Image Filtering using Halide and a new Denoising Algorithm for Gradient-Domain Rendering

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Delio Vicini

2015

Leiter der Arbeit:
Prof Dr. Matthias Zwicker
Institut für Informatik und angewandte Mathematik

Abstract

In this thesis, we implemented two state-of-the-art denoising algorithms using Halide [RKAP+12]. Halide is a domain-specific programming language for high performance image processing. We implemented both "Robust Denoising using Feature and Color information" [RMZ13] and "Dual-Domain Image Denoising" [KZ15] using Halide. Our implementations on the GPU are faster than the preexisting code provided by the respective authors. Furthermore, we present a new denoising algorithm for gradient-domain rendering [LKL+13]. Gradient-domain rendering algorithms compute not only a Monte Carlo estimate of the image itself, but also of its finite difference gradients. The final image is then reconstructed by solving a screened Poisson equation. Our denoising algorithm extends the Poisson problem by adding regularization constraints based on local feature patches. We also present an efficient implementation of our algorithm using CUDA and compare it to the existing biased L1-reconstruction for gradient-domain rendering, which we outperform by a significant factor.

Contents

Chapter 1

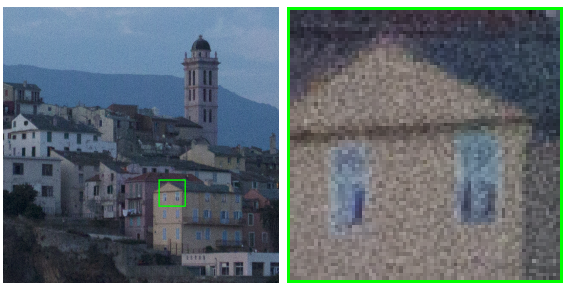
Introduction

1.1 Motivation

One major area of research in computer graphics and vision is the reconstruction of data corrupted by noise. Noise can occur in many situations. Measurements of a natural quantity can be corrupted by noise, but also results of numerical simulations. In this thesis, we focus on denoising of images as an important special case. Many techniques applicable to denoising of images can also be used on other types of data.

Noisy images can occur in a variety of different contexts. The images produced by digital cameras in low-light situations typically suffer from noise. The noise is produced by the image sensor of the camera, which cannot reliably measure the colors of the scene if there is not enough light available. An example of this is given in Figure 1.1. Denoising algorithms can help to improve the quality of images tremendously. These algorithms can be applied after the image has been captured, without adding additional cost to the production of digital cameras.

Another application of denoising algorithms is in realistic rendering. Producing realistic images of artificial scenes has a wide range of applications, including movies, games and visualizations. Computing realistic images is hard, since one has to simulate how light is transported in a scene. The light in a scene can be



(a) Noisy image (b) Details

Figure 1.1: Example of noise produced by a digital camera in a low-light situation.

(a) Noisy image (b) Details

Figure 1.2: Example of noise produced by a Monte Carlo rendering algorithm.

reflected or refracted multiple times before it reaches the sensor of the virtual camera. From a mathematical point of view, computing the light transport in a scene amounts to solving an integral equation. The image produced by the light in a scene can unfortunately not be computed in closed form.

Rendering algorithms can in general be divided into two categories: *Biased* algorithms reduce the rendering time at the cost of accuracy. The resulting images can look very realistic, even though they are not the result of a physically correct light transport simulation. These algorithms typically require a skilled artist to tweak many parameters to get convincing results. Nevertheless, they are used very widely, as they can give good enough results in a short amount of time. Another approach is taken by *unbiased* algorithms, which try to compute the image by fully simulating the light transport in a scene. This has the advantage, that parameter tweaking becomes less important and realistic scenes can be created in less time. These algorithms however rely on Monte Carlo methods and can take a very long time to converge to the true solution. Given a limited computation time, the images produced by these algorithms can be very noisy, as demonstrated in Figure 1.2.

Due to increasing computation power, there has been a steady shift towards more physically correct rendering in many areas of computer graphics applications.[ZJL+15] For offline rendering, it has become fairly popular, to first compute a potentially noisy image using Monte Carlo methods and then remove the noise in a post-processing step. This adds some bias to the rendered results, but increases the overall image quality.

The filtering step is typically very fast compared to the rendering process itself. This is important, since the filtering step should in the end accelerate the whole rendering process. High performance is usually achieved by implementing the filter to run in parallel on a *graphics processing unit* (GPU). All algorithms presented in this thesis have thus been implemented to run on the GPU.

1.2 Goals and contributions

In the first part of this thesis, we implemented two different denoising algorithms using the Halide [RKAP+12] programming language. Halide is a domain-specific language, which allows the programmer to write very efficient image processing code. Because the language is still fairly new, the amount of publicly available

source code is limited. Our goal was to evaluate how useful Halide is to implement research level image filtering algorithms. The usefulness on one hand depends on the achievable performance, but on the other hand also on the ease of use of the language itself from a programmers point of view. One of Halide's goals is to allow the code both to be readable and optimized at the same time.

We first implemented the "Robust Denoising using Feature and Color information" (RDFC) [RMZ13] algorithm. This filter is used to denoise images created by Monte Carlo rendering techniques. We also implemented the "Dual-Domain Image Denoising" (DDID) [KZ15] algorithm. This second algorithm is more general and is mostly suitable for denoising digital photographs.

In the second part of this thesis we present a new denoising algorithm for gradient-domain rendering [LKL+13, MRK+14, MKA+15, KMA+15]. Gradient-domain rendering algorithms have been introduced recently as a promising way to speed up Monte Carlo rendering. These algorithms work by computing, additionally to the conventional rendering process, image space gradients of the output image. The final image is then computed by solving a *screened Poisson equation*. We originally intended to implement the Poisson reconstruction using Halide, but then quickly discovered, that Halide does not provide all the necessary operations to do so. Our denoising algorithm works by extending the Poisson reconstruction by additional regularization constraints based on local feature patches. In rendering, the term "feature" refers to additional per-pixel information about the scene accumulated during the rendering processes. In short, our contributions are

- Fast implementations of RDFC and DDID using Halide. Our code outperforms the existing implementations of these algorithms. We provide both GPU and CPU schedules for our Halide code.
- A novel denoising algorithm for gradient-domain rendering, which improves over the previous reconstructions in terms of empirical mean squared error to the reference image. A fast implementation of the algorithm using CUDA is also presented.

1.3 Thesis structure

In the second chapter of this thesis, a small overview over the required background is given. This includes a few mathematical definitions, a short section on Monte Carlo rendering and an introduction to Halide. We assume the GPU programming model to be familiar to the reader. In the third chapter, the RDFC filter and its implementation using Halide are discussed. The fourth chapter does the same thing for the DDID filter. The fifth chapter gives a short overview of gradient-domain rendering and presents our new denoising algorithm. We also discuss the efficient implementation using CUDA and compare our algorithm to the existing reconstructions for gradient-domain rendering.

Chapter 2

Background

2.1 Mathematics

This section will introduce some basic mathematical concepts and definitions, that will be used in the rest of this thesis. A knowledge of fundamental probability theory, linear algebra and analysis is assumed.

2.1.1 Statistical tools

Denoising of images is always estimation of unknown values from some random input. The quality of an estimator $\hat{X}$ of an unknown variable X can be measured by the *mean squared error* (MSE)

$$\mathrm{MSE}[\hat{X}] = \mathrm{E}[(\hat{X} - X)^2] \tag{2.1}$$

where $\mathrm{E}[\dots]$ denotes the expected value. Another often used measure is the *bias* of an estimator, given as:

$$\mathrm{Bias}[\hat{X}] = \mathrm{E}[\hat{X} - X] \tag{2.2}$$

An estimator is called *unbiased* if $\mathrm{Bias}[\hat{X}] = 0$ and *biased* if $\mathrm{Bias}[\hat{X}] \neq 0$. By expanding the definition of the MSE, we get the equality

$$\mathrm{MSE}[\hat{X}] = \mathrm{Bias}[\hat{X}]^2 + \mathrm{Var}[\hat{X}] \tag{2.3}$$

In denoising, the assumption is usually, that the noisy pixel value is an unbiased estimate for the true pixel value. The goal of denoising algorithms is to compute a new estimate with a lower MSE than the original estimate. The new estimate should have a lower variance, at the cost of introducing some bias. The difficulty is thus to compute an estimate, which balances bias and variance to get a low MSE value. This is also called *bias-variance trade-off*.

In image denoising, the MSE of a filter can usually not analytically be determined. An estimate for the true per-pixel MSE is the squared distance from the denoised pixel to the reference pixel from a ground truth image. The per-pixel MSE can then be averaged over the whole image in order to get one single number, which quantifies the quality of a filter.

In rendering, we usually need to filter high dynamic range images, where individual pixels can have values larger than 1. The standard empirical MSE

is in this context often not optimal to measure the quality of a filter, since the total error can be dominated by the error at a few bright pixels. This can for example happen, if a filter accidentally removes a small, but very bright specular highlight. A useful alternative error measure is the *relative mean squared error* (RMSE) [RKZ12], which divides the per-pixel error by the magnitude of the reference image. The per-pixel RMSE is defined in channel c at pixel p as

$$\mathrm{RMSE}_c(p) = \frac{(I_c(p) - \hat{I}_c(p))^2}{((I_1(p) + I_2(p) + I_3(p))/3)^2 + 10^{-3}} \ , \tag{2.4}$$

where I is the reference image and $\hat{I}$ is the denoised image. Similar to the MSE, the RMSE is averaged over the whole image. In this thesis, all measurements of the quality of a filter are made using the RMSE.

2.1.2 Discrete Fourier transform

In many signal processing applications, including denoising, it is of interest, to look at a signal from a frequency point of view. Understanding how a signal is composed of low and high frequencies can be very useful for denoising. The dual-domain image denoising algorithm relies on frequency domain noise estimation. This section will therefore give a short overview of the *discrete Fourier transform* (DFT) and define the terminology used later on.

It is a well known fact, that many periodic functions can be represented as a series of sine and cosine functions. This is generalized to non-periodic functions by the continuous Fourier transform. For an integrable function $f : \mathbb{R} \to \mathbb{C}$, its Fourier transform $\hat{f} : \mathbb{R} \to \mathbb{C}$ is defined as

$$\hat{f}(t) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i t x} dx \ . \tag{2.5}$$

The Fourier transform maps a function from the *spatial domain* to the *frequency domain*. In the frequency domain, the value $\hat{f}(t)$ is the contribution of frequency t to the function f. The Fourier transform is invertible under certain conditions on the underlying function space. We will not go into the details of these conditions, as they do not matter for our applications.

It turns out, that discretizing a function and then transforming it to the frequency domain is equivalent to first transforming to frequency domain and then discretizing. The discrete frequency domain coefficients of a function can be computed from the discretized function and do not require the continuous function to be known. For a discrete signal $x = (x_0, \ldots, x_{N-1})$, the discrete Fourier transform is defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N} \ , \tag{2.6}$$

where $k \in \{0, \ldots, N-1\}$. The DFT is invertible and the inverse transform is given as

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{2\pi i k n / N} \ . \tag{2.7}$$

The DFT is a change of basis from the standard basis to the Fourier basis. The transformation given above can be written as a matrix-vector multiplication. Implemented naively, the DFT has a complexity of $O(N^2)$. The complexity can be reduced by employing a *fast Fourier transform* (FFT) algorithm. The idea of FFT algorithms is to compute the DFT of the input by computing many small DFTs. The FFT can reduce the complexity to $O(N \log N)$.

For our application, we need the Fourier coefficients of a 2D image. The DFT of a 2D signal of size $N \times M$ is defined as

$$X_{kl} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x_{nm} e^{-2\pi i(kn/N + lm/M)} \ . \tag{2.8}$$

These coefficients can be computed by first applying the DFT along one dimension and then transforming this result again along the second dimension.

The Fourier transform is also often used to efficiently evaluate convolutions of two sequences. The *circular convolution* of two finite sequences $x = (x_0, \ldots, x_{N-1})$ and $y = (y_0, \ldots, y_{N-1})$ is defined as

$$(x * y)_n = \sum_{l=0}^{N-1} x_l \cdot y_{n-l \mod N} \ . \tag{2.9}$$

The *convolution theorem* for the discrete Fourier transform states that

$$\mathrm{DFT}[x * y] = \mathrm{DFT}[x] \circ \mathrm{DFT}[y] \ , \tag{2.10}$$

where $\circ$ denotes the pointwise multiplication of two sequences. The frequency domain coefficients of the convolution of two sequences in the spatial domain are exactly the pointwise product of the frequency domain coefficients of the individual sequences. The convolution can therefore be evaluated by first transforming both sequences to the frequency domain, multiplying their coefficients and then applying the inverse DFT to get back to the spatial domain. If the wrap around effect of the circular convolution is undesirable, the input signal has to be zero-padded before transforming it. The result then needs to be cropped back to the original size.

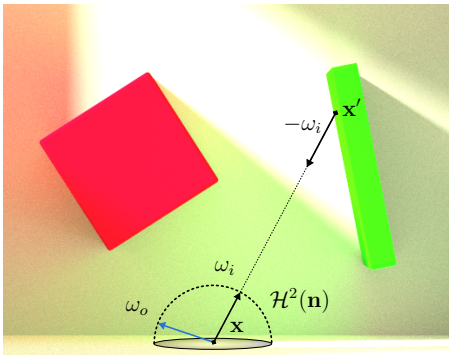


Figure 2.1: Visualization of some of the terms used in Equation (2.11). The radiance emitted in direction ω_o depends on the radiance coming from the environment. In this case, the point $\mathbf{x}$ is lit indirectly by other objects, most notably by the green box on the right reflecting the incoming light sunlight.

2.2 Monte Carlo rendering

In computer graphics, rendering realistic computer generated images is one of the main research areas. To create a realistic image computationally, one has to simulate how light is transported through a scene. This section gives a very brief introduction to Monte Carlo rendering. Light transport depends on the geometry of the objects in a scene and their material attributes. The connection of these properties to light transport is given by the *rendering equation* [Kaj86]:

$$L(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\mathcal{H}^2(\mathbf{n})} f(\mathbf{x}, \omega_o, \omega_i) L(\mathbf{x}'_i, -\omega_i) \cos\theta_i d\omega_i \ , \tag{2.11}$$

where $L(\mathbf{x}, \omega_o)$ denotes the *radiance*, i.e. the amount of light energy, transported along a ray leaving point $\mathbf{x}$ in direction ω_o. This radiance is composed of the radiance $L_e(\mathbf{x}, \omega_o)$ emitted by the object itself and radiance reflected from the environment. The reflected radiance is computed by integrating the incoming radiance over the hemisphere $\mathcal{H}^2(\mathbf{n})$, where $\mathbf{n}$ is the surface normal. The incoming radiance $L(\mathbf{x}'_i, -\omega_i)$ is weighted by the *bidirectional reflectance distribution function* (BRDF) value $f(\mathbf{x}, \omega_o, \omega_i)$, which describes how light is reflected by a certain material. It is also multiplied by the cosine of the angle θ_i between the surface normal and ω_i. The cosine factor accounts for the fact, that the incoming light strength depends on the angle of the ray hitting the surface. A visualization of the terms in Equation (2.11) is given in Figure 2.1. The rendering equation in this form does not account for effects such as refractions or subsurface scattering. To generalize, one would need to integrate over the whole sphere of directions instead of the hemisphere and use the *bidirectional scattering distribution function* (BSDF) instead of the BRDF. The BSDF also describes the refraction and scattering properties of a material.

The recursive nature of the rendering equation and the irregular structure of geometries and materials in a typical scene make it impossible to find a closed form solution. A numerical solution to the integral equation can be computed using Monte Carlo methods. From the rendering equation, it is possible to derive a *path integral formulation* of the value in each pixel of an image recorded by a

camera in the scene. [Vea98] The value of pixel j is given as

$$I_j = \int_\Omega f_j(\bar{x}) d\mu(\bar{x}) \tag{2.12}$$

where Ω is the space of all possible light paths in the scene. A path is defined as the connection of an ordered list of vertices on surfaces in the scene. $f_j(\bar{x})$ is the *measurement contribution function* and $d\mu(\bar{x})$ is the *area-product measure*. The measurement contribution function measures the amount of light carried along path $\bar{x}$ to pixel j. It is a product of the light emitted by the object at the end of the path, factors accounting for the geometry of the path, the BRDF values at each path vertex and the sensor responsivity. The area product measure is the product of the canonical area measures of the objects at each path vertex.[1]

The path integral can be estimated using Monte Carlo integration. An unbiased estimate for the pixel value I_j is given as

$$\hat{I}_j = \frac{1}{N} \sum_{i=1}^{N} \frac{f_j(\bar{x}_i)}{p(\bar{x}_i)} \tag{2.13}$$

where N is the number of samples taken, $p(\bar{x}_i)$ is the probability density function of a probability distribution over path space and $\bar{x}_1, \ldots, \bar{x}_N \in \Omega$ are paths sampled using this distribution. The unbiasedness of this estimate follows directly from the definition and the definition of the expected value.

The choice of the probability distribution has a big impact on the convergence of the algorithm. First of all, the paths are never sampled uniformly from all paths, since the majority of possible paths carries no light and does not contribute to the final image. Instead, paths are generated by starting from the camera and then tracing a path through the scene, which is then at one point randomly connected to a light source. The resulting algorithm is called *path tracing* [Kaj86].

Since computation power is limited, one can usually only evaluate a fairly limited number of samples per pixel. The number of samples required for a completely converged image is often very high, since the integrand is high dimensional. The estimate computed using path tracing suffers from a high variance, i.e. noise. There are many different approaches to reduce noise in path tracing. The probability density used to sample paths can be created using a variety of strategies, which can drastically improve the convergence in many common scenarios. [Vea98] Another approach is to filter the rendered image. This does introduce bias, but can be very effective in reducing MSE.

[1] For a manifold $\mathcal{M}$, the canonical area measure dA is defined such that $\int_\mathcal{M} 1 \, dA(x)$ equals the surface area of $\mathcal{M}$. A 3D scene can now be interpreted as a union of manifolds. The area product measure for a path is simply the product of the area measures at each path vertex.

2.3 Halide

The Halide language [RKAP+12] is a domain specific programming language which primarily focuses on high performance image processing. Halide follows a functional paradigm and separates the *functionality* from the *schedule* of an algorithm. This allows the programmer to optimize the performance of his implementation by trying out different schedules without affecting the correctness of the computations. The aim of Halide is not automatic performance optimization, but giving the programmer the means to optimize code more quickly than by using conventional tools. Traditionally, optimization often reduced the readability and maintainability of the code. To use for example vectorization in C++ , one has to use platform specific instructions, which make the code fairly difficult to read for most programmers. Using Halide, optimization does not diminish the quality of the code.

Halide is not Turing complete, since it is not possible to write infinite loops in Halide.[2] Conceptually, a Halide program consists of a number of for-loops with known iteration counts. Branching using if-statements is also supported. Halide then allows the programmer to reorder the computations and to specify where temporary results are stored. The loops can be parallelized on the CPU or GPU and vectorization is also supported. All this can be controlled manually by the programmer with very little programming effort. All index calculations and memory management operations are done automatically.

Halide also supports automatic *sliding window optimizations* and *storage folding*. Sliding window optimization allows results from previous calculations to be reused. Storage folding is used to minimize the amount of storage used for temporary buffers. The programmer does not have much control over these two optimizations, which is contrasting the philosophy of Halide of allowing the programmer to control the schedule.

