotchy circles in the reconstruction result, and no performance gain in dark regions.

When we introduce DOF support into the network, we see a fall in performance, in particular in focused areas of the frame. We get significant over-blurring for points in focus and thus lose valuable detail in these areas. There are multiple approaches we suggest taking to fix this. It would be interesting to add another term to the loss function that penalizes areas in focus harder for missing edges and detail. The circle of confusion auxiliary buffer could potentially be used to determine which areas to penalize more. Additionally, we do not, in either of our scenes, have a particularly large set of frames that has an aperture of 0, i.e., most of our frames include points that are out of focus. It could potentially be largely beneficial for convergence to allow a more significant part of the training set to be frames were no areas are out of focus.

The average SSIM stays almost the same as we extend the network with DOF. Although this sounds good in theory, we believe this result is slightly misleading. Manually comparing the reconstruction results, we see significantly better results before adding DOF support. Which, again, is mostly due to a significant amount of detail missing in focused areas after adding DOF support. It is interesting then that the SSIM does not change significantly. We believe the reason for this is that even if edges and details disappear, it is easier for the network to reconstruct blurry areas, and so since large parts of many frames are blurry, we get a similar SSIM score.

As mentioned in Section 5.3, our final result in out of focus areas depends heavily on the amount of noise in the albedo buffer. If our current network were to counteract this, it would have to learn to produce noise that in turn counteracts the noise introduced when re-modulating by the noisy albedo. Given that the network

struggles with reconstructing fine details in geometry, it seems unlikely the network learns to reconstruct this type of exact noise. We instead propose the introduction of another pre-processing step. We suggest denoising the albedo map separately, before demodulating the input and reference using the map. The amount of noise in a specific area of the albedo frame only depends on how out of focus that area is. For areas that are considered acceptably sharp, the albedo map contains no noise, and for areas that are heavily out of focus, the albedo map contains significantly more noise. See Figure 5.8 for a visual representation of this relationship. While out of focus areas contain noise, because of the simple, gradient-like, nature of the out of focus areas in the albedo map, a less noisy version generally seem quite simple to estimate from that noise. Even for a single sample per pixel albedo map, we predict that a fairly straight forward average across an area of pixels that depends on the size of the circle of confusion should work fairly well in most scenarios. Alternatively, provided one applies a rather fast filter, it would be interesting to consider how the approach presented by Kalantari and Sen [13] could be utilized to denoise the albedo map.

6.2 Conclusion

In conclusion, we have managed to extend a DAE developed by Chaitanya et al. [5], capable of denoising Monte Carlo sampled images with low sample counts, to distinguish between points in and out of focus. We have developed a new auxiliary buffer that provides the network with information on how out of focus each point in a frame should be. We used rasterized versions of all auxiliary buffers except for the albedo, which is sampled through a thin lens, similar to ordinary illumination. Because of its stochastic nature, and the pre-, and post-processing steps performed using the albedo map, the inherent noise in this map limits the amount of noise in the final reconstructed image.

We did not manage to reproduce the results presented by Chaitanya et al. in their entirety. Our network, even without DOF support, was missing quite a bit of quality in the reconstruction. Additionally, our extensions to the network cause significant loss of detail in the reconstructed results, even for points in focus. To avoid this loss of quality in details, we propose a change to the loss function as a first next step to correct for this.

We manage to perform denoising in around 5 ms per frame. However, the overall low quality of details in the denoised frames, and the inherent noise in the albedo map indicate that both additional and more complex denoising steps might be a requirement to apply the denoiser in practice. Most likely increasing this time significantly.

6.3 Future Work

We see multiple avenues for possible extension and improvements upon the work we have presented in this report. We believe that the most relevant work lies in trying to avoid losing important details in the reconstructed results. Thus, as mentioned in Section 6.1, we believe that adding an additional term to the loss function penalizing the loss of detail in focused areas might be a good way forward. Additionally, to further increase quality and applicability of the work, adding a separate denoiser for the albedo map as an additional pre-processing step is something we would like to see tested.

It should also be highly relevant to incorporate the recurrent connections implemented by Chaitanya et al. [5]. In theory, recurrent connections should significantly improve the results of the network without DOF support, and it would be interesting to evaluate how adding those recurrent connections would affect the changes we have made to the network.

It would be interesting to evaluate how the size of the frame that is being reconstructed impacts the performance of the DOF reconstruction. In theory, the acceptable circle of confusion depends on the area within a scene covered by a single pixel. The area covered by a single pixel changes as the number of pixels in the frame changes, thus the acceptable circle of confusion should change too. If one finds that the performance of the DOF reconstruction changes significantly depending on frame size, an additional extension would perhaps be to avoid having to specify the acceptable circle of confusion while calculating the auxiliary buffer.

Another extension, which is less directly relatable, is extending the network to support additional stochastic primary ray effects. We focused on DOF in this thesis, we would, however, like to see motion blur implemented as an additional extension.

Bibliography

[1] Jinwon An and Sungzoon Cho. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE*, 2(1), 2015.