Since it abstracts the algorithm from the underlying implementation, Halide is very portable. Halide code can be compiled for a variety of architectures and builds on the LLVM [LA04] compiler framework. The Halide language provides a C++ front end in which Halide programs can be specified. There also exists a Python binding, but the C++ front end is better maintained and more commonly used.

The most important primitive in Halide programs are Halide functions. They are used to specify the algorithm and are also used for scheduling. A Halide function is always a total function mapping a tuple of integers to a scalar or tuple of arbitrary numerical type. A simple three-by-three box blur can for example be specified as part of a regular C++ program as

[2] It is however possible to call external C functions from within Halide code, which can then do arbitrary computations. This functionality is thought for special use cases, such as using external C libraries, which will not be covered here.

```
1  Image<float> in = loadImage(...)
2  Var x, y, xo, yo, xi, yi;
3  Func img, bX, bY;

4  img(x, y) = in(clamp(x, 0, in.width()-1,
5                       clamp(y, 0, in.height()-1));
6  bX(x, y) = (img(x-1, y) + img(x, y) + img(x+1, y))/3;
7  bY(x, y) = (bX(x, y-1) + bX(x,y) + bX(x, y+1))/3;

8  bY.tile(x, y, xo, yo, xi, yi, 256, 32).parallel(yo)
9    .vectorize(xi, 8);
10 bX.compute_at(bY, yi).store_at(bY, xo).vectorize(x, 8);
```

The description of the blur algorithm (blue) is independent from its schedule (green).[3] This example code here is scheduled to run on the CPU. We mostly used Halide to run code on the GPU, but CPU scheduling is better suited to explanatory purposes, as it is slightly less technical.

The input image is assumed to be a black and white image loaded into the buffer *in*. The input is first wrapped into the function *img*. The wrapper function clamps all access to the input buffer in order to prevent out of bounds access. The filter then first blurs the image horizontally. The result is then used as input to the vertical blur pass.

The interesting part about this code is its schedule. The computation of the vertical blur pass is split into tiles of 256×32 pixels, where the variables *xo* and *yo* index the tiles and *xi* and *yi* index pixels within tiles. The code is parallelized over *yo*, which means rows of tiles are processed in parallel. Within each tile, the vertical blur is vectorized along the horizontal direction, where 8 elements are collected into one vector. The result of the horizontal blur pass is computed as needed for each row of the vertical blur pass. This improves memory access performance, since the horizontal blur result is most likely still in cache while being used by the second blur pass. The result of the horizontal blur is stored at the level of the loop over *xo*. Each thread just stores the result of the horizontal blur for the currently processed tile. In fact, there is no need to store the whole result of bX for the current tile, since only three rows of it are required simultaneously. Halide thus automatically folds the buffer to store the horizontal blur into a buffer containing only the three lines of bX currently used. This buffer has then the size 256×4, because the tiles are 256 pixels wide and we need to store 3 rows in each tile (Halide rounds the 3 up to the next power of two). Leaving out vectorization for brevity, the described schedule is in pseudocode:

[3] The schedule is an updated version of the schedule proposed in the original paper by Ragan-Kelley et al. [RKAP+12]. The updated version has been proposed by Andrew Adams, 5.1.2015, `https://lists.csail.mit.edu/pipermail/halide-dev/2015-January/001341.html`, last retrieved 28.7.2015.

```
1  parallel for all rows of tiles yo
2      for all tiles xo in row yo
3          allocate buffer for bX of size 256 × 4
4          compute first three rows of bX
5          compute first row of bY from these
6          for yi = 1 to 31
7              compute row yi + 1 of bX
8              compute row yi of bY
9          free buffer for bX
```

The Halide code does not yet compute anything. The following code evaluates the function bY at each pixel of the output buffer *out*:

```
1 Image<float> out(in.width(), in.height());
2 bY.realize(out);
```

The Halide function bY would in this case be just-in-time (JIT) compiled when the code is executed as part of the surrounding C++ program. Halide also allows the code to be compiled ahead of time and then be called as a normal C function, instead of using "realize". Compiling ahead of time is usually desirable for execution speed, but JIT compilation can be useful to quickly see results, without having to set up the whole ahead of time compilation process.

Since it can be difficult to completely understand what is going on in terms of scheduling, Halide provides a variety of debug options. It is possible to get a pseudocode representation of the compiled pipeline, which precisely shows when each function is evaluated and where the results are stored. The pseudocode also shows where memory is allocated and the size of each buffer. This proved to be very useful to find optimization potential and also to identify bugs in Halide. It is even possible using Halide to create animations, which visualize how schedules access memory.

Chapter 3

Robust Denoising using Feature and Color Information

3.1 Overview

Evaluating the integral in Equation (2.12) using Monte Carlo integration results in a fairly noisy image. Even high numbers of samples per pixel often cannot eliminate the noise completely. The *Robust Denoising using Feature and Color Imformation* (RDFC) algorithm by Rousselle et al. [RMZ13] tries to remove this noise after the rendering process has finished. This allows to get high quality results in much less time than without filtering. The filtering step reduces the variance at the cost of introducing bias to the estimate of the integral, but usually leads to a significant reduction in relative mean squared error. There exist a variety of techniques which try similar things, but none does significantly improve over RDFC.

Many recent algorithms for denoising Monte Carlo renderings are based on two key ideas: using *feature information* to guide the filtering process and employing a statistical *error estimate* for optimal filter parameter selection.[ZJL+15]

In the context of rendering, the term "feature" refers to information about the rendered scene, that can be collected during the rendering process without much overhead. This additional information is usually stored for each pixel of the output image. Typical features are the normals of the objects in the scene, texture information and the position of the objects. Because these factors need to be computed for the path tracing algorithm, it does not incur much overhead to store them to separate buffers. Denoising algorithms that rely on features, assume that the features are highly correlated to the true color image and therefore guide the filtering of the noisy color image by the feature information. The features typically have a lot less variance than the sampled color information. There are far fewer random choices involved when sampling the features. Most features only depend on the primary hit of a ray in the scene. Despite the relatively simple sampling process, the features are not completely noise-free. Some noise can still occur at the edges of objects or in the presence

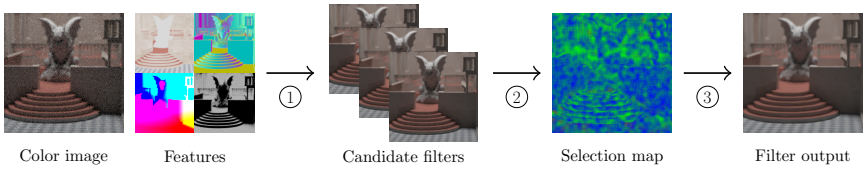


Figure 3.1: High-level overview of the RDFC filter. The color and feature information is used to compute three candidate filters (step 1). For each candidate filter the MSE is estimated using SURE. A spatially varying candidate filter selection map is then constructed from these estimates to minimize the estimated MSE (step 2). The output is then computed as the sum of the candidate filters weighted by the selection map (step 3).

of motion or out of focus blur.

Similar to many other denoising algorithms, RDFC tries to denoise the input image by computing the denoised result at pixel p as a weighted sum of noisy pixels from a neighborhood of pixel p. The output of the filter is then given as

$$F_i(p) = \frac{1}{C(p)} \sum_{q \in \mathcal{N}_p} w(p,q) u_i(q) \ . \tag{3.1}$$

Here, $u_i(p)$ denotes channel i of the noisy image at pixel p, $\mathcal{N}_p$ the square neighborhood of pixel p, $w(p,q)$ filtering weights for the individual neighboring pixels and $F_i(p)$ channel i of the filtered output at pixel p. $C(p)$ is a normalization constant defined as $C(p) = \sum_{q \in \mathcal{N}_p} w(p,q)$.

If n noisy pixels with the same expected value and variance would be averaged, the result would still have the same expected value and n times less variance than the individual pixels. In practice it is unknown which pixels have the same expected value, but it's possible to heuristically average similar pixels together and get a significant improvement in mean squared error over the noisy input. The challenge is to find suitable filter weights. There is a trade-off between increasing bias and reducing variance. If the weights are not restrictive enough, unrelated pixels are mixed together, which results in a blurred, strongly biased result. If the weights are too restrictive, the noise does not get removed and the filter output still has a high variance.

The weights used in RDFC are based on the *non-local means filter* (NL-means) [BCM05, RKZ12] for the sampled color information and a *cross-bilateral filter* [TM98, LWC12] for the feature information. By using the less noisy features, the similarity between two pixels can be estimated much better than from just the noisy color image.

Many denoising algorithms can be controlled by a few user selected parameters, for which the optimal setting usually cannot be analytically determined. Therefore, using a statistical error estimate is crucial to select good parameter settings on a per pixel basis. In RDFC, the sensitivity of the filter to color and to feature information can be selected independently, which offers a great deal of control over the filter. There can be image regions, where using the feature information is very effective and the color values can be nearly ignored. But there can also be image content, that is not present in the features and would therefore be blurred too much, if the filter is not sensitive enough to changes

in color. The parameters thus need to be selected spatially varying in order to achieve high quality results. The RDFC algorithm filters the input three times using different filter parameters and then combines these results by estimating the mean squared error of each using *Stein's unbiased risk estimate* (SURE) [Ste81]. SURE allows to estimate the mean squared error of an estimator in an unbiased way. This risk estimate has been applied to different denoising problems, including to denoising of Monte Carlo renderings by Li et al. [LWC12].

The first section of this chapter describes the computation of filtering weights from the noisy color image, the second section gives the details of the weight computation from the features, and the third section covers the application of SURE to RDFC. After this the RDFC algorithm is explained and its implementation using Halide described.

3.2 Non-local means color weights

3.2.1 The non-local means filter

The RDFC filter computes NL-means filtering weights from the noisy color image. The NL-means weights can be understood and motivated by first looking at the the weighting function used in the original bilateral filter by Tomasi and Manduchi [TM98]. The bilateral filter is an edge-aware smoothing filter which computes the output as a weighted sum using the weighting function

$$w(p,q) = exp\left(\frac{-d^2(p,q)}{2\sigma_r^2}\right) exp\left(\frac{-(\|p-q\|_2^2)}{2\sigma_s^2}\right), \tag{3.2}$$

where

$$d^2(p,q) = \sum_{i=1}^{3}(u_i(p) - u_i(q))^2 \tag{3.3}$$

and p and q are the 2D coordinates of the center and neighboring pixel and $\|p-q\|_2^2$ is the squared euclidean distance. The first exponential term weights down pixels with a large difference in value and the second exponential term pixels that are spatially far away from the center pixel. The parameter σ_s is the spatial filtering range and controls how large the neighborhood considered by the filter is. The intensity filtering range σ_r determines how strict the filtering is. A large value of σ_r will make the filtering less sensitive to pixel differences and similar to standard Gaussian filtering. If σ_r is small, the filtering weights are very restrictive and only very similar pixels will be averaged together.

The bilateral filter is limited by the quality of the distance estimate between two pixels. Because the input image is noisy, the measured distance is also corrupted by noise and not very reliable. The non-local means filter tries to overcome this issue by computing a more robust distance estimate. Instead of directly comparing the value of two noisy pixels, the NL-means filter compares small patches around these two pixels. The distance between two patches centered at pixels p and q is calculated as

$$d^2(P(p),P(q)) = \frac{1}{3(2f+1)^2}\sum_{i=1}^{3}\sum_{n\in P(0)}\Delta_i^2(p+n,q+n)\ , \tag{3.4}$$

where f is the radius of the compared patches and $P(0)$ denotes all pixel offsets within a patch. The function Δ_i^2 computes the distance in channel i between pixel p and q. In the original NL-means formulation $\Delta_i(p,q)$ is defined as

$$\Delta_i(p,q) = (u_i(p) - u_i(q))^2 - 2\sigma^2 \ , \tag{3.5}$$

where σ^2 is the noise variance. The standard NL-means filter assumes the variance of the noise to be uniform.

The subtraction of $2\sigma^2$ makes the distance estimate an unbiased estimate of the true squared distance between two pixels. The squared distance between two values corrupted by additive noise overestimates the true distance exactly by a factor of $2\sigma^2$. This follows from the basic properties of the expected value: Let $y_1 = x_1 + n_1$ and $y_2 = x_2 + n_2$ be two pixel values corrupted by additive noise. The random variables n_1 and n_2 are assumed to be independent and identically distributed (i.i.d) with expected value 0 and variance σ^2 and x_1 and x_2 are the true pixel values. The expected value of the squared difference between the noisy measurements is then:

$$\begin{aligned}
\mathrm{E}\left[(y_1 - y_2)^2\right] &= \mathrm{E}\left[(x_1 - x_2)^2 + 2(x_1 - x_2)(n_1 - n_2) + (n1 - n2)^2\right] \\
&= (x_1 - x_2)^2 + 2(x_1 - x_2)(\mathrm{E}[n_1] - \mathrm{E}[n_2]) + \mathrm{E}\left[(n_1 - n_2)^2\right] \\
&= (x_1 - x_2)^2 + \mathrm{E}[n_1^2] - 2\,\mathrm{E}[n_1]\,\mathrm{E}[n_2] + \mathrm{E}[n_2^2] \\
&= (x_1 - x_2)^2 + 2\sigma^2 \ .
\end{aligned}$$

Thus we get $\mathrm{E}[(y_1 - y_2) - 2\sigma^2] = (x_1 - x_2)^2$. Note that this also holds if we replace the true variance σ^2 by an unbiased variance estimate.

Comparing small patches significantly improves denoising quality compared to the standard bilateral filter. As small patches of pixels are compared, the filter does not need to be spatially constrained like the bilateral filter and is thus "non-local". There is no spatial weighting function. To reduce complexity, in practice still only pixels from a neighborhood of radius r around the central pixel are compared.

In Figure 3.2 on the next page the NL-means distance estimates are visualized for different patch radii f. For a radius of 0 the distances correspond to the distances used by the bilateral filter. The bilateral distances are very noisy. Increasing the patch radius f reduces the noise in, at the cost of slightly blurring the distance estimates.

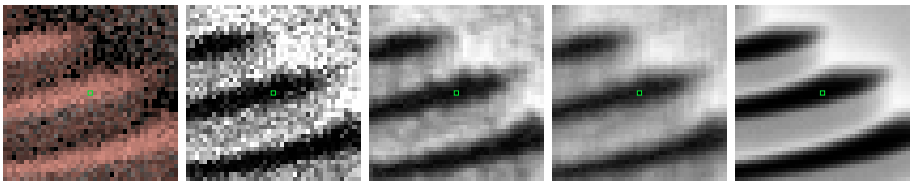
From the distance estimate the weights are then calculated as

$$w(p,q) = \exp\left(-\frac{\max(d^2(P(p), P(q)), 0)}{2\sigma^2 k^2}\right) \ . \tag{3.6}$$

Here k is a user selected scaling factor to control the sensitivity of the filter. The division by σ^2 makes sure, that the filter strength depends on the variance of the signal and the filter is consistent. The distance estimate is clamped to avoid negative distances, which could occur as the result of the variance subtraction.

3.2.2 Non-uniform variance

The previously described weights are not directly suitable for RDFC, because they assume the noise variance to be uniform. The variance in Monte Carlo



(a) Noisy image (b) $f = 0$ (c) $f = 1$ (d) $f = 2$ (e) True distances

Figure 3.2: Comparison of the NL-means distance estimates for different patch radii f. Visualized are the distances measured from the marked central pixel (scaled for optimal display). The distances in (b) – (d) can be compared to the true distances in (e), which are computed using the noise-free ground truth image.

rendered images is highly dependent on the structure of the scene and thus spatially non-uniform. The NL-means filter weights have to be modified to account for this. This can be done by defining the distance between two pixels as

$$\Delta_i^2(p,q) = \frac{(u_i(p) - u_i(q))^2 - (\mathrm{Var}_i[p] + \min(\mathrm{Var}_i[p], \mathrm{Var}_i[q]))}{\varepsilon + k_c^2(\mathrm{Var}_i[p] + \mathrm{Var}_i[q])} , \tag{3.7}$$

where k_c controls the sensitivity of the filter to color differences and ε is a small constant to avoid division by zero.[RKZ12] The variance term $2\sigma^2$ is replaced by the sum of the noise variance at pixel p and pixel q. The estimation of the per-pixel noise variance is non trivial and will be explained in Section 3.2.5. In the numerator the minimum of both variances is taken instead of the variance of the neighbor pixel. This prevents blurring if the variance of the neighboring pixel is higher than the variance at the center pixel. This introduces bias to the distance estimate, but improves overall filtering quality. The color weights for RDFC are then given as

$$w_c(p,q) = \exp(-\max(d_c^2(P(p), P(q)), 0)) , \tag{3.8}$$

where $d_c^2(P(p), P(q))$ is the distance estimate between two patches calculated using the above non-uniform distance formulation. To avoid confusion with the feature weights and distance estimates, the color weights and distances are written with the subscript c.

Note, that the above distance estimate assumes that the noise at pixels p and q is independent. This only holds, if the image uses a box filter as pixel filter. In rendering, the pixel filter describes how samples are averaged into image pixels. Using a box filter, each sample only contributes to one pixel in the image plane, and all samples within one pixel have the same weight. The noise of pixel p is thus independent from the noise of any other pixel q. This does not hold, if a pixel filter with a larger support is used, where each sample contributes to multiple pixels.

3.2.3 Patch-wise weight computation

The RDFC filter uses the patch-wise formulation of the NL-means filter. Instead of copying the pixel q to position p with weight $w_c(p,q)$, the whole patch $P(q)$

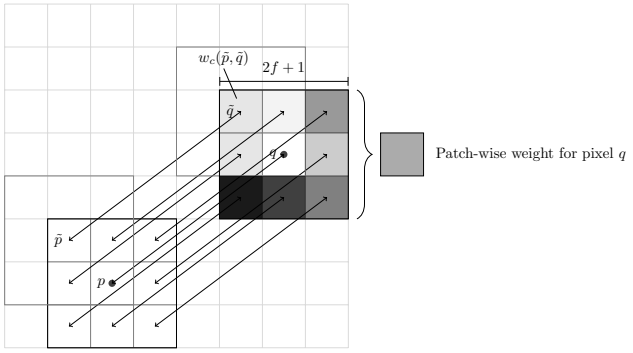


Figure 3.3: Patch-wise extension of the NL-means filter. First, the weights between the pixels with the same offset are calculated as the standard NL-means weights. The different weights for the pixels are visualized as grayscale values inside these pixels. The pixel q is then weighted by the average of these weights.

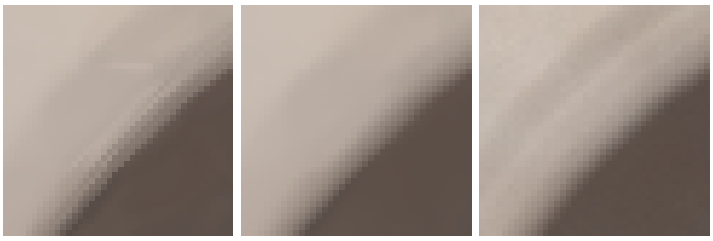
of $(2f+1)\times(2f+1)$ pixels is copied to the patch $P(p)$ with the weight $w_c(p,q)$. The pixel q is thus copied $(2f+1)\times(2f+1)$ times to position p, each time multiplied with a different weight. The patch-wise extension is equivalent to weighting pixel q by the average of all weights $w_c(\tilde{p},\tilde{q})$, where $\tilde{q}$ has the same offset to $\tilde{p}$ as q to p, i.e. $\tilde{q}-\tilde{p}=q-p$ and $\tilde{q}$ is in the patch of radius f around q. In Figure 3.3, the patch-wise weight calculation is visualized. In practice, all weights for the same offset between p and q are calculated at the same time and then filtered using a box filter of radius f to compute the averaged weights. Pixel q is then copied to pixel p just once using the averaged weight. Blurring the weights using a box filter is relatively cheap and gives a slightly smoother filter output. [RKZ12] From now on $w_c(p,q)$ will denote the patch-wise weights.

3.2.4 Symmetric weights

Another modification to the weight computation is the introduction of a *symmetric* distance computation. The observation has been made, that smooth gradients are filtered poorly by NL-means, because the filter is constrained to filter orthogonally to the direction of the gradient in the image. [RKZ12] In a symmetric structure denoising can be improved by averaging radially symmetric pixels. If the neighbor pixel q_1 is brighter than the central pixel p and neighbor q_2 darker by the same factor, the average of both neighbors will have the same value as the central pixel. The effect of the symmetric weight formulation is demonstrated in Figure 3.4 on the next page.

To use symmetries in the image, a symmetric distance estimate is computed for two radially symmetric pixels q_1 and q_2. The symmetric distance estimate for these two pixels is used if there is a high enough chance that the data is actually symmetric. The symmetric distance estimate is

$$\Delta_i^2(p,\bar{q}) = \frac{(u_i(p)-u_i(\bar{q}))^2 - (\mathrm{Var}_i[p]+\mathrm{Var}_i[p,\bar{q}])}{\varepsilon + k_c^2(\mathrm{Var}_i[p]+\mathrm{Var}_i[\bar{q}])}\ , \tag{3.9}$$



(a) Using only asymmetric weights (b) Using symmetric weights (c) Reference image

Figure 3.4: Effect of symmetric weights when filtering a smooth gradient. Filtering using only the asymmetric weights gives clear banding artifacts in the smooth gradient. The symmetric weight formulation allows to recover the smooth gradient structure much better.

where

$$\begin{aligned} u_i(\bar{q}) &= \frac{1}{2}(u_i(q_1) + u_i(q_2)) \\ \mathrm{Var}_i[\bar{q}] &= \frac{1}{4}(\mathrm{Var}_i[q_1] + \mathrm{Var}_i[q_2]) \\ \mathrm{Var}_i[p, \bar{q}] &= \frac{1}{4}(\min(\mathrm{Var}_i[p], \mathrm{Var}_i[q_1]) + \min(\mathrm{Var}_i[p], \mathrm{Var}_i[q_2])) \ . \end{aligned}$$

This distance estimate compares the central pixel with the average of the two symmetric neighbors. If the estimated distance is sufficiently small, it can be assumed that the data is symmetric. In RDFC the final weight is a linear combination of the symmetric and asymmetric weights for each pixel. The weighting factor a is computed as

$$\tilde{a} = \begin{cases} \frac{w_c(p,\bar{q})}{w_c(p,q_1)+w_c(p,q_2)} - 1 & d^2(P(p), P(q_1)) < d^2_{max} \text{ and } d^2(P(p), P(q_2)) < d^2_{max} \\ 0 & \text{otherwise} \end{cases}$$

$$a = \max(0, \min(1, \tilde{a})) \ ,$$

where $d^2_{max} = 25$ is a constant and $w_c(p, \bar{q})$ is the symmetric weight computed from the symmetric distance $\Delta^2_i(p, \bar{q})$. Essentially, the symmetric weight term only affects the final weight if it is larger than the sum of the asymmetric weights and both asymmetric distances are below some threshold.

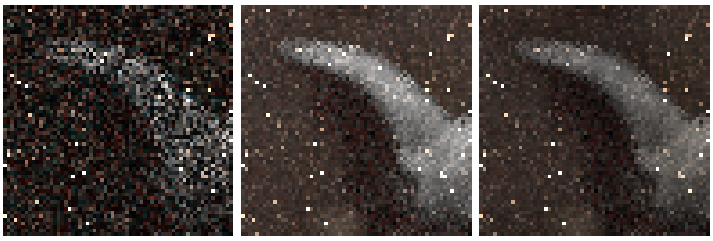
The asymmetric weights are then combined with the symmetric weights to give the final weights for pixels q_1 and q_2 as

$$\begin{aligned} \tilde{w}_c(p, q_1) &= a \cdot w_c(p, \bar{q}) + (1 - a) \cdot w_c(p, q_1) \\ \tilde{w}_c(p, q_2) &= a \cdot w_c(p, \bar{q}) + (1 - a) \cdot w_c(p, q_2) \end{aligned}$$

In order to simplify notation, we will denote the new weights again by $w_c(p, q)$ in the following. There is no need to refer to the original asymmetric weights again.

3.2.5 Variance estimation and two-buffer filtering

To achieve good denoising performance with the previously described filter, having a good variance estimate for the noise at each pixel is key. The noise



(a) Two-buffer variance (b) Sample variance (c) Scaled sample variance

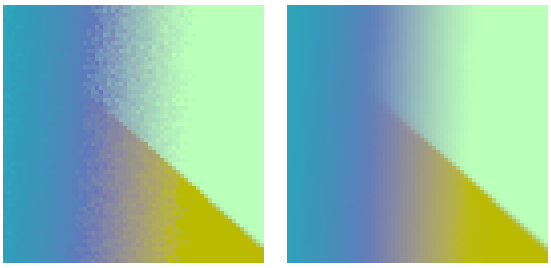
Figure 3.5: Comparison of the two-buffer variance to the biased sample variance and the scaled sample variance. The scaled sample variance has the same magnitude as the two-buffer variance estimate, but the noise characteristics of the sample variance (all images are scaled by the same factor for optimal display).

variance of the output pixel can be estimates by computing the sample variance of all Monte Carlo samples contributing to one pixel and dividing it by the number of samples, to get a variance estimate for the sample mean. In case of naive random sampling, where the samples at each pixel are i.i.d, this gives an unbiased estimate of the variance at each pixel. However, the samples used in Monte Carlo path tracing are often pseudo-random samples drawn from a low-discrepancy sequence. These sequences are carefully constructed to achieve faster convergence of the Monte Carlo integration process. [Vea98] When using low-discrepancy sampling, the sample variance is biased, since the individual samples are in fact correlated. The sample variance then overestimates the variance of the noise. [RKZ12]

A possible solution to this problem is to accumulate samples into two different buffers instead of just one. This means half the samples are written to one buffer and the other half to other one. The samples accumulated in each buffer are generated from a different low discrepancy sequence and are thus independent. The two buffers then give two estimates for a single pixel, from which a variance estimate of the sample mean at pixel p can be computed as $\mathrm{Var}_i[p] = \frac{1}{4}(v_i(p) - w_i(p))^2$, where v and w are the two different buffers. This variance estimate will be called *two-buffer variance* from now on.

Even though unbiased, the two-buffer variance is very noisy, since it is computed from just two observations. On the other hand, the biased sample variance is less noisy. In RDFC these two different variance estimates are combined by scaling the biased sample variance to have a similar magnitude as the two-buffer variance. This is done by first smoothing both variance estimates with a large 21×21 pixel box filter and then computing the per pixel ratio of the blurred estimates. The ratio is then used to scale the sample variance. In Figure 3.5 both variance estimates and the result of the variance scaling are displayed.

The two-buffer approach also allows for a simple estimation of the output variance of the NL-means filter. The filtering weights can be computed using the mean of the two buffers, but be applied to each buffer separately. The mean of the two filtered buffers is then exactly the same as if the filter would have been run on the mean directly. The NL-means filter is linear for given filtering weights, as it computes a weighted sum at each pixel. The two filtered



(a) Noisy feature (b) Prefiltered feature

Figure 3.6: Prefiltering of the features demonstrated on the example of the normals. The object in the foreground (blue) is strongly blurred since it is out of focus, while the background (green/yellow) is in focus and practically noise-free. The noise is caused mainly by the out of focus blur and is removed effectively by the NL-means filter. The hard edge in the background is preserved.

buffers can be used to estimate the residual variance after filtering. Note, that this is not an unbiased estimate, since the filtering weights are correlated with both buffers. To get an unbiased estimate, one would need to compute the filtering weights for each buffer separately. This would then give an unbiased variance estimate, at the cost of changing the filter result. In practice, it is more important to have high quality filtering weights from the mean of the two buffers than an unbiased estimate of the output variance.

3.3 Cross-bilateral feature weights

3.3.1 Feature prefiltering

The sampled features are not completely noise-free. Effects like out of focus blur or motion blur can cause the features to be noisy. Even without these effects there is often some noise at sharp edges. There are also features which are inherently more noisy than others. An example for this is the visibility feature used in RDFC, which stores whether a light source is directly visible or not. The visibility captures direct lighting shadows quite effectively, but is more noisy than other features. The noise in the features is usually still fairly limited compared to the noise of the color image. Therefore, denoising the features is much easier than denoising the color image. In RDFC the features are prefiltered using the same NL-means filter formulation which is used to calculate the color weights. This is very effective, as shown in Figure 3.6.

3.3.2 Feature weights

After the prefiltering, the features are used to calculate feature weights, which complement the existing color weights. The feature weights are computed using a cross-bilateral filter. Similar to the NL-means algorithm, the cross-bilateral filter tries to improve on the standard bilateral filter by computing a better distance estimate. Instead of extending the distance measure itself to be more robust, the cross-bilateral filter computes the weights based on one or more guide

images. The guide images should have less noise than the noisy color image and have a similar structure. In RDFC the guide images for the cross-bilateral filter are the prefiltered features.

Similar to the color distance, the feature distance between two pixels in channel i of feature f_j is measured as

$$\Phi^2_{j,i}(p,q) = \frac{(f_{j,i}(p) - f_{j,i}(q))^2 - (\mathrm{Var}_{j,i}[p] + \min(\mathrm{Var}_{j,i}[p], \mathrm{Var}_{j,i}[q]))}{\varepsilon + k_f^2(V_{j,i}[p] + V_{j,i}[q]))} \ , \quad (3.10)$$

where k_f controls the sensitivity of the filter to feature information and ε is a small constant which helps to avoid division by zero. $\mathrm{Var}_{j,i}[p]$ is the variance of feature j in channel i. $V_{j,i}[p]$ is defined as $V_{j,i}[p] = \max(\tau_j, \mathrm{Var}_{j,i}[p], \|\nabla_{j,i}[p]\|^2)$ and $V_{j,i}[q]$ is defined equivalently. The gradient magnitude term $\|\nabla_{j,i}[p]\|^2$, which essentially measures local feature contrast, helps to avoid constraining the filter too much if the central pixel p is directly on an edge of feature j. The gradient magnitude is computed from a finite difference estimate of the image space gradient. The value τ_j is a per feature threshold. It prevents the feature weights from becoming too restrictive if the feature variance and gradient are both close to zero. For most features τ_j is set to a global threshold τ. For features, that have values of a different order of magnitude than $[0, 1]$, e.g. the vertex position, the threshold is set as $\tau_j = \tau \cdot (\max f_j(p))^2$, where the maximum is taken over all pixels. This prevents these features from being more important than those with smaller values. The per-channel distances are then summed up to get the final feature distance as

$$\Phi^2_j(p,q) = \frac{1}{c_j} \sum_{i=1}^{c_j} \Phi^2_{j,i}(p,q) \ , \quad (3.11)$$

where c_j is the number of channels of feature j. From the distance estimate $\Phi^2_j(p,q)$ the weights for feature j are computed as

$$w_{f_j}(p,q) = \exp(-\max(\Phi^2_j(p,q), 0)) \quad (3.12)$$

Given the feature weights $w_{f_j}(p,q)$, the final feature weights are then defined as

$$w_f(p,q) = \min_{j \in \{1,\dots,M\}} w_{f_j}(p,q) \ , \quad (3.13)$$

where M is the number of features. This always selects the most constraining filter weights. If there is a strong edge in any feature, the filter should not blur across this edge. The same approach is taken in combining the color weights $w_c(p,q)$ and the feature weights $w_f(p,q)$. The final weights of the filter are then

$$w(p,q) = \min(w_c(p,q), w_f(p,q)) \quad (3.14)$$

This also shows again why the threshold τ_j is needed. Without the threshold, one single noise-free feature could constrain the filter too much, even if the color image is very noisy.

The symmetric weights formulation is also used for the feature weights $w_{f_j}(p,q)$. The numerator takes on the same form as for the color weights in Equation (3.7). In the denominator, the formulation is the same as with

the color weights, except that, like in the asymmetric feature weights, all the variance terms are replaced by the maximum over the variance, the gradient magnitude and the threshold τ_j.

Note, that the formulation of the feature distances given in Equation (3.10) on the previous page slightly differs from the original definition given in the RDFC paper [RMZ13]. The formulation used here is the one which is implemented in the CUDA code published by the authors. The Halide version of RDFC therefore also uses these feature distances.

3.4 Stein's unbiased risk estimate

3.4.1 Definition

Stein's unbiased risk estimate (SURE) [Ste81] is an unbiased estimate for the mean squared error of a statistical estimator. The SURE is used to estimate the per-pixel MSE of each candidate filter used in RDFC. The per-pixel MSE of a filter is estimated as

$$\mathrm{SURE}(p) = \sum_{i=1}^{3} \mathrm{SURE}_i(p) = \sum_{i=1}^{3} \|F_{u_i}(p) - u_i\|^2 - \sigma_i^2 + 2\sigma_i^2 \frac{\partial F_{u_i}(p)}{\partial u_i} \ . \quad (3.15)$$

Here, u_i denotes the noisy pixel value $u_i(p)$, σ_i^2 is the variance in channel i at pixel p and $F_{u_i}(p)$ is the filter output in channel i at pixel p, depending on the value u_i of the unfiltered pixel.

Because the Monte Carlo estimate u_i is the mean of a number of n i.i.d samples, it follows by the central limit theorem, that it converges in probability towards a normal distribution $N(x_i, \sigma_i^2)$, where x_i is the true pixel value and σ_i^2 is the true variance of the Monte Carlo estimate with n samples. It turns out, that assuming the noise in the Monte Carlo estimate to follow a normal distribution is a good approximation of the true noise distribution even for a small number of samples. [LWC12]

Under the assumption, that the noisy pixel value u_i is given as the true value x_i plus additive noise n_i, where n_i follows a Gaussian distribution with mean zero and variance σ_i^2, the unbiasedness of the estimator can be proved using elementary probability theory. A simple proof is given by Blu and Luisier [BL07].

3.4.2 Application to RDFC

The SURE can be computed without much overhead alongside the existing filtering routine in RDFC. Instead of the true noise variance, the scaled sample variance is used. The squared difference to the unfiltered value is also trivial to compute. The differential expression is, in contrast to Li et al. [LWC12], not analytically calculated, but approximated using a finite difference formulation:

$$\frac{\partial F_{u_i}(p)}{\partial u_i} \approx \frac{F_{u_i+\delta}(p) - F_{u_i}(p)}{\delta} \ , \quad (3.16)$$

where δ is a small value set proportionally to the value of the unfiltered image as $\delta = 0.01 \times u_i$.

The differential is then computed by simultaneously computing the filtered result $F_{u_i}(p)$ in each pixel and the result of the filter if the central pixel is replaced by $1.01 \times u_i$.

3.5 The RDFC algorithm

The inputs of the RDFC algorithm are the noisy color image c, the color sample variance c_svar, the array of features $feat[]$, the array of feature sample variances $feat\_svar[]$ and the user specified filter radius R. The complete RDFC filter is given by pseudocode on the following page.

All color and feature data is assumed to be stored in two separate buffers, which can then be used to estimate the variance. The algorithm uses the following functions

- ScaleSampleVar: Implements the previously described sample variance scaling technique. The inputs are the sample variance and the buffers used to compute the two-buffer variance. The output is the sample variance scaled by the two-buffer variance.
- TwoBufferVar: Computes a variance estimate from two buffers. Since the estimate is very noisy, it is blurred using a small Gaussian filter with a standard deviation of 0.5.
- FeatureFilter: Implements the color and feature filter routine, which filters the input data using the previously described color and feature weights. The inputs are a noisy image, a guide color image, the variance of the guide color image, the array of features and the array of feature variances, as well as a parameter data structure. Note that for $k_f = \infty$ the filter degrades to the NL-means filter and does not use any feature information. The noisy image is assumed to be stored in two buffers. The output does then consist of the two individually filtered buffers and optionally the gradient term used to compute the SURE.
- Sure: Computes the SURE from the noisy color image, its variance, the filter result and the filter gradient term.

In lines 2–4, the sample variance scaling takes place. After this step, the input sample variances are not used anymore. The features are then prefiltered in lines 6–9 using the NL-means filter. The residual variance of the features is estimated using the TwoBufferVar function.

After these preprocessing steps, the color and feature filter is evaluated three times using different parameters. The first candidate filter is most sensitive to differences in the color image. The second candidate filter is less sensitive, since it uses a larger patch radius. The third filter then only uses feature information and does ignore the noisy color image entirely.

For each candidate filter, the SURE is computed and filtered using the NL-means filter in lines 18–21. Even though unbiased, the SURE suffers from variance inherited from the different terms used in its computation. Filtering the error estimate can thus increase it's quality in terms of MSE to the true error of each candidate filter. In the implementation the three single channel

RDFC$(c, c_svar, feat[], feat_svar[], R)$

// Sample variance scaling
var_c = SCALESAMPLEVAR(c_svar, c)
for $j = 1$ **to** M
    $var\_f[j]$ = SCALESAMPLEVAR$(feat_svar[j], f[j])$
// Feature prefiltering
$p = \{k_c = 1, k_f = \infty, f = 3, r = 5\}$
for $j = 1$ **to** M
    $flt\_f[j]$ = FEATUREFILTER$(f[j], f[j], var_f[j], nil, nil, p)$
    var_flt_f = TWOBUFFERVAR$(flt_f[j])$
// Candidate filters
$p = \{k_c = 0.45, k_f = 0.6, f = 1, r = R, \tau = 10^{-3}\}$
$[r, d_r]$ = FEATUREFILTER$(c, c, var_c[j], flt_f[], var_flt_f[], p)$
$p = \{k_c = 0.45, k_f = 0.6, f = 3, r = R, \tau = 10^{-3}\}$
$[g, d_g]$ = FEATUREFILTER$(c, c, var_c[j], flt_f[], var_flt_f[], p)$
$p = \{k_c = \infty, k_f = 0.6, f = 1, r = R, \tau = 10^{-4}\}$
$[b, d_b]$ = FEATUREFILTER$(c, c, var_c[j], flt_f[], var_flt_f[], p)$
// Filter SURE estimates
$p = \{k_c = 1.0, k_f = \infty, f = 3, r = 1\}$
e_r = FEATUREFILTER(SURE$(c, var_c, r, d_r), c, var_c, nil, nil, p)$
e_g = FEATUREFILTER(SURE$(c, var_c, g, d_g), c, var_c, nil, nil, p)$
e_b = FEATUREFILTER(SURE$(c, var_c, b, d_b), c, var_c, nil, nil, p)$
// Compute binary selection maps
$sel_r = e_r < e_g$ && $e_r < e_b$ && $d_r \leq d_g$? 1 : 0
$sel_g = e_g < e_b$ && $(e_g \leq e_r$ || $d_r > d_g)$? 1 : 0
$sel_b = e_b \leq e_g$ && $(e_b \leq e_r$ || $d_r > d_g)$? 1 : 0
// Filter selection maps
$p = \{k_c = 1.0, k_f = \infty, f = 3, r = 5, \tau = 10^{-3}\}$
sel_r = FEATUREFILTER$(sel_r, c, var_c[j], nil, nil, p)$
sel_g = FEATUREFILTER$(sel_g, c, var_c[j], nil, nil, p)$
sel_b = FEATUREFILTER$(sel_b, c, var_c[j], nil, nil, p)$
// Candidate filter averaging
$pass1 = r \cdot sel_r + g \cdot sel_g + b \cdot sel_b$
// Second filtering pass
$p = \{k_c = 0.45, k_f = 0.6, f = 3, r = \lfloor R/2 \rfloor, \tau = 10^{-4}\}$
var_pass1 = TWOBUFFERVAR$(pass1)$
$pass2$ = FEATUREFILTER$(pass1, pass1, var_pass1, nil, nil, p)$
return *pass2*

SURE images are combined to one color image and then filtered in one step to save some time.