[2] Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony DeRose, and Fabrice Rousselle. Kernel-predicting convolutional networks for denoising monte carlo renderings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2017)*, 36(4):97:1–97:14, 2017.

[3] Pablo Bauszat, Martin Eisemann, S John, and M Magnor. Sample-based manifold filtering for interactive global illumination and depth of field. In *Computer Graphics Forum*, volume 34, pages 265–276. Wiley Online Library, 2015.

[4] Laurent Belcour, Cyril Soler, Kartic Subr, Nicolas Holzschuch, and Fredo Durand. 5d covariance tracing for efficient defocus and motion blur. *ACM Transactions on Graphics (TOG)*, 32(3):31, 2013.

[5] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):98, 2017.

[6] Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, et al. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics (TOG)*, 37(3):30, 2018.

[7] Robert L Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *ACM SIGGRAPH computer graphics*, volume 18, pages 137–145. ACM, 1984.

[8] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik Lensch. Edge-avoiding à-trous wavelet transform for fast global illumination filtering. In *Proceedings of the Conference on High Performance Graphics*, pages 67–75. Eurographics Association, 2010.

[9] Eduardo SL Gastal and Manuel M Oliveira. Adaptive manifolds for real-time

high-dimensional filtering. *ACM Transactions on Graphics (TOG)*, 31(4):33, 2012.

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

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] James T Kajiya. The rendering equation. In *ACM SIGGRAPH computer graphics*, volume 20, pages 143–150. ACM, 1986.

[13] Nima Khademi Kalantari and Pradeep Sen. Removing the noise in monte carlo rendering with general image denoising algorithms. In *Computer Graphics Forum*, volume 32, pages 93–102. Wiley Online Library, 2013.

[14] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[15] Tzu-Mao Li, Yu-Ting Wu, and Yung-Yu Chuang. Sure-based optimization for adaptive sampling and reconstruction. *ACM Transactions on Graphics (TOG)*, 31(6):194, 2012.

[16] Edward Liu, Ignacio Llamas, Patrick Kelly, et al. Cinematic rendering in ue4 with real-time ray tracing and denoising. In *Ray Tracing Gems*, pages 289–319. Springer, 2019.

[17] Chris Lomont. Fast inverse square root. *Tech-315 nical Report*, 32, 2003.

[18] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[19] Xiao-Jiao Mao, Chunhua Shen, and Yu-Bin Yang. Image restoration using convolutional auto-encoders with symmetric skip connections. *arXiv preprint arXiv:1606.08921*, 2016.

[20] Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. An efficient denoising algorithm for global illumination. *Proceedings of High Performace Graphics*, 6, 2017.

[21] Morgan McGuire. Computer graphics archive, July 2017. `https://casual-effects.com/data`.

[22] Soham Uday Mehta, Brandon Wang, and Ravi Ramamoorthi. Axis-aligned filter-

ing for interactive sampled soft shadows. *ACM Transactions on Graphics (TOG)*, 31(6):163, 2012.

[23] Soham Uday Mehta, Brandon Wang, Ravi Ramamoorthi, and Fredo Durand. Axis-aligned filtering for interactive physically-based diffuse indirect lighting. *ACM Transactions on Graphics (TOG)*, 32(4):96, 2013.

[24] Soham Uday Mehta, JiaXian Yao, Ravi Ramamoorthi, and Fredo Durand. Factored axis-aligned filtering for rendering multiple distribution effects. *ACM Transactions on Graphics (TOG)*, 33(4):57, 2014.

[25] Jacob Munkberg, Karthik Vaidyanathan, Jon Hasselgren, Petrik Clarberg, and Tomas Akenine-Möller. Layered reconstruction for defocus and motion blur. In *Computer Graphics Forum*, volume 33, pages 81–92. Wiley Online Library, 2014.

[26] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[27] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[28] Fabrice Rousselle, Marco Manzi, and Matthias Zwicker. Robust denoising using feature and color information. In *Computer Graphics Forum*, volume 32, pages 121–130. Wiley Online Library, 2013.

[29] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, page 2. ACM, 2017.

[30] Christoph Schied, Christoph Peters, and Carsten Dachsbacher. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):24, 2018.

[31] Peter Shirley, Timo Aila, Jonathan Cohen, Eric Enderton, Samuli Laine, David Luebke, and Morgan McGuire. A local image reconstruction algorithm for stochastic rendering. In *Symposium on Interactive 3D Graphics and Games*, pages PAGE–5. ACM, 2011.

[32] Charles M Stein. Estimation of the mean of a multivariate normal distribution. *The annals of Statistics*, pages 1135–1151, 1981.

[33] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.

[34] Thijs Vogels, Fabrice Rousselle, Brian McWilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. Denoising with kernel prediction and

asymmetric loss functions. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2018)*, 37(4):124:1–124:15, 2018.

[35] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4):143, 2014.

[36] Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 195–206. Eurographics Association, 2007.

[37] Zhou Wang, Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[38] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and S-E Yoon. Recent advances in adaptive sampling and reconstruction for monte carlo rendering. In *Computer Graphics Forum*, volume 34, pages 667–681. Wiley Online Library, 2015.