The filtered SURE is then used to compute a binary selection mask for each filter in lines 23–25. This step at each pixel selects the filter with the lowest SURE. Since the first filter has the tendency to keep too much noise, it is only selected if it filters more than the second one. This is measured by comparing the differential terms d_r and d_g. If d_r is smaller than d_g in pixel p, changing the noisy pixel p does affect the result r less than the result g and thus the candidate filter r filters this pixel more aggressively.

The filter selection maps are then filtered using the NL-means filter in lines 27–30 to get a smooth selection map. Again, the three different selection maps are filtered jointly. Using the smooth selection map, the result is then simply a weighted sum of the candidate filters. It's output variance can then be estimated using the two-buffer technique. The result $pass1$ is then filtered again in a second pass to remove residual noise.

3.6 Implementation using Halide and results

We implemented the filter as a standalone program, where the input is loaded from OpenEXR files. The original CUDA code from Rousselle et al. [RMZ13] is integrated in the PBRT rendering framework [PH10] and cannot be used on file input. Aside from this difference, the implementation in Halide follows closely the original CUDA code. The CUDA code is already fairly optimized and takes around 4.3 s to filter a 1 megapixel image on a Nvidia GTX Titan X GPU. The time for copying the CPU buffers to GPU memory is included in this and all following performance measurements. The Halide RDFC code has both a GPU and a CPU schedule. The CPU schedule was implemented mostly for debugging reasons and did not require much additional work, as only the scheduling of the code needed to be adjusted.

We implemented all functions used in the pseudocode as individual Halide functions and the high level algorithm structure directly in C++. One large monolithic Halide function to compute the whole RDFC filter would work in theory, but would make it impossible to compile different parts of the pipeline separately. The compilation of Halide code can get quite slow for longer functions. Compiling the whole RDFC Halide code for the GPU takes around 45 seconds and compiling the Halide CPU code takes about 4.5 minutes. Therefore it's very useful to be able to recompile only parts of the code, especially for experimenting with different schedules.

The main performance bottleneck of the RDFC algorithm is the FeatureFilter routine. The key optimization which has to be made for this function, is to avoid computing the same pixel distances over and over again. A naive implementation of the NL-means weight computation would compute the distance $\Delta_i^2(p, q)$ between pixels p and q many times, since there are multiple patches where the term $\Delta_i^2(p, q)$ is needed. It turns out, that this can easily be avoided by reordering the computation of the NL-means weights. All distances between pixels with the same offset are computed at the same time and then convolved with a box filter of radius f to get all distances between patches of the same offset. This is essentially the same computation as has been described for the patch-wise weight computation, but on the pixel distances instead of the filter

Image size	**1024 × 1024**	
Radius R	**10**	**20**
CUDA code	4.3	13.3
Halide (GPU)	3.5	10.2
CPU code	396.5	1246.9
Halide (CPU)	177.7	647.8

Table 3.1: Execution times in seconds for the different implementations of RDFC and user selected radii R. Times are averaged over five runs of the RDFC filter. The CPU code was executed on two Intel E5645 processors with a total of 12 cores at 2.67 GHz and the GPU code on a Nvidia GTX Titan X.

weights. This optimization is used in the original CUDA code and in the Halide implementation.

Halide allows for some optimizations in the code, which would have been more difficult to perform in the original CUDA code. Using Halide it is for example possible to merge subsequent computations to take place in one GPU kernel. This allows to reduce the number of global memory accesses by using shared memory or registers to store intermediate results. Of course these kind of optimizations could be made in the original CUDA code too, but only at the cost of degrading code readability and maintainability. In Halide the readability of the code is not affected by such optimizations. The Halide implementation of RDFC uses shared memory and loop unrolling where it gave a performance increase. All the loops over color channels are unrolled at compile time. This gave a performance increase of about half a second at the cost of requiring a recompilation of the code if images with a different number of color channels should be processed. This should however not be an issue for most use cases.

There are also limitations in Halide, which can cost some performance compared to the original code. The CUDA implementation by Rousselle et al. reuses the same few temporary buffers in each call of FeatureFilter. Halide cannot do this by default and the temporary buffers are reallocated in each call of FeatureFilter. A workaround for this could be to override the Halide default memory allocator, but the performance gain would most likely not be worth the significant amount of work required for this.

The Halide GPU implementation uses OpenCL as compilation target, which gives slightly better performance than compiling the Halide code for CUDA. The Halide GPU implementation is faster than the existing CUDA code by about 20%. See the detailed benchmark results in Table 3.1. Even though there is probably still some optimization potential in the Halide code, a further speedup of the code by a significant factor seems quite unlikely.

The Halide CPU schedule is compared to the existing RDFC CPU code. This code is a direct port of the CUDA implementation, where all CUDA kernel calls were replaced by parallel for-loops. The Halide CPU code outperforms the existing RDFC CPU code by factor of 2. The Halide code makes use of vectorization and parallelizes over tiles instead of scanlines. It also uses the fast blur schedule presented in Section 2.3. Again, these optimizations would be possible in the original CPU code too, but much more time-consuming to manually implement. Even though the CPU code is far from being useful, it is

still interesting to see how much can be gained using Halide over a naive CPU implementation.

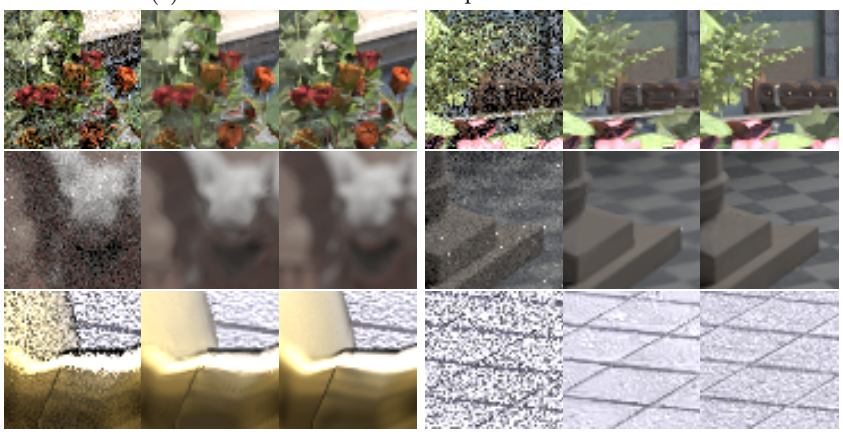
Implementing RDFC using Halide did not provide any significant problems. The Halide code is slightly more compact than the existing CUDA and CPU code. Since the existing code included some additional functions required for adaptive sampling and debugging, comparing lines of code is not very meaningful. Both implementations are well over 1000 lines of code. The output of the Halide implementation matches the output of the original code almost perfectly, but there are some small differences. The RMSE of the Halide output to the CUDA output is however around one to two order of magnitudes smaller than the RMSE to the converged reference image and thus negligible. A few example outputs of the filter are shown in Figure 3.7 on the following page.

(a) Reference images

(b) Noisy images

3.07×10^{-2} 1.3×10^{-3} 6.63×10^{-2}

(c) RDFC results and the respective RMSE values



(d) Detail comparison of the noisy image (left), the denoised image (middle) and the reference image (right).

Figure 3.7: Example images and results of the RDFC algorithm applied to these images. All images have been rendered using PBRT and the scenes provided by Rousselle et al. [RMZ13].

Chapter 4

Dual-Domain Image Denoising

4.1 Overview

In contrast to RDFC, the *dual-domain image denoising filter* (DDID) [KZ13, KZ15] is a more general image denoising algorithm, which is not specifically tailored for denoising rendered images. It combines a variation of the bilateral filter with local frequency domain filtering to estimate the noise at each pixel. The estimated noise can then be subtracted from the original noisy image to get a denoised result.

The used frequency domain filtering is similar to a *wavelet shrinkage*. The idea behind wavelet shrinkage algorithms is the following: First, the noisy input signal is transformed to a wavelet domain, which is essentially a transform domain, where the coefficients contain both spatial and frequency information. Then, all small wavelet coefficients are assumed to belong to the noise and are weighted down accordingly to remove the noise. The denoised signal can then be retrieved by inverting the wavelet transform. Wavelet shrinkage is based on the assumption, that the representation of the signal without noise in the transform domain is *sparse*, i.e. most coefficients should be zero. The noise can therefore be removed by either thresholding small coefficients to zero or by weighting them down using a continuous weighting function.

The motivation for combining spatial domain filtering with frequency domain filtering, is to overcome the limitations of these two filtering techniques. [KZ13] Many state-of-the-art filtering techniques, such as Block-Matching 3d denoising (BM3D) [DFKE07], use a combination of spatial and transform domain techniques to achieve high quality results. The bilateral filter has the tendency to remove too many low-contrast details, while it keeps strong edges very well. On the other hand, frequency domain filtering can introduce ringing artifacts in presence of high contrast edges. These limitations are demonstrated in Figure 4.1 on the next page. The iterative application of the combined spatial and frequency domain filtering yields a denoising algorithm which offers a similar denoising performance as other state-of-the-art methods such as BM3D, but is much easier to implement. The combined spatial and frequency domain filter is called *dual-domain filter*.

(a) Noisy image (b) Bilateral filtering (c) Frequency domain filtering (d) Reference

Figure 4.1: Comparison of the denoising quality of the bilateral filter and the used frequency domain filter. The bilateral filter loses details in the low contrast region (top row), but can filter high contrast regions fairly well (bottom row). On the other hand, the frequency domain filter introduces ringing artifacts near high contrast edges, but loses less details than the bilateral filter in low contrast regions.

In the following chapter, the explanation of the DDID filter will closely follow the latest paper on dual-domain filtering [KZ15] and focus on denoising of color images. For the sake of brevity, other interesting applications of the dual-domain filter, such as removing JPEG artifacts or denoising artifacts introduced by other denoising methods, will not be covered. The first section of this chapter describes the dual-domain filter. Afterwards, it's iterative application to denoising is described and the implementation of the filter using Halide discussed.

4.2 Dual-Domain Filtering

4.2.1 Noise estimation in the spatial domain

The first step of the dual-domain filter is to estimate the noise in the spatial domain using a modified bilateral filter. The dual-domain filter assumes an additive noise model, where pixel p of the noisy image y is given by $y_p = x_p + n_p$. Here, x_p denotes the true pixel value at position p and n_p additive Gaussian noise with zero mean. The standard bilateral filter tries to estimate the true value of pixel p as

$$\bar{x}_p = \sum_{q \in \mathcal{N}_p} y_q k_q / \sum_{q \in \mathcal{N}_p} k_q \ , \tag{4.1}$$

where k_q is a bilateral kernel $k_q = k(\|y_p - y_q\|^2, \|p - q\|^2)$ and $\mathcal{N}_p$ is the square neighborhood of pixel p. The function $k : \mathbb{R}^2 \to \mathbb{R}$ is a suitable kernel function, which usually is the product of a range kernel and a spatial kernel. It takes on only positive values and decreases monotonically in both arguments. Using the bilateral filter, an estimate of the noise n_p at pixel p can be computed by

subtracting the pixel value estimate $\bar{x}_p$ from the noisy value y_p:

$$\bar{n}_p = a(y_p - \bar{x}_p) = a \sum_{q \in \mathcal{N}_p} (y_p - y_q) k_q / \sum_{q \in \mathcal{N}_p} k_q \ . \tag{4.2}$$

The value $a \in [0, 1]$ is simply a confidence factor, where a high value implies that we trust the noise estimate.

4.2.2 Noise re-estimation in the frequency domain

Using the initial noise estimate $\bar{n}_p$ and the bilateral weights k_q, the final noise estimate is then computed using a frequency domain filter. On an abstract level, the frequency domain filter works as follows: First, the differences $y_q - y_p$ between the central pixel and all of its neighbors in $\mathcal{N}_p$ are transformed using the discrete Fourier transform. In contrast to a typical wavelet shrinkage, the dual-domain filter now tries to estimate the noise. Instead of removing the noise from the transformed signal by weighting down low amplitude coefficients, the filter weights down high amplitude coefficients to get rid of the signal. The remaining low amplitude coefficients should then represent only the noise. These can then be transformed back to the spatial domain, to get the final noise estimate at pixel p.

Before transforming the differences $y_q - y_p$ to the Fourier domain, the spatial domain noise estimate $\bar{n}_p$ is subtracted from y_p to make the difference estimate less biased by the noise at the center pixel. Additionally, the differences are multiplied by the bilateral weights k_q. This helps to suppress high contrast edges. These high contrast edges would lead to many low amplitude coefficients in the frequency domain. The assumption of a sparse representation of the signal does not hold for high contrast edges. This is essentially caused by the fact, that the continuous Fourier transform of a box function is the $\mathrm{sinc}(x) = \sin(x)/x$ function, which only slowly approaches zero as $x \to \pm\infty$ and therefore results in many small non-zero coefficients in the DFT of the box function. If a traditional wavelet shrinkage is applied to such a signal, the removal of low amplitude coefficients, of which many are due to the hard edge in the signal, causes ringing artifacts. Figure 4.2 on the following page demonstrates the ringing effect. In summary, the preprocessed differences are then

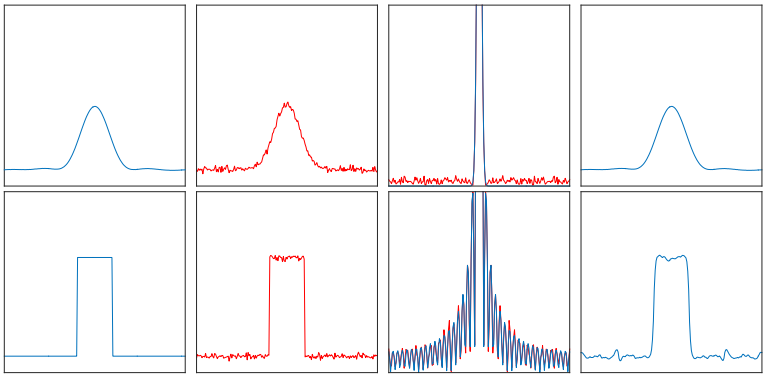
$$d_q = (y_q - (y_p - \bar{n}_p)) k_q \ . \tag{4.3}$$

The used local Fourier transform is also called *windowed Fourier transform* or *short-time Fourier transform* (STFT). Although related to a wavelet transform from a conceptual point of view, it's technically not a wavelet transform. Given a fixed radius of the local window, which is transformed using the Fourier transform, the spatial and frequency resolution of the STFT are fixed. For a wavelet transform, the resolution varies for the different frequencies of the signal.

The 2D Fourier coefficients of the modified differences are computed as

$$D_f = \sum_{q \in \mathcal{N}_p} d_q e^{-i \frac{2\pi}{2r+1} f \cdot (q-p)}, \ f \in \{-r, \ldots, r\}^2 \ . \tag{4.4}$$

Note, that f and $q - p$ are 2D vectors and the "$\cdot$" denotes the dot product. This is the standard 2D DFT combined with coordinate transform of the input



(a) Signal (b) Noisy signal (c) Power spectrum (d) Denoised

Figure 4.2: Denoising of two different input signals by weighting down coefficients in the Fourier domain. From left to right the original signal, the noisy signal, the power spectrum of the original signal (blue) and the noisy signal (red) and the denoised result. For the top example, the denoising works quite well, as the sparsity assumption holds and the small coefficients in the power spectrum of the noisy signal mostly correspond to noise. For the bottom signal, this is clearly not the case.

signal, such that the neighborhood around pixel p has indices ranging from $-r$ to r along both image axes. This does not affect the magnitude of the Fourier coefficients. Since the filtering weights in the Fourier domain rely only on the magnitude of the coefficients, the filtering does not depend on the indexing of the signal. Choosing the indices this way, also simplifies the inverse DFT afterwards. Also note, that we evaluate the Fourier coefficients D_f for all $f \in \{-r, \dots, r\}^2$. Again, this is done for convenience, since this simplifies the guided formulation in Section 4.2.3.

The resulting frequency domain coefficients are then multiplied with a frequency domain kernel, which weights down large coefficients. The weights for each frequency are computed as

$$K_f = K\left(|D_f|^2 \Big/ \sum_{q \in \mathcal{N}_p} k_q^2\right) . \tag{4.5}$$

The function $K : \mathbb{R} \to \mathbb{R}$ is monotonically decreasing and weights down large coefficients. The coefficients are normalized by the sum of squared bilateral weights. The weighted coefficients $D_f K_f$ should now represent the noise in the frequency domain. To get the final noise estimate, we need to calculate the inverse Fourier transform. Because we are only interested in the noise estimate for the central pixel, which has the spatial domain coordinates $(0, 0)$, the inverse Fourier transform reduces to computing the sum of the weighted coefficients and dividing it by the number of coefficients:

$$\hat{n}_p = A \sum_{f \in \mathcal{F}_p} D_f K_f / (2r+1)^2 . \tag{4.6}$$

$\mathcal{F}_p$ is the set of frequencies and $A \in [0,1]$ is another confidence factor. The output of the dual-domain filter in pixel p is then defined as $\mathrm{DDF}_p(y,a,A,k,K) = \hat{n}_p$, where a and A are the confidence factors and k and K the kernel functions. The final estimate of the true image is then $\hat{x}_p = y_p - \mathrm{DDF}_p(y,a,A,k,K)$.

4.2.3 Formulation as a guided filter

The algorithm can be extended to not only take the noisy image as an argument, but also an additional guide image. The filter weights are then computed using the guide image and are independent from the noisy image. Since the input and the guide image are both real valued, but the DFT can transform complex valued inputs, the filter is naturally formulated as a guided filter by substituting y with $z = g + iy$ in the previous equations, where g is the guide image, y the noisy input image and i the imaginary unit.

The computation of weights in the spatial and frequency domain is then adjusted to consider only the guide image:

$$k_q = k(\|\mathrm{Re}(z_p - z_q)\|^2, \|p-q\|^2) \tag{4.7}$$

$$K_f = K\left(\left|\frac{D_f + D^*_{-f}}{2}\right|^2 \Big/ \sum_{q\in\mathcal{N}_p} k_q^2\right) \tag{4.8}$$

While taking the real part of the differences as the argument for the bilateral kernel function is trivial, the term $\frac{1}{2}(D_f + D^*_{-f})$ for the frequency weight computation is slightly more complicated. It calculates the Fourier coefficients of $(g_q - g_p - \mathrm{Re}(\bar{n}_p))k_q$ from the Fourier coefficients of the complex signal $(z_q - z_p - \bar{n}_p)k_q$. This formula can be derived directly from the definition of the discrete Fourier transform: Let $z = x + iy \in \mathbb{C}^N, N \in \mathbb{N}$ be any complex signal. The Fourier coefficients of z can then be written as:

$$Z_k = \sum_{n=0}^{N-1} z_n e^{-2\pi ikn/N} = \sum_{n=0}^{N-1} x_n e^{-2\pi ikn/N} + i\sum_{n=0}^{N-1} y_n e^{-2\pi ikn/N} \ .$$

Using the basic properties of complex conjugation, we get the equality

$$Z^*_{-k} = \sum_{n=0}^{N-1} x_n e^{-2\pi ikn/N} - i\sum_{n=0}^{N-1} y_n e^{-2\pi ikn/N} \ .$$

Therefore it holds, that the values $\frac{1}{2}(Z_k + Z^*_{-k})$ are exactly the Fourier coefficients of the real part of the signal z. The terms D_{-f} do not require any additional calculations, as they already have been calculated. Using a guide image, the final noise estimate is computed as $\hat{n}_p = \mathrm{Im}(\mathrm{DDF}_p(g+iy,a,A,k,K))$.

4.3 The DDID algorithm

The previously described dual-domain filtering algorithm is applied iteratively to get a state-of-the-art denoising result. For the denoising filter, the weighting functions in the dual-domain filter are defined as follows:

$$k_n(x,y) = \cos\left(\min\left(\frac{\pi}{2}, \sqrt{\frac{x}{T_n n}}\right)\right)^n e^{-\frac{y}{S_n}} \ , \tag{4.9}$$

$$K_n(x) = \cos\left(\min\left(\frac{\pi}{2}, \sqrt{\frac{x}{Vn}}\right)\right)^n \ . \tag{4.10}$$

The weighting functions in the spatial and in the frequency domain both depend on the iteration variable n, which counts down from the number of iterations N to 1. The weighting function for the bilateral filter is the product of a clamped cosine, which accounts for the difference in value, and a Gaussian kernel, which weights down pixels with a large spatial distance to the center pixel. T_n and S_n are scaling factors, which depend on the iteration variable. They are defined as

$$S_n = 2\sigma_s^2 \alpha^{\frac{1-n}{2N}} \ ,$$

$$T_n = \gamma_r \sigma^2 \alpha^{\frac{n-1}{N}} \ ,$$

where σ^2 is the noise variance and γ_r, α and σ_s are positive constants. The windows considered by the bilateral filter gets larger in every iteration, since decreasing n increases the scaling factor S_n. On the other hand, the filtering becomes more strict, because T_n decreases and therefore scales up the color differences. The value V, which is used to scale the Fourier coefficient magnitude, is constant over all iterations and depends only on the noise variance σ^2 and a constant scaling factor γ_f:

$$V = \gamma_f \sigma^2 \ .$$

In both weighting functions, the cosine gets raised to the power n and the argument divided by n. This has the effect, that the falloff becomes steeper, as n decreases. While the other parameters only scale the weighting functions, raising to a power changes the overall shape. In the beginning, the weighting functions resemble Gaussians. As n decreases however, the functions reject outliers more aggressively. The division of x by n makes sure, that raising the cosine to a power does not influence the width of the falloff functions too much. Additionally, the confidence factors a and A are initially set to a quite small value and then increase in every iteration:

$$a_n = A_n = \cos\left(\frac{n-1}{N}\frac{\pi}{2}\right) \ .$$

Furthermore, the radius of the neighborhood considered by the filter is adapted to the spatial filter width by setting it to:

$$r_n = \max(4, \mathrm{round}(2\sqrt{S_n/2})) \ .$$

A visualization of how different parameters change over the iterations is given in Figure 4.3 on the next page.

The denoising filter uses the guided formulation of the dual-domain filter. In the first iteration, the noisy image is also taken as guide image. After computing the dual-domain filter in the first iteration, the guide image is then set to the denoised image. The noisy image remains unchanged. This is then repeated in every iteration.

To improve denoising quality, the color input image is first transformed by applying a discrete cosine transform (DCT) along the color axis. The discrete cosine transform is related to the Fourier transform, but uses only real numbers

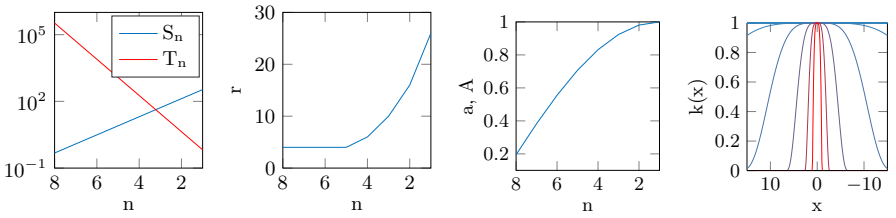


Figure 4.3: Change of DDID parameters over iterations as n decreases from N to 1. The number of iterations N is set to 8. In plot of the spatial kernel $\mathrm{k_n}(x)$, the line colors go from blue to red as the iteration variable n decreases. Figure based on the parameter visualization in [KZ15].

and represents the signal as a sum of cosine functions. The variant of the DCT used for the DDID algorithm is defined as

$$y_k = w_k \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi}{2N}(2n+1)k\right), k = 0, 2, \ldots, N-1, \tag{4.11}$$

where y_k are the output coefficients, x_k the coefficients of the input signal and

$$w_k = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } k = 0 \\ \sqrt{\frac{2}{N}} & \text{if } 1 \leq k \leq N-1 \ . \end{cases} \tag{4.12}$$

The transform along the color axis corresponds to a change of color space and is implemented as the multiplication of the three-dimensional color vector at each pixel by a three-by-three DCT matrix M. The inverse DCT can easily be calculated by exploiting that M is orthogonal and therefore the inverse equals the transpose. This is a desirable property, since an orthogonal transform does not change the variance of the noisy input. [KZ13] In summary, the DDID algorithm is described by the following pseudocode:

DUALDOMAINIMAGEDENOISING(y, σ^2)

// Define constants
$N = 8$
$\sigma_s = 13$
$\gamma_r = 5.3/N$
$\gamma_f = 13/N$
$\alpha = e^{15}$
$V = \gamma_f \sigma^2$
// Apply a DCT transform along the color channel
$y = \mathrm{dct}_c(y)$
// Initially, the noisy image is also used as guide image
$x = (1 + i) \cdot y$
for $n = N$ **to** 1
    $S_n = 2\sigma_s^2 \alpha^{\frac{1-n}{2N}}$
    $T_n = \gamma_r \sigma^2 \alpha^{\frac{n-1}{N}}$s
    $k_n = [x, y] \cos\left(\min\left(\frac{\pi}{2}, \sqrt{\frac{x}{T_n n}}\right)\right)^n e^{-\frac{y}{S_n}}$
    $K_n = [x] \cos\left(\min\left(\frac{\pi}{2}, \sqrt{\frac{x}{V n}}\right)\right)^n$
    $r_n = \max(4, \mathrm{round}(2\sqrt{S_n/2}))$
    $a_n = A_n = \cos\left(\frac{n-1}{N}\frac{\pi}{2}\right)$
    // Update the guide image
    $x = (1 + i) \cdot y - \mathrm{Im}(\mathrm{DDF}(x, a, A, k_n, K_n))$

// Transform the result back to the original color space
$x = \mathrm{dct}_c^{-1}(\mathrm{Re}(x))$
return x

The inputs to the algorithm are the noisy image y and noise variance σ^2. The variance is assumed to be known, estimated by another technique or manually specified. The function $\mathrm{dct_c}$ is the DCT transform along the color dimension and k_n and K_n are functions of the arguments listed within the square brackets.

4.4 Implementation using Halide and results

Even though the DDID algorithm is less complex than RDFC, the implementation proved to be more difficult than expected. The implementation of the spatial domain bilateral filter is straightforward. Similar to RDFC, the bilateral filter has been implemented naively. The performance bottleneck is clearly the frequency domain filtering step, as the Fourier transform needs to be evaluated at each pixel on windows of up to 53×53 pixel.

With most programming languages, computing a Fourier transform is simply a matter of using a good FFT library, such as the *Fastest Fourier Transform in the West* (FFTW) [FJ05] library. Using Halide however, as of writing this thesis, external C functions can only be called if the Halide code is scheduled to run on the CPU. Since we were mostly interested in using Halide to target the GPU, external libraries could not be used.[1] There exists a basic FFT implementation written in Halide, which is part of the Halide code examples and

[1] There seem to exist workarounds for this problem, but they are not very clean and could easily break in newer Halide versions.

is based on the work of Govindaraju et al.[GLD+08]. Although the Halide FFT implementation gives reasonable performance for standard use cases, it does not effectively handle the special case when the input size is prime or contains large prime factors. In these cases, traditional FFT algorithms, which rely on factoring the problem size into prime factors, offer no significant performance improvement over a naive DFT implementation.

Performing a FFT on prime sized inputs can be accelerated in multiple ways. Two well-known algorithms for this problem are Rader's algorithm [Rad68] and Bluestein's algorithm [Blu70]. Both algorithms reformulate the problem of computing the DFT of a prime sized input such that it can be solved by computing the FFT of a problem of a different size. This can then be done by using an existing FFT implementation, as the new problem size usually is composite or even a power of two. The key idea of both algorithms is to use the basic properties of the complex exponential terms in the DFT matrix.

Bluestein's algorithm works by expressing the DFT of length N as a convolution of two sequences of length $N-1$. This convolution can be computed efficiently by using the convolution theorem and an existing FFT implementation. The sequences used in Bluestein's algorithm need to be zero padded to a length of at least $2N-1$ to compute the convolution. [Smi07] Because we need to compute the local 2D Fourier transform in two passes, the DDID algorithm needs a lot of memory to store the intermediate result. The local neighborhood of each pixel has to be stored at the same time to allow for parallelization over pixels. Therefore, Bluestein's algorithm is less suited for the DDID algorithm, as it would increase memory usage.

Rader's algorithm also reformulates the DFT as a convolution of two sequences, but in a way that does not require zero padding. Thus, we tried applying it to accelerate the existing Halide FFT implementation. The technical details of Rader's algorithm are explained in Appendix A. We implemented the algorithm on top of the existing Halide FFT implementation in a non-recursive fashion and without zero padding.

Unfortunately, the implementation of Rader's algorithm was slower than the naive DFT for our problem sizes. Rader's algorithm, similar to Bluestein's algorithm, tries to decrease the complexity of the Fourier transform in the length of the input N, but introduces larger constant factors to the computation. For a black and white image of size 53×53, our naive DFT implementation took 8.15×10^{-4} s and Rader's algorithm 1.368×10^{-3} s. The unmodified Halide FFT, which falls back to the naive DFT for prime sized inputs, took 1.288×10^{-3} s. All times were recorded by executing the respective algorithms 25 times on a single thread on an Intel E5645 processor at 2.67 GHz. For Rader's algorithm, the DFT coefficients, all required permutations and the Fourier transform of the sequence, which is independent of the signal, were precomputed at compile time. To put the speed of the different FFT implementations in perspective: Matlab's fft2 method took only about 2.55×10^{-4} s on the same image. The overhead of the Halide FFT was too big to achieve any performance improvements. The final implementation of the DDID algorithm on the GPU therefore uses a naive DFT implementation.

Aside from the Fourier transform, the implementation of the DDID algorithm on the GPU is straightforward. As already mentioned, memory usage was an issue, since the Fourier transform needs to be computed in two steps. For radii smaller than 6, it is possible to store the local neighborhood of each

Image size	**256 × 256**		**512 × 512**	
	B/W	Color	B/W	Color
Matlab	26.0	46.9	90.9	180.0
Halide (GPU)	2.0	4.6	7.0	16.7
Halide (CPU)	45.7	393.8	210.8	1779.6

Table 4.1: Execution times in seconds for different image sizes. Times are averaged over three runs of the DDID filter. The CPU code was executed on two Intel E5645 processors with a total of 12 cores at 2.67 GHz and the GPU code on a Nvidia GTX Titan.

pixel at the same time in one global memory buffer (at least for images of size up to one mega-pixel). Note that using shared memory was not an option for this, as the amount of shared memory is too limited. For larger radii, the computation is scheduled to run in tiles, where the tiles are processed serially and the computation within each tile is parallelized on a per pixel level. As already has been noted by Knaus and Zwicker [KZ13], no sliding window optimization, as would typically be used to compute the standard short-time Fourier transform, is possible in the DDID algorithm.

For a summary of the running time of the algorithm for different image sizes and comparison with the existing Matlab implementation [KZ15], see Table 4.1.

The Halide CPU implementation is quite slow, as the naive DFT is not nearly as fast as the FFT implementation used by Matlab. This demonstrates, that the existing Matlab code is much faster than a naive parallel CPU implementation. The main focus was however the GPU implementation, which is about 10 times faster than the Matlab code. All times are heavily dominated by the execution time of the last iteration, where the window radius is 53×53. For the GPU implementation, the last iteration amounts for about 65% of the total execution time. For the CPU implementation, the fraction increases to about 78%. A few examples of the application of the filter are shown in Figure 4.4 on the next page.

(a) Original images

(b) Noisy images

3.1×10^{-3} 1.7×10^{-3} 1.8×10^{-3}

(c) Dual-domain image denoising results and the respective RMSE values

(d) Detail comparison of the noisy image (left), the denoised image (middle) and the original image (right).

Figure 4.4: Example images and results of the dual-domain denoising algorithm applied to these images. All images have been corrupted by zero-mean Gaussian noise with variance 0.002. The same variance was then used as a parameter for the DDID algorithm. Despite the spatial domain filtering step, some slight ringing artifacts still occur near sharp edges.

Chapter 5

Denoising for Gradient-Domain Path Tracing

5.1 Overview

As already discussed, one fundamental problem of path tracing algorithms is their slow convergence rate. While filtering algorithms such as RDFC can be very useful for a lot of applications, they cannot accelerate the underlying rendering algorithms. Gradient-domain rendering algorithms have been introduced recently as a promising approach to speed up unbiased rendering.

Gradient-domain rendering was originally introduced as *gradient-domain Metropolis light transport* (G-MLT) [LKL+13, MRK+14]. *Metropolis light transport* (MLT) [Vea98] is a rendering algorithm, which tries to reduce rendering time by distributing the samples in path space using a probability density which is approximately proportional to the contribution each path makes to the final image. The gradient-domain Metropolis light transport algorithm improves on the standard MLT algorithm by also sampling image space gradients of the output image, i.e. differences between the values of (neighboring) pixel. The motivation behind gradient-domain rendering algorithms is the observation, that the gradients in a natural image are much sparser than the image itself. There are often large areas, where the image gradients are almost zero. The G-MLT algorithm simultaneously samples a coarse base image and horizontal and vertical image gradients. The sampled gradient information is then used to distribute more samples in regions where the image has a large gradient. The idea of G-MLT is to distribute samples where the image changes, and not where it's just bright.

The final image is then reconstructed by taking into account the sampled base image as well as the sampled gradients. The reconstructed image should be close to the base image and its gradients close to the sampled gradients. This optimization problem amounts to solving a *screened Poisson equation* [BCCZ08]. This can be done efficiently on the GPU. The reconstructed image is then an unbiased estimate of the true image. An example of the reconstruction and the

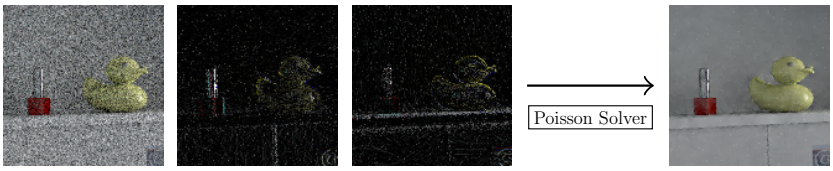


Figure 5.1: The base image and the gradients in both horizontal and vertical direction are combined using a Poisson solver to get the result on the right.

input is shown in Figure 5.1. Note, how the gradient images contain mostly zero or near-zero values.

In practice, the MLT algorithm can often give very non-uniform convergence in image space. Some regions in the final image may converge very quickly, while others remain too dark or too bright for a very long time. The noise is much less uniform than in naive path tracing. The G-MLT algorithm inherits these convergence issues.

However, it turns out that sampling gradient information can also benefit conventional path tracing. The gradients can be sampled with less variance than the pixel values and therefore reduce the noise in the reconstructed final image by a significant factor. The introduction of both gradient-domain path tracing (G-PT) [KMA+15] and gradient-domain bidirectional path tracing (G-BDPT) [MKA+15] proved that sampling gradients can significantly speed up Monte Carlo rendering, even without concentrating the samples on regions of change in the image. Like G-MLT, both these techniques can be used to produce unbiased images.

Since gradient-domain rendering algorithms give a considerable improvement in the unbiased setting, it is interesting to investigate whether they can also be used effectively in combination with a biased filtering technique. For many applications, rendering until full convergence is not a viable option and using a suitable filter is key to get a usable result from a path tracer. Filtering the results from gradient-domain path tracing is non-trivial, since the noise characteristics are different from the noise characteristics of conventional path tracing.

We propose a novel filtering algorithm, which extends the Poisson reconstruction of the final image to achieve a denoising effect. We do this by constraining the final image to be locally close to the subspace spanned by local feature patches. In other words, we want each small patch of the reconstructed image to be close to a weighted sum of small feature patches.

Similar to RDFC, we compute multiple candidate filters using different parameters. We then estimate the error of each candidate filter using the simple and general error estimation technique proposed by Bauszat et al. [BEEM15]. The idea of this technique is to sparsely sample reference pixels, which are used to compute a sparse error estimate for each candidate filter. The error estimates can then be interpolated and used to compute a filter selection map.

In the first part of this chapter, a short explanation of the technical details of gradient-domain path tracing is given. This will mostly focus on the Poisson reconstruction, since our filter extends the standard reconstruction. After that

our new filtering technique is explained and evaluated.

5.2 Gradient-domain path tracing

Gradient-domain rendering relies on sampling finite difference gradients between neighboring pixels. Instead of only sampling pixel values I_i, also differences $\Delta_{ij} = I_i - I_j$ between neighboring pixels i and j are sampled. Like conventional path tracing, G-PT samples a fixed number of paths going through each pixel. These paths are called *base paths* and are accumulated in the base image. From each base path, four *offset paths* going through the four directly neighboring pixels are generated. The mapping which is used to generate the individual offset paths from the base path is called *shift mapping*. The shift mapping is deterministic and tries to map the base path to an offset path with similar throughput. This means, that each offset path should carry more or less the same amount of light as the base path. The difference in throughput between the base and an offset path should ideally be very small, in order to minimize the variance of the sampled gradient. The four sampled differences are then accumulated into gradient images. A separate gradient image is stored for each of the four shifting directions.

Conceptually, the described sampling scheme amounts to merging the two path integrals I_i and I_j under one integration sign and then Monte Carlo integrating the resulting integral. The shift mapping reduces variance, since the offset path is highly correlated to the base path, instead of randomly generated from scratch. In general, similar tricks can be used to reduce the variance of the Monte Carlo estimate of differences or sums of arbitrary Monte Carlo integrals. Reusing the already generated random numbers to compute a correlated sample is known as *common random numbers* variance reduction.

A good strategy to find offset paths with a similar throughput as the base path, is to try to reconnect the offset path to the existing base path as soon as possible. In order to to this, the shift mapping has to make a distinction between the specular and diffuse vertices of a path. A material is classified specular, if incoming light from a fixed direction can only be refracted or reflected in a discrete set of directions. An illustration of the shift mapping used in G-PT is given in Figure 5.2. It works as follows: First, a ray is traced through the offset pixel and through the chain of following specular interactions, until a diffuse vertex is found. In the illustration, this means that the offset ray is traced through the glass plate and then hits the diffuse ground. The chain of specular vertices consists of vertices x_1 and x_2 in the base path and $\tilde{x}_1$ and $\tilde{x}_2$ in the offset path. Since the diffuse vertex on the ground is followed by a specular vertex in the base path, we cannot yet reconnect to the base path. If we would connect $\tilde{x}_3$ directly to vertex x_4 of the base path, the resulting path would carry no light. Instead, the path is continued from vertex $\tilde{x}_3$ such that the half-vector at this vertex projected onto the tangent plane matches the projected half-vector at vertex x_3 of the base path. The offset path is traced through the following specular chain and can then be reconnected in our example, since the base path has two consecutive diffuse vertices. If there was only one diffuse vertex, we would again match the half-vector and trace the next specular chain and so on.

G-MLT and G-BDPT use slightly different shift mappings and rely on the *manifold walk* [JM12] algorithm. This algorithm allows to determine an outgo-

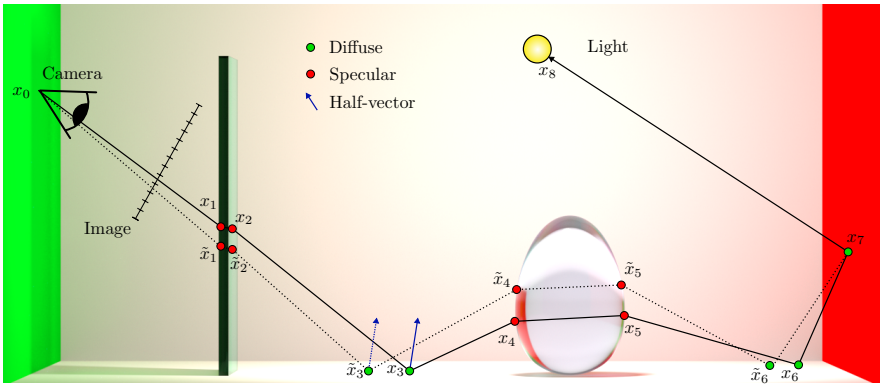


Figure 5.2: The base path (solid line) is mapped to the offset path (dashed line).

ing direction at a diffuse vertex, in our example vertex $\tilde{x}_3$, such that the offset path directly reconnects with the base path after the following specular chain (the offset path could connect at vertex x_6 in the example). This means, that the shift mappings used in G-MLT and G-BDPT do not require two consecutive diffuse vertices to reconnect to the base path. They can thus give slightly better offset paths, at the cost of requiring more time to compute the shift mapping.

Gradient-domain rendering works very well, if the shift mapping is successful and the offset path has nearly the same throughput as the base path. The gradients in natural images are also typically sparse and have overall less energy than the color image. [LKL+13, KMA+15] We can therefore expect the true difference between two neighboring pixels to have a small magnitude most of the times. The variance in Monte Carlo integration is proportional to the magnitude of the integrand and thus the finite differences can be sampled with very low variance compared to the image itself. A complete analysis of the variance reduction achieved using gradient-domain rendering is given by Kettunen et al. [KMA+15].

Problematic are pixels, where the shift mapping fails to find a similar path. This can for example happen on edges of objects, where a small shift in screen space can drastically change the throughput of the resulting path compared to the throughput of the base path. Since the path is always shifted by one pixel, the quality of the sampled gradients also depends on the image resolution. If the image resolution is low, the shift mapping will typically fail more often, since the shift covers a greater distance in world space. The shift mapping is also more likely to fail if there are many specular interactions involved.

There are a variety of technical difficulties involved in the sampling process, which go beyond the scope of this short overview and will not be covered here. The shift mapping is for example not necessarily bijective. This is one reason why all four gradient directions are sampled instead of just one vertical and one horizontal direction, which would in theory contain the same information. Also merging the two integrals I_i and I_j under one integration sign requires a change of variables in one of the integrals, which means that the integrand, i.e. the throughput, has to be multiplied by the Jacobian determinant corresponding to this change of variables.

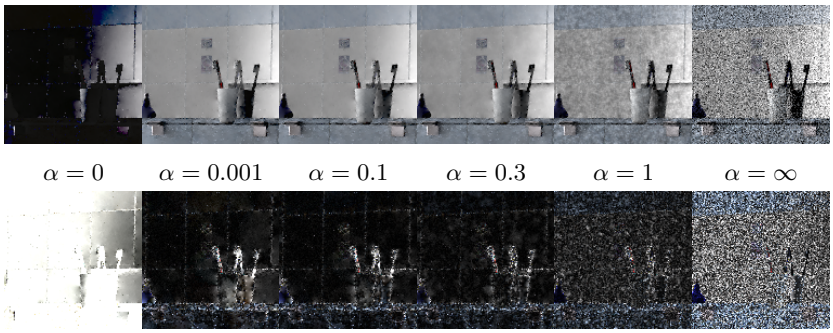


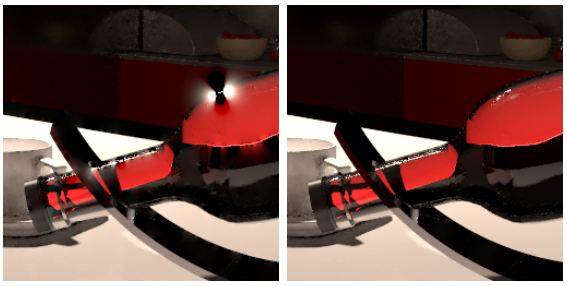
Figure 5.3: Comparison of the L2-reconstruction using different values for α. The top row of images is the reconstruction result and the bottom row the per-pixel error. The quality of the reconstruction is measured by the RMSE, denoted below each error map.

After the sampling process is completed, the final image is reconstructed by solving a Poisson equation. The four sampled gradient images are first combined into just two images, because each gradient was sampled in both directions. For two neighboring pixels i and j, the differences $\Delta_{ij} = I_i - I_j$ and $\Delta_{ji} = I_j - I_i$ have been sampled. They both estimate the same difference between neighboring pixels, just with a different sign. They can be merged into one value by reversing the sign of the second gradient and then computing the average. The input to the reconstruction is then the sampled base image and the sampled horizontal and vertical gradients. The final image is then reconstructed as the solution of the Poisson problem

$$I = \arg\min_{\hat{I}} \left\| \alpha(\hat{I} - I^b) \right\|_2^2 + \left\| \begin{pmatrix} H^{dx}\hat{I} \\ H^{dy}\hat{I} \end{pmatrix} - \begin{pmatrix} I^{dx} \\ I^{dy} \end{pmatrix} \right\|_2^2 , \tag{5.1}$$

where I^b denotes the base image and I^{dx} and I^{dy} the sampled horizontal and vertical gradients. The images are all interpreted as $n \times 3$, matrices, where n is the number of pixels and each row represents the RGB values of one pixel. The L2-norm is calculated as if the $n \times 3$ matrix was a vector of length $3n$ and is simply the root of the sum of squared matrix components. The matrices H^{dx} and H^{dy} compute the finite difference gradients of an image in horizontal and vertical direction respectively. The parameter α controls the importance of the base image. If it is set too high, the reconstructed image will contain a lot of noise from the base image. If it is set too low, the reconstructed image can suffer from color shifting. The effect of changing α is demonstrated in Figure 5.3. If α equals zero, the base image is not used at all and the correct brightness cannot be determined for the reconstruction. The gradients do not contain any information about the absolute brightness. Increasing α weights the information from the base image more and reduces color shift. Setting α to infinity removes the effect of the gradients entirely. For the given sampled data in Figure 5.3, a value of 0.3 is the optimal α in terms of the RMSE.

The optimization problem above is just a least squares problem and can be



(a) L2-norm (b) L1-norm

Figure 5.4: Using the L2-norm for the reconstruction, gradient outliers can cause visibly distracting artifacts. These artifacts are removed by using the L1-norm.

solved using the normal equations. The Poisson problem can be written as

$$I = \arg\min_{\hat{I}} \left\| \underbrace{\begin{pmatrix} \alpha I_n \\ H^{dx} \\ H^{dy} \end{pmatrix}}_{A} \hat{I} - \underbrace{\begin{pmatrix} \alpha I^b \\ I^{dx} \\ I^{dy} \end{pmatrix}}_{b} \right\|_2^2, \tag{5.2}$$

where I_n denotes the identity matrix of size n. The unique solution I is then given as

$$I = (A^T A)^{-1} A^T b \tag{5.3}$$

The matrix A is very sparse and the solution can thus be computed efficiently. It can be shown, that the reconstructed image I is an unbiased estimate for the true image. [LKL+13]

In practice it is often beneficial, to replace the squared L2-norm in the optimization problem by the L1-norm. The L1 optimization problem can be solved using the *iteratively reweighted least squares* (IRLS) algorithm. The iterative optimization works by solving the least squares problem repeatedly and weighting down constraints, which lead to a large error in the reconstruction. The sampled gradients can have non-zero curl, which means that there is no image that simultaneously satisfies all gradient constraints. If the curl is large, the conflicting gradient information can cause very distracting errors in the reconstruction. These kind of errors are mostly fixed by resorting to the L1-norm, see also Figure 5.4. An extreme example of conflicting gradient information is here at top of the bottle, where the light source is directly reflected in the glass material. Since the reflection of the light on the bottle only covers a few pixels, the offset paths will almost never be able to reconnect to the base path. The shift mapping thus fails very often and the variance of the sampled gradients is high. Reconstructing under the L1-norm removes these kind of artifacts entirely in this example. Note, that the reconstruction using the L1-norm is a non-linear operation and introduces some bias to the result. Most notably, images reconstructed using the L1-norm can be slightly too dark. The L1-optimization is solved using the following IRLS algorithm:

SolveL1($A, b, numIter$)

```
1  W = I_{A.rows}
2  for i = 1 to numIter
3       // Solve weighted least squares problem min_x ||W(Ax − b)||_2^2
4       x = (A^T W^2 A)^{-1} A^T W^2 b
5       // Compute weights W
6       e = Ax − b
7       for j = 1 to A.rows
8            W_jj = 1/(||e_j||_2 + regInit · regIter^{i−1})
9       W = W · A.rows/ ||W||_1
10 return x
```

where A is the constraint matrix and b is right-hand side of the overdetermined equation system. The number of iterations $numIter$ of the algorithm has to be set by the user, a default of 7 iterations works well most of the time. The scalar $\|e_j\|_2$ is the norm of row j of the matrix e and the value $\|W\|_1$ is the sum of all entries in the matrix W. The values $regInit$ and $regIter$ are constants, set to 0.05 and 0.5 respectively.

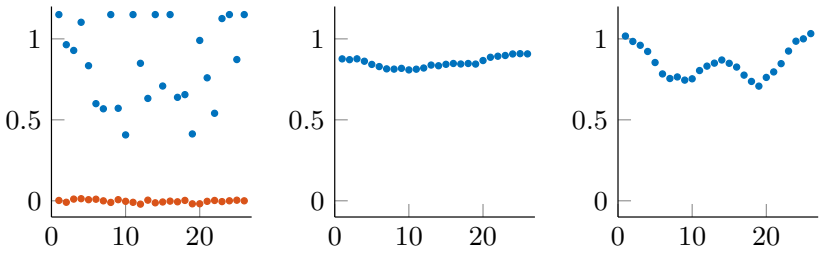
5.3 Denoising using subspace projections

5.3.1 Extending the Poisson problem by patch constraints

We extend the original Poisson problem by additional constraints to get a denoising effect. The new constraints can be motivated by first having a look at an alternative idea. One way to extend the Poisson reconstruction by denoising, is to solve the optimization under a sparsity constraint, similar to what is done in denoising techniques using *overcomplete dictionaries* [EA06]. The idea behind these denoising algorithms is similar to the idea behind wavelet shrinkage. The assumption is that the noise-free image has a sparse representation in a certain basis. Wavelet shrinkage algorithms work by transforming the noisy image to a wavelet basis and then weighting down small basis coefficients and transforming back to the standard basis. Dictionary based methods on the other hand use an overcomplete system of vectors called dictionary. A vector in the dictionary is called dictionary entry or atom. The idea is to represent the denoised image as a sparse linear combination of dictionary entries. One advantage of dictionary techniques over wavelet shrinkage is that the dictionary can have any number of entries, and usually has a larger number of entries than dimensions. The entries can be arbitrary vectors and do not need to be orthogonal or normalized. A common approach is to learn dictionary entries from a set of noise-free images. We tried several variants of explicitly constructing a dictionary from the features, such that the reconstructed image is locally the sum of small feature patches. The Poisson optimization problem is then modified to solve for a sparse coefficient vector c as

$$c = \arg\min_{\hat{c}} \left\|\alpha(S\hat{c} - I^b)\right\|_2^2 + \left\| \begin{pmatrix} H^{dx} \\ H^{dy} \end{pmatrix} S\hat{c} - \begin{pmatrix} I^{dx} \\ I^{dy} \end{pmatrix} \right\|_2^2, \text{ subject to } \|\hat{c}\|_1 < \lambda \ , \tag{5.4}$$

where S is the dictionary matrix. The constraint on the L1-norm of the coefficients enforces sparsity. The constant λ controls the sparsity of the coefficient



(a) Base image (blue) and gradients (red) (b) Global optimization (c) Local optimization

Figure 5.5: Comparison of the reconstruction using global optimization and using local optimization (both using the L2-norm) for a 1D image. The local reconstruction has been computed by solving the reconstruction on a patch of radius 2 around each pixel and then averaging the local solution patches.

vector. The final image can then be computed as $I = Sc$. The least squares problem under an L1-norm constraint on the solution is called *Lasso* problem. Even though there exist efficient solvers for this problem [VF08], this denoising scheme does not work very well for gradient-domain rendering. Traditional dictionary based denoising methods run constrained least squares optimization methods on small image patches and then combine the local patches into one result by averaging. To get the full benefit out of the sampled gradients, it is necessary to solve the Poisson optimization problem for the whole image, or very large regions, at once. Solving only locally cannot effectively remove low frequency noise from the base image. This effect is demonstrated on a 1D example in Figure 5.5. Furthermore, directly representing the reconstructed image as a sum of features often filters out image content not present in the features.

A much better approach is to constrain the reconstructed image to be locally close to a sum of feature patches – and not necessarily equal. A visualization of this is given in Figure 5.6. We define the local feature subspace F_p to be the subspace spanned by the feature patches around pixel p. The reconstructed image should be locally close to this subspace. We achieve this by extending the Poisson problem by a *patch constraint term* as

$$I = \arg\min_{\hat{I}} \left\| \alpha(\hat{I} - I^b) \right\|_2^2 + \left\| \begin{pmatrix} H^{dx} \\ H^{dy} \end{pmatrix} \hat{I} - \begin{pmatrix} I^{dx} \\ I^{dy} \end{pmatrix} \right\|_2^2 + \sum_{p=1}^{n} P_p(\hat{I}) \ , \tag{5.5}$$

where p iterates over all pixels and $P_p(\hat{I})$ is defined as

$$P_p(\hat{I}) = \left\| D_p \cdot (B_p B_p^T - I_s) \Gamma_p \hat{I} \right\|_2^2 \ . \tag{5.6}$$

The matrix $\Gamma_p \in \mathbb{R}^{s \times n}$ selects the pixels in the local neighborhood $\mathcal{N}_p$ of radius r of pixel p and s denotes the number of pixels in such a patch, i.e. $s = (2r+1)^2$. The whole unknown image $\hat{I}$ is mapped onto a vector containing only the pixels of the patch $\mathcal{N}_p$. The matrix $D_p \in \mathbb{R}^{s \times s}$ is a diagonal weighting matrix, weighting the patch constraints at each pixel. B_p is a $s \times m$ matrix, where m denotes the dimension of the local feature subspace. The columns of

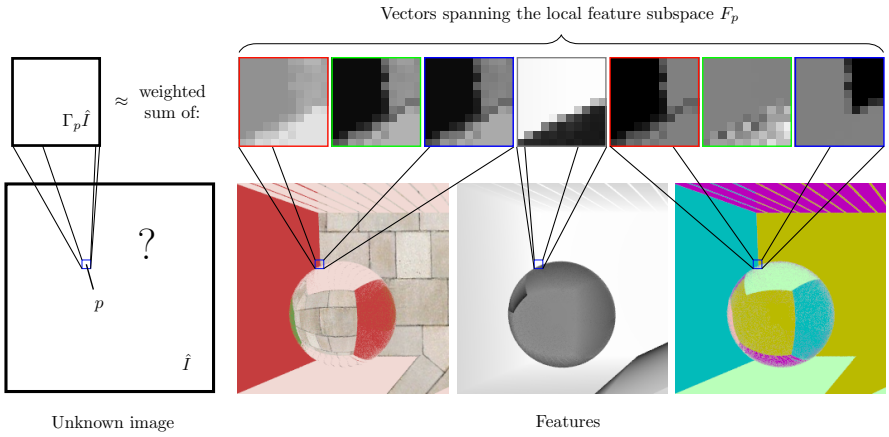


Figure 5.6: Visualization of the patch constraint for pixel p.

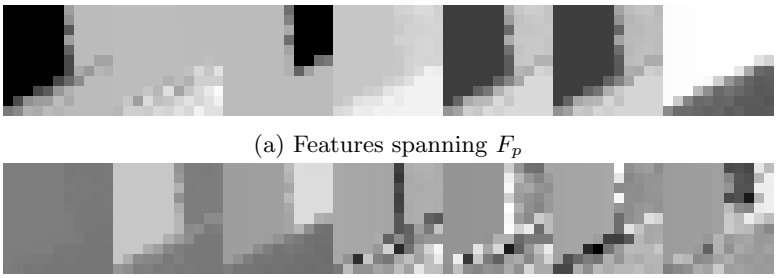
B_p are basis vectors of the local feature subspace. The projection of the local image patch onto the feature subspace is given by definition as

$$\text{proj}_{F_p}(\Gamma_p \hat{I}) = \sum_{i=1}^{m} \langle b_{pi}, \Gamma_p \hat{I} \rangle b_{pi} = B_p B_p^T \Gamma_p \hat{I} \ , \tag{5.7}$$

where b_{pi} is the i'th column of the matrix B_p and $\langle \cdot, \cdot \rangle$ denotes the dot product. The projection of a vector onto a subspace is exactly the vector in the subspace closest to the original vector. By measuring the distance of the projected vector $B_p B_p^T \Gamma_p \hat{I}$ to the original vector $\Gamma_p \hat{I}$ one can thus measure how close the vector is to the subspace. For any vector contained in the subspace, the distance will be zero.

The matrix B_p is computed using a singular value decomposition (SVD). The SVD factors any matrix $M \in \mathbb{R}^{m \times n}$ into a product of the form USV^T, where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are both orthogonal matrices and $S \in \mathbb{R}^{m \times n}$ is a rectangular diagonal matrix containing the singular values. The first n columns of U then form a basis of the range of M (assuming that the columns of M are linearly independent). We use the SVD to compute an orthogonal basis of the feature subspace. Similar to Moon et al. [MCY14] we use a *truncated SVD* to remove noise from the feature subspace. The vectors in U are ordered by the magnitude of the corresponding singular value. The smaller the singular value gets, the more noise is captured in the corresponding singular vector. This is demonstrated on an example in Figure 5.7 on the next page. By discarding singular vectors with small singular values, one can remove noise from the subspace. In practice, we just discard the singular vector with the smallest singular value. Moon et al. [MCY14] use a more sophisticated scheme to discard basis vectors, but it seems that the number of discarded vectors does not affect the result that much in our technique. Other filtering parameters have a bigger influence on the filter result.

The matrix $D_p \in \mathbb{R}^{s \times s}$ is a diagonal matrix, which weights the patch constraints for each pixel. The weights are crucial to control the importance of the patch constraints. The detailed weight computation is explained in Sec-



(a) Features spanning F_p

0.4312 0.1472 0.0794 0.0340 0.0307 0.0139 0.0037

(b) Orthogonalized features and their corresponding singular values

Figure 5.7: The feature vectors for a small patch and the corresponding orthogonal basis computed by the SVD. The smaller the singular values get, the more noise is captured in the corresponding singular vector.

tion 5.3.2. Even when extended by patch constraints, the optimization is still a least squares problem. The optimization problem can be written as

$$I = \arg\min_{\hat{I}} \left\| \underbrace{\begin{pmatrix} \alpha I_n \\ H^{dx} \\ H^{dy} \\ D_1(B_1 B_1^T - I_s)\Gamma_1 \\ D_2(B_2 B_2^T - I_s)\Gamma_2 \\ \vdots \\ D_n(B_n B_n^T - I_s)\Gamma_n \end{pmatrix}}_{A} \hat{I} - \underbrace{\begin{pmatrix} \alpha I^b \\ I^{dx} \\ I^{dy} \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}}_{b} \right\|_2^2 . \tag{5.8}$$

The constraint matrix A is sparse, but also very large. The matrix has $(3 + s)n$ rows and n columns. Even though it is very sparse, the number of non zero entries is roughly $(5 + s^2)n$, since a $s \times s$ matrix has to be stored for each pixel. This slightly overestimates the true number of non-zero coefficients, because this number does not account for the special treatment of boundary pixels. The number of non-zero coefficients grows proportionally to r^4. We therefore use a radius r of only 2 in our implementation. For a one megapixel image, storing all the non-zero matrix coefficients as single-precision floating point values requires around 2.6 gigabytes of memory. Solving the least squares problem efficiently is thus quite challenging. The implementation using CUDA is discussed in Section 5.3.4.

As with the original Poisson problem, it is beneficial to replace the L2-norm with the L1-norm. The system can then be solved using iteratively reweighted least squares, just as the original Poisson problem. In order to avoid changes in brightness due to the L1-reconstruction, we compute a second reconstruction pass using the noisy base image and the gradients from the result of our extended reconstruction as inputs. The second pass is simply a standard Poisson reconstruction using the L2-norm and a value of 0.05 for α. The low value of α limits the contribution of the noisy image to the overall brightness.

5.3.2 Patch constraint weights

Even though the feature subspace already includes information about edges in the image, the patch constraints still can lead to strongly overfiltered results if used without a suitable weighting matrix D_p. It does not make sense to enforce a common patch constraint on very different pixels. There are also always details, which are not present in the features. The weights can to some degree prevent overblurring in this case.

We weight the patch constraints by multiplying each row of $(B_p B_p^T - I_s)\Gamma_p \hat{I}$ with a suitable weight. Each row of $(B_p B_p^T - I_s)\Gamma_p \hat{I}$ measures the difference between the projection onto the feature subspace and the original patch in one pixel for each color channel. By weighting a row, it is possible to control the importance of the patch constraint at one pixel. We weight each row by a factor which depends both on the distance from the respective neighbor pixel to the center pixel of the patch as well as the variance of the Monte Carlo estimate at both pixels. Since the constraints are weighted based on the variance, our filter is consistent and the filtered image converges to the true image if more Monte Carlo samples are computed.

We calculate weights very similar to the weights used in RDFC. There are a few differences, which are worth discussing. First of all, the weights are not normalized. In our context, it does not make sense to have them sum up to one. The weights control the importance of the different rows in the equation system. Like RDFC, we compute both color and feature weights and then take their minimum to get the final weights.

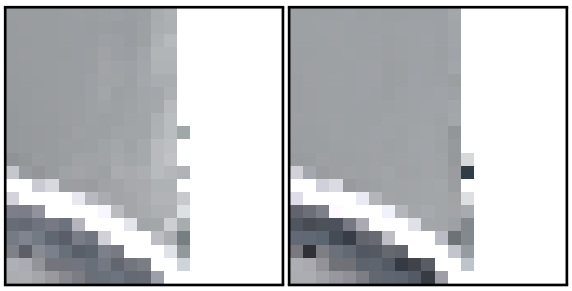
We compute the color weights from the solution of the original Poisson problem (using the L1-norm). The variance of the L1-reconstruction is computed using a two-buffer approach. We accumulate samples in two separate buffers, both for the base image and the gradients. We then solve the L1-reconstruction two times to compute the two-buffer variance. The resulting variance is then slightly blurred to reduce noise and the final variance estimate is then the maximum of the blurred two-buffer variance and the raw two-buffer variance. Taking the maximum prevents the variance estimate from being systematically too small, which can be an issue when dealing with strong outliers. We compute the color distance between two pixels as

$$\Delta_i^2(p,q) = \frac{(u_i(p) - u_i(q))^2 - (\mathrm{Var}_i[p] + \mathrm{Var}_i[q])}{\varepsilon + k_c^2(\mathrm{Var}_i[p] + \mathrm{Var}_i[q])} \ , \tag{5.9}$$

where p denotes the center pixel of the patch and q a neighboring pixel. The formulation is identical to the distance used in RDFC, except that we replace $\min(\mathrm{Var}_i[p], \mathrm{Var}_i[q])$ by $\mathrm{Var}_i[q]$ in the numerator. In RDFC, the minimum was taken to avoid blurring the central pixel p if the pixel q had a high variance. The patch constraints however filter both the central and the neighboring pixel. The color distances $\Delta_i^2(p,q)$ are then used to compute NL-means color weights $w_c(p,q)$ just like in RDFC. We compute the distance between two patches as

$$d^2(P(p), P(q)) = \frac{1}{3(2f+1)^2} \sum_{i=1}^{3} \sum_{n \in P(0)} \Delta_i^2(p+n, q+n) \ , \tag{5.10}$$

where f is the radius of the NL-means filter patches and $P(0)$ denotes all pixel offsets within a patch. The color weights are computed from the patch distance



(a) Without threshold (b) With threshold

Figure 5.8: Effect of thresholding patch constraint weights and adjusting feature vectors accordingly.

as

$$w_c(p,q) = \exp(-\max(d_c^2(P(p),P(q)),0)) \tag{5.11}$$

We do not use a symmetric weight formulation, as this would not make sense for our filtering technique. We also do not use the patch-wise extension. The feature distances are measured as

$$\Phi_j^2(p,q) = \frac{(f_j(p)-f_j(q))^2 - (\mathrm{Var}_j[p]+\mathrm{Var}_j[q])}{\varepsilon + k_f^2(V_j[p]+V_j[q]))} \ , \tag{5.12}$$

where $V_j[p] = \max(10^{-3}, \mathrm{Var}_j[p], \|\nabla_j[p]\|^2)$ and $V_j[q]$ is defined equivalently. We do not use the feature specific thresholds τ_j and instead simply normalize all features to have values in $[0,1]$. We again do not use the minimum in the numerator. Furthermore, for features with multiple color channels, e.g. the normals, we view each channel as one individual feature. We compute NL-means feature patch distances $d_{f_j}^2(p,q)$ equivalently to $d_c^2(p,q)$. The feature weights are then given as

$$w_f(p,q) = \min_{j\in\{1,\dots,M\}} \exp(-\max(d_{f_j}^2(p,q),0)) \ . \tag{5.13}$$

From the color and feature weights we compute the weights

$$w(p,q) = \min(w_c(p,q), w_f(p,q)) \tag{5.14}$$

Weighting the rows of the local patch constraint can in some cases not be sufficient. If the row corresponding to one pixel is weighted to nearly zero, it does not mean that this pixel has no influence on the reconstruction of the local patch anymore. The value of this pixel still occurs in the equations for the other pixels in the patch. An example of a case where this is a problem is shown in Figure 5.8a. The bright surface on the right leads to visible color bleeding artifacts. We reduce this problem by setting weights below a small threshold to zero. We then also set pixels in the features to zero if their weight is zero. This can be understood as reshaping the local neighborhood to not include any pixels which are very different from the central pixel. The new neighborhood is then in general not square anymore. The modified feature vectors are orthogonalized using SVD after the thresholding step. The pixels with zero weights then do

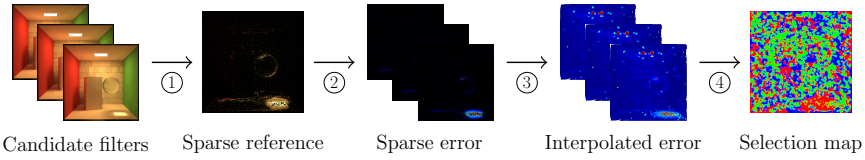


Figure 5.9: Overview of the used error estimation technique.

not occur in the constraints for the current patch anymore. The final weights are given as

$$\tilde{w}(p,q) = \begin{cases} \beta \cdot w(p,q) & \text{if } w(p,q) > \delta \\ 0 & \text{otherwise} \end{cases}, \tag{5.15}$$

where δ is a small threshold value and β scales the overall importance of the patch constraints. The weights $\tilde{w}(p,q)$ are the diagonal entries of the diagonal matrix D_p.

5.3.3 Error estimation using sparse error estimates

Because the reconstructed image is the result of an iterative least squares optimization, estimating the mean squared error is hard. Applying SURE does not seem to be possible, since this would require the computation of the differential of the filter. We ended up using the simple error estimation technique by Bauszat et al. [BEEM15].

This technique is specially made for error estimation in denoising of Monte Carlo rendered images. Figure 5.9 gives an overview of the error estimation technique. The idea is to first compute a number of candidate filters using different filter parameters. Based on these candidate filters, a sparse set of locations is generated, where a higher quality radiance estimate is computed by putting more samples in these pixels (step 1). These high-quality pixels are called *filter caches*. The sparse reference pixels are then used to compute a sparse error estimate for each candidate filter, by computing the squared difference between the sparse reference pixels and the candidate filter pixels (step 2). The sparse error estimates are then interpolated over the whole image to get an error estimate in each pixel (step 3). A filter selection map is then computed by selecting the locally best filter using the interpolated error (step 4).

The cache locations are selected based on the variance over the candidate filters: a pixel with very different values in each candidate filter needs a correct error estimation, while a pixel which has pretty much the same value in all candidate filters is less critical. Also, pixels with a lower variance in the unbiased Monte Carlo estimate should be selected more likely, since these will give a more reliable radiance estimate if sampled more. The probability density of the cache positions is defined on the set of all pixel locations Ω as

$$f(p) = \frac{f_F(p) f_V(p)}{\sum_{p \in \Omega} f_F(p) f_V(p)}, \tag{5.16}$$

where $f_F(p)$ is the variance over all candidate filters in pixel p and $f_V(p)$ is defined as

$$f_V(p) = e^{-\frac{\mathrm{Var}[p]}{2\sigma_r}} \ . \tag{5.17}$$

The term $\mathrm{Var}[p]$ is the variance of the Monte Carlo estimate in pixel p, averaged over the color channels of pixel p. We estimate the variance using the two-buffer variance. The constant σ_r is set to 0.15.

Only selecting caches using $f(p)$ can potentially leave larger regions without any caches. Bauszat et al. [BEEM15] therefore propose to generate a fraction of the cache locations using 2D Poisson-disc sampling. Poisson-disc sampling is a way of generating evenly distributed pseudo-random points. We use simple jittered sampling instead of Poisson-disc sampling, since we did not have a fast Poisson-disc implementation at our disposal and jittered sampling produces similar results.

The sparse error estimate is then interpolated by computing a Delaunay triangulation of the sparse locations and linearly interpolating the error over the triangles. The Delaunay triangulation algorithm computes a high-quality triangulation of the cache locations by maximizing the minimal triangle angles. Triangles with one or more very small angles are considered bad, since they can give very uneven interpolation results. Bauszat et al. [BEEM15] found the linear interpolation using the Delaunay triangulation to be just as good as more complicated interpolations. We also tried to interpolate the error using our patch constraints. The patch constraints can be used to interpolate the sparse error by optimizing for a solution equal to the sparse error pixels in the cache locations and close to a projection onto the feature subspace in all other pixels. This did however not give a significant quality increase over the linear interpolation and we thus ended up not using this idea. We do not use the graph cut based filter selection proposed by Bauszat et al. [BEEM15], because we found the seams in the optimal filter combination to be visually unproblematic.

5.3.4 Implementation using CUDA

We implemented our denoising algorithm on the GPU using CUDA. In the first step, the non-zero entries of the constraint matrix A need to be calculated. This requires computing the SVD of the feature vectors at each pixel. We used code by Moon et al. [MCY14] to achieve this. The code is based on the algorithm presented by Nash [Nas90]. Aside from the orthogonalization of the feature vectors, computing the constraints is straightforward.

The majority of computation time in our filtering algorithm is spent on solving the least squares problem. Our implementation is based on the Poisson solver written by Tero Karras for G-MLT [LKL+13]. This code is written in CUDA and is fairly optimized. Because the solver requires the computation of reductions over the whole image, we also used CUDA to implement our algorithm. Halide currently does not support fast parallel reductions on the GPU.

Like the original Poisson solver, our implementation uses the *conjugate gradient* (CG) method to efficiently solve our optimization problem. The conjugate gradient method iteratively solves a linear system $Cx = d$, where $C \in \mathbb{R}^{n \times n}$ is symmetric positive-definite. The CG method is guaranteed to give the exact solution after n steps, but can also be used with much less steps to get an

approximate solution.

To solve our least squares problem, we need to find the solution to the normal equation $A^TAx = A^Tb$. The matrix A^TA is symmetric and positive-definite and therefore the conjugate gradient method can be applied. In the iteratively reweighted least squares (IRLS) algorithm, we need to repeatedly solve equation systems of the form $A^TW^2Ax = A^TW^2b$. The CG algorithm is then:

CGSolve$(A, b, W, x_0, cgIter)$
1 $r_0 = A^TW^2b - A^TW^2Ax_0$
2 $p_0 = r_0$
3 **for** $i = 1$ **to** *cgIter*
4 $\tilde{p} = A^TW^2Ap_i$
5 $\alpha = (r_i^T r_i)/(p_i^T \tilde{p})$
6 $x_{i+1} = x_i + \alpha p_i$
7 $r_{i+1} = r_i - \alpha\tilde{p}$
8 $\beta = (r_{i+1}^T r_{i+1})/(r_i^T r_i)$
9 $p_{i+1} = r_{i+1} + \beta p_i$
10 **return** x_{i+1}

The matrix A and the vector b are given from our least squares problem, W is the IRLS weighting matrix, x_0 is a suitable initial guess of the solution (we use the base image for this) and *cgIter* is the number of iterations of the solver. We will not go into the derivation and geometric interpretation of the CG method, a good overview is given by Shewchuk [She94].

The most expensive step in the algorithm is the computation of the sparse matrix-vector product in line 4. The performance of this operation is limited by memory access, since all non-zero matrix elements need to be loaded. For our constraint matrix A, it is beneficial to precompute the matrix product A^TW^2A. For a patch radius of 2, the product has about seven times less non-zero elements than the matrix A itself. This reduces the time required to read matrix elements for the computation of the matrix-vector product. The performance increase due to the faster matrix-vector multiplication outweighs the cost of precomputing the sparse matrix product.

We implemented all matrix operations manually in CUDA. The regular structures of our sparse matrices allow to not use any index arrays and only load the non-zero values from memory. As the algorithm is limited by memory bandwidth, this is a very important optimization which directly translates into performance. We furthermore use shared memory and coalesced memory access patterns to achieve good performance.

We use a fixed number of conjugate gradient iterations, but it would also be possible to return earlier from the algorithm if it already has converged sufficiently. The convergence speed of the conjugate gradient algorithm depends on the root of the *condition number* of the matrix A^TW^2A. The condition number can be defined as the quotient of the largest and smallest eigenvalues. If it is close to one, the matrix is close to the identity matrix. Using our patch constraints, the condition number can get significantly larger than in the original Poisson problem and the algorithm can suffer from slow convergence. We thus use a default of 500 conjugate gradient iterations, while the original Poisson solver only uses 50 iterations. A typical solution to this problem is to resort

to the *preconditioned conjugate gradient* method. The idea is to multiply the constraint matrix by an invertible matrix M such that the condition number gets smaller and the algorithm converges faster. We did not attempt this, since finding a good preconditioner can be quite challenging. The algorithm could probably be significantly sped up using a suitable preconditioner.

5.4 Results

We integrated our filtering pipeline into the Mitsuba renderer [Jak10], on top of the implementation of G-BDPT [MRK+14]. We compute three different candidate filters with varying values for k_c, k_f and β. We set k_c to 0.5, 4 and 8 and k_f to 2, 8 and 8. The weight multiplier β is set to 0.5, 1 and 2. The NL-means patch radius is set to 2, the weight threshold δ to 10^{-11} and α to 0.5 for all candidate filters.

The samples needed to compute the sparse error estimates are subtracted from the samples available per pixel. This means that the total number of samples does not change with our method. Overall, our filtering method takes about half a minute to filter a 1 mega-pixel image on a Nvidia GTX Titan X GPU. The largest part of this time is spent on the computation of the three candidate filters. The cost for the Delaunay triangulation and interpolation is negligible.

In the following, we show several test scenes filtered using our algorithm and compare our reconstruction method to the result of the unbiased L2- and the biased L1-reconstruction. With our current parameter settings, our method consistently outperforms the L1-reconstruction by a factor of two to four.

(a) Filter output

L2: Full: 1.458 Inset: 3.335 | L1: Full: 1.750 Inset: 3.786 | Ours: Full: 0.501 Inset: 1.532 | Reference

(b) Detail comparison and MSE. All MSE values have been multiplied by 10^2.

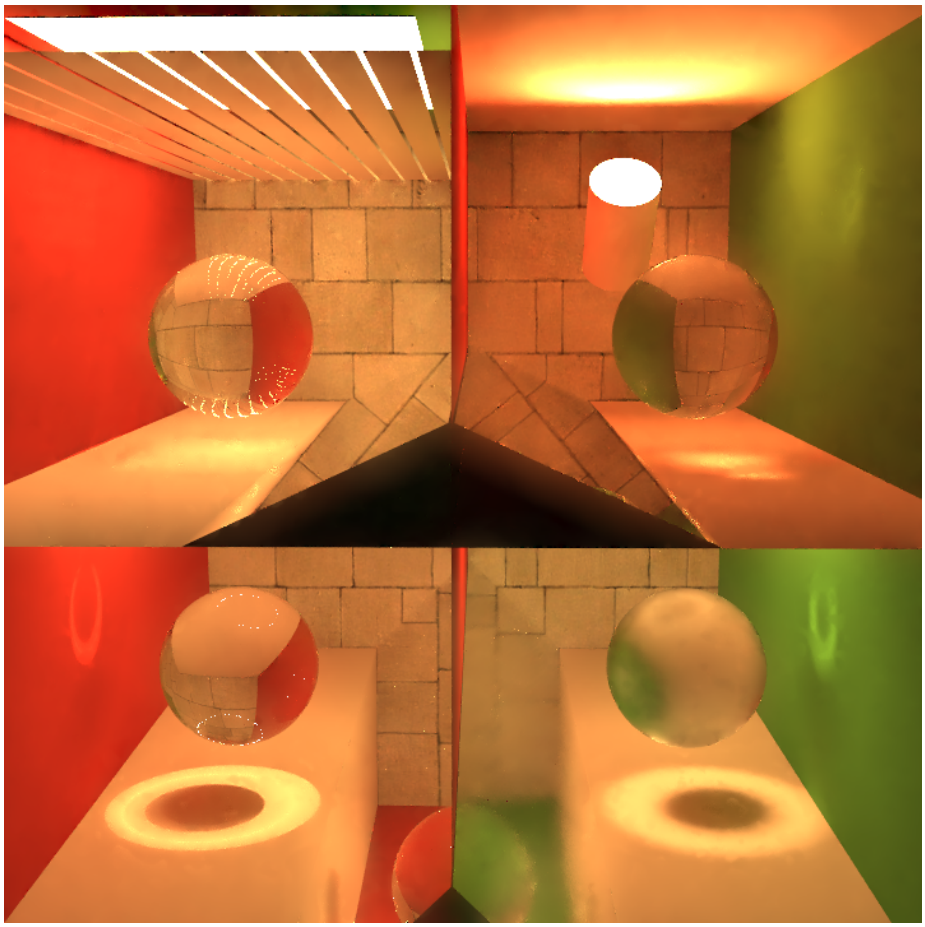
Figure 5.10: Result of our algorithm using the BATHROOM scene at 128 spp.

(a) Filter output

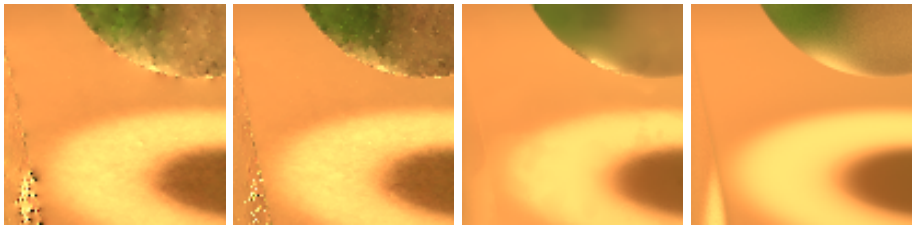
L2: Full: 0.716 Inset: 4.321 | L1: Full: 0.604 Inset: 4.210 | Ours: Full: 0.289 Inset: 0.992 | Reference

(b) Detail comparison and MSE. All MSE values have been multiplied by 10^2.

Figure 5.11: Result of our algorithm using the BOOKSHELF scene at 64 spp.



(a) Filter output

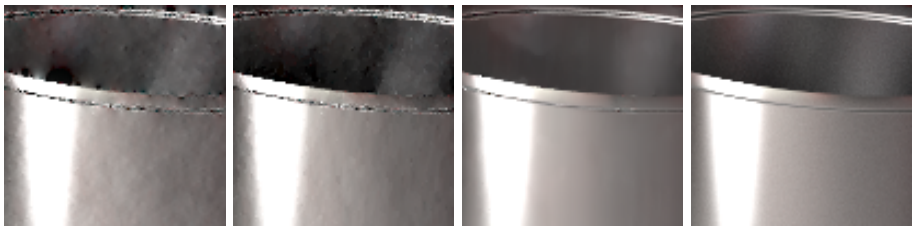


L2: Full: 10.436 Inset: 3.427 L1: Full: 7.566 Inset: 5.246 Ours: Full: 2.319 Inset: 0.831 Reference

(b) Detail comparison and MSE. All MSE values have been multiplied by 10^2.

Figure 5.12: Result of our algorithm using the BOXES scene at 32 spp.

(a) Filter output



L2: Full: 2.899 Inset: 4.437 | L1: Full: 3.366 Inset: 4.978 | Ours: Full: 1.056 Inset: 1.011 | Reference

(b) Detail comparison and MSE. All MSE values have been multiplied by 10^2.

Figure 5.13: Result of our algorithm using the KITCHEN scene at 128 spp.

(a) Filter output

L2: Full: 1.332 Inset: 1.884 | L1: Full: 2.448 Inset: 2.281 | Ours: Full: 0.624 Inset: 1.136 | Reference

(b) Detail comparison and MSE. All MSE values have been multiplied by 10^2.

Figure 5.14: Result of our algorithm using the Sponza scene at 16 spp.

Chapter 6

Conclusions

6.1 Working with Halide

We successfully implemented two different state-of-the-art denoising algorithms using Halide. For both algorithms we provide a sophisticated GPU as well as a simple CPU implementation. We were able to beat the performance of the existing implementations of both algorithms. The implementations of the two algorithms amount to a total of over 1400 lines of Halide code.

Halide does many things very well, but there are also a few downsides. Even though already in a usable state, Halide is still a fairly new language. Throughout working on this thesis, we encountered a variety of different bugs in the Halide framework. The bugs ranged from minor issues to more severe ones. At one point we had the problem, that Halide was not allocating the properly sized buffer on the GPU. Some of the Halide debug outputs did not even correctly show the size of the buffer that Halide internally allocated, which made it quite hard to locate the issue. We also had the automatic storage folding fail multiple times, which means that the allocated buffers were much larger than needed and often did not fit into memory. As already mentioned earlier, there is no way to manually control storage folding. A possible improvement to this would be to somehow allow the programmer to manually force Halide to fold a certain buffer. Not working storage folding can be a big obstacle in writing efficient code. Luckily, all the severe bugs we reported to the Halide developers, including storage folding bugs, were fixed within a matter of days.

As every new technology, Halide also does suffer from a lack of publicly available source code and libraries, especially for numerical computations. This can be an issue when working on the GPU, where it is not easily possible to call external C code from within Halide.

Since Halide, even though allowing for many low-level tweaks, is still an abstraction of the hardware, there is no access to some special platform specific commands. This is always the trade-off of new programming languages, which introduce a new level of abstraction on the hardware, and should be considered before using Halide. On the GPU, we cannot use multiple streams, textures and atomic operations.

As promised, Halide allows to quickly explore different schedules and to easily target different platforms, without having to rewrite the algorithm itself.

Optimizing code is much easier using Halide than using raw C++. In our applications, we found the benefit of working with Halide bigger when writing CPU code than when writing GPU code. The code we wrote using Halide to run on the CPU typically had a considerably more complex schedule than hand-written CPU code would have had. The Halide GPU code in contrast was a bit closer to "standard" GPU code. This is obviously also a matter of personal experience. Someone with more experience in writing fast CPU code would probably find the difference between their hand-written code and a fast Halide schedule to be smaller.

The functional paradigm of Halide seems to work very well for many image processing operations. Halide is typically very expressive and comes with little boilerplate code. As soon as complex schedules are involved, Halide code is much more readable than equivalent C++ code. This also makes Halide code less error prone. The seamless integration within existing C++ projects is also an advantage. The active support from the Halide developers and the continuous development of the Halide language are also very valuable.

In its current state, Halide is certainly not a general replacement for CUDA or OpenCL. The uncertainty coming with bugs, especially in the area of storage folding, is simply too big at the moment. Also some of the GPU features which are currently missing can be very important in some applications. The parallel reductions used in the conjugate gradient method could currently not be implemented efficiently in Halide. These issues make Halide not the first choice if deadlines are tight and the code only needs to run on a very limited number of machines. However, if portability to many different platforms is important and the limitations of Halide do not matter, it should seriously be considered. In these use cases, the benefits of Halide can outweigh the cost of fixing a few minor bugs in it. There are many frameworks which can be used to write portable image processing code, for example OpenCV [Bra00]. As far as we know, there is none other than Halide which both enables the programmer to write very fast code and still target different platforms without rewriting large parts of the algorithm.

6.2 Denoising for gradient-domain rendering

We introduced a new denoising algorithm for gradient-domain rendering and also showed that it can be implemented efficiently using CUDA. We integrated our algorithm into the Mitsuba renderer. Our reconstruction outperforms the existing L1-reconstruction by a significant factor. There are still many areas where the algorithm could be improved. By systematically optimizing the different filter parameters one could certainly get a slight overall improvement in quality. Furthermore, it probably makes sense to prefilter the features similar to RDFC. Even though truncating the SVD removes some of the noise from the feature subspace, there is still noise in the features which is not removed. Especially on edges of objects, there can be quite some noise in the features. Prefiltering would also allow to use noisy features, such as a caustic buffer, more effectively. There is also potential for performance improvement in the overall pipeline. Currently, all operations except the extended Poisson reconstruction run on the CPU. There are some computations, which could easily be ported to the GPU and are currently running on the CPU.

It would also be interesting, to make the patch constraints more adaptive. One idea would be to adjust the patch size of the constraints depending on the image. Instead of adding a patch constraint to each pixel, one could cover larger regions using less constraints. The method currently also does not explicitly handle animations. Extending the method into the time domain could be problematic due to memory requirements, but could also improve the filtering result. Finally, the convergence of the solver could possibly be improved using preconditioning. Even though the filtering process is already reasonably fast, it is still slower than many other denoising filters.

Besides denoising, it could also be interesting to explore other applications of our patch constraints. As already mentioned, they could also be used to interpolate sparse data. It would be interesting to try to apply them compressive rendering [SD11]. The idea behind compressive rendering is to only render a selection of pixels and interpolate the holes in the rendered image after rendering. We did some experiments in this direction using our patch constraints, but did not thoroughly evaluate how well this could work in practice.

6.3 Acknowledgments

I would like to thank Prof. Matthias Zwicker for his ongoing support throughout working on this thesis. He provided many ideas used in this thesis and always found time to discuss technical questions. My sincere thanks also go to my advisor, Marco Manzi, who spent countless hours discussing technical details with me and also integrated our denoising algorithm into the Mitsuba renderer.

Next I would like to thank the developers of Halide for their support when working with Halide. Andrew Adams answered many questions on Halide and fixed all Halide bugs I reported in a very short amount of time.

Last but not least, I would like to thank my friends and family, who always supported and encouraged me while working on this thesis.

Appendix A

Rader's algorithm

Similar to Bluestein's algorithm, Rader's algorithm [Rad68] reformulates the computation of the DFT using properties of the complex exponential terms. The derivation is based on basic concepts from the theory of groups and fields, but can also be understood without prior knowledge of these topics.

Let N be prime and $x = (x_0, \ldots, x_{N-1}) \in \mathbb{C}^N$. By definition, the DFT coefficients of the signal x are computed as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n/N} \ . \tag{A.1}$$

Because N is prime, the ring $\mathbb{Z}/N\mathbb{Z}$ is a field and therefore the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^*$ is cyclic and equals $((\mathbb{Z}/N\mathbb{Z})\backslash\{0\}, \cdot)$. In other words, there exists a number $g \in \{2, \ldots, N-1\}$ such that $\{g^i \mod N | i \in \{1, \ldots, N-1\}\} = \{1, \ldots, N-1\}$. This implies, that the map $k \mapsto g^k \mod N$ is a bijective map of the set $\{1, \ldots, N-1\}$ onto itself. Furthermore, the map $k \mapsto g^{-k} \mod N$ is also bijective on the same set. Here, $g^{-k} \mod N$ is simply the multiplicative inverse of $g^k \mod N$ in $(\mathbb{Z}/N\mathbb{Z})^*$. The bijectivity of this second map is given by the fact, that in a group, every element has a unique inverse. Using these two maps, the definition of the DFT can be rewritten as a convolution. First, we write the definition as:

$$X_k - x_0 = \sum_{n=1}^{N-1} x_n e^{-2\pi i k n/N} \tag{A.2}$$

Then, we substitute $g^{-r} \mod N = k$ and $g^q \mod N = n$, where $q, r \in \{1, \ldots, N-1\}$ are uniquely determined by the bijections described previously. Using $g^0 = g^{N-1} \mod N$, this gives the following equality:

$$X_{g^{-r}} - x_0 = \sum_{q=0}^{N-2} x_{g^q} e^{-2\pi i g^q g^{-r}/N} = \sum_{q=0}^{N-2} x_{g^q} e^{-2\pi i g^{-(r-q)}/N} \tag{A.3}$$

We assume, that the signal is N periodic, i.e. $x_k = x_{k \mod N}$. The same property holds for the exponential term. The DFT is now the cyclic convolution of the sequences $a_k = x_{g^k}$ and $b_k = e^{-2\pi i g^{-k}/N}$. This uses the periodic behavior

of the exponential term $e^{-2\pi i g^{-k}/N}$ and the signal x. Note, that X_0 cannot be calculated this way and still is computed as:

$$X_0 = \sum_{n=0}^{N-1} x_n \tag{A.4}$$

The cyclic convolution can now be calculated by applying the convolution theorem to the two sequences of length $N-1$. The required Fourier transforms can be computed efficiently by using an existing FFT implementation, as $N-1$ is not prime. The Fourier transform of the sequence $(b_0, \ldots, b_{N-2})$ can be precomputed, as it does not depend on the signal. It is also possible, to apply the algorithm recursively to deal with large prime factors, that could occur when factoring $N-1$. If $N-1$ contains large prime factors, it can also be more efficient to pad the sequences with zeros to a highly composite length $N' \geq 2N-3$.

List of Tables

List of Figures

Bibliography

[BCCZ08] Bhat, Pravin ; Curless, Brian ; Cohen, Michael ; Zitnick, C. L.: Fourier Analysis of the 2D Screened Poisson Equation for Gradient Domain Problems. In: *Proceedings of the 10th European Conference on Computer Vision: Part II*, 2008, S. 114–128

[BCM05] Buades, Antoni ; Coll, Bartomeu ; Morel, Jean-Michel: A Review of Image Denoising Algorithms, with a New One. In: *Multiscale Modeling & Simulation* 4 (2005), Nr. 2, S. 490–530

[BEEM15] Bauszat, Pablo ; Eisemann, Martin ; Eisemann, Elmar ; Magnor, Marcus: General and Robust Error Estimation and Reconstruction for Monte Carlo Rendering. In: *Computer Graphics Forum* 34 (2015), Nr. 2, S. 597–608

[BL07] Blu, Thierry ; Luisier, Florian: The SURE-LET Approach to Image Denoising. In: *IEEE Transactions on Image Processing* 16 (2007), Nr. 11, S. 2778–2786

[Blu70] Bluestein, Leo: A linear filtering approach to the computation of discrete Fourier transform. In: *IEEE Transactions on Audio and Electroacoustics* 18 (1970), Nr. 4, S. 451–455

[Bra00] Bradski, G.: OpenCV Library. In: *Dr. Dobb's Journal of Software Tools* (2000)

[DFKE07] Dabov, Kostadin ; Foi, Alessandro ; Katkovnik, Vladimir ; Egiazarian, Karen: Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering. In: *IEEE Transactions on Image Processing* 16 (2007), Nr. 8, S. 2080–2095

[EA06] Elad, Michael ; Aharon, Michal: Image Denoising Via Sparse and Redundant Representations Over Learned Dictionaries. In: *IEEE Transactions on Image Processing* 15 (2006), Nr. 12, S. 3736–3745

[FJ05] Frigo, Matteo ; Johnson, Steven G.: The Design and Implementation of FFTW3. In: *Proceedings of the IEEE* 93 (2005), Nr. 2, S. 216–231

[GLD+08] Govindaraju, Naga K. ; Lloyd, Brandon ; Dotsenko, Yuri ; Smith, Burton ; Manferdelli, John: High Performance Discrete Fourier Transforms on Graphics Processors. In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008

[Jak10] Jakob, Wenzel: *Mitsuba renderer.* 2010 `http://www.mitsuba-renderer.org`

[JM12] Jakob, Wenzel ; Marschner, Steve: Manifold exploration. In: *ACM Transactions on Graphics* 31 (2012), Nr. 4, S. 1–13

[Kaj86] Kajiya, James T.: The Rendering Equation. In: *SIGGRAPH Comput. Graph.* 20 (1986), Nr. 4, S. 143–150

[KMA+15] Kettunen, Markus ; Manzi, Marco ; Aittala, Miika ; Lehtinen, Jaakko ; Durand, Frédo ; Zwicker, Matthias: Gradient-Domain Path Tracing. In: *ACM Transactions on Graphics, to appear* (2015)

[KZ13] Knaus, Claude ; Zwicker, Matthias: Dual-domain image denoising. In: *20th IEEE International Conference on Image Processing (ICIP)*, 2013, S. 440–444

[KZ15] Knaus, Claude ; Zwicker, Matthias: Dual-Domain Filtering. In: *SIAM Journal on Imaging Sciences* (2015)

[LA04] Lattner, Chris ; Adve, Vikram: LLVM: A compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004.*, 2004, S. 75–86

[LKL+13] Lehtinen, Jaakko ; Karras, Tero ; Laine, Samuli ; Aittala, Miika ; Durand, Frédo ; Aila, Timo: Gradient-domain metropolis light transport. In: *ACM Transactions on Graphics* 32 (2013), Nr. 4

[LWC12] Li, Tzu-Mao ; Wu, Yu-Ting ; Chuang, Yung-Yu: SURE-based optimization for adaptive sampling and reconstruction. In: *ACM Transactions on Graphics* 31 (2012), Nr. 6

[MCY14] Moon, Bochang ; Carr, Nathan ; Yoon, Sung-Eui: Adaptive Rendering Based on Weighted Local Regression. In: *ACM Transactions on Graphics* 33 (2014), Nr. 5, S. 1–14

[MKA+15] Manzi, Marco ; Kettunen, Markus ; Aittala, Miika ; Lehtinen, Jaakko ; Durand, Frédo ; Zwicker, Matthias: Gradient-Domain Bidirectional Path Tracing. In: *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*, 2015

[MRK+14] Manzi, Marco ; Rousselle, Fabrice ; Kettunen, Markus ; Lehtinen, Jaakko ; Zwicker, Matthias: Improved sampling for gradient-domain metropolis light transport. In: *ACM Transactions on Graphics* 33 (2014), Nr. 6, S. 1–12

[Nas90] Nash, John C.: *Compact numerical methods for computers: Linear algebra and function minimisation.* 2nd ed. 1990

[PH10] Pharr, Matt ; Humphreys, Greg: *Physically based rendering: From theory to implementation.* 2nd ed. 2010

[Rad68] Rader, Charles M.: Discrete Fourier transforms when the number of data samples is prime. In: *Proceedings of the IEEE* 56 (1968), Nr. 6, S. 1107–1108

[RKAP+12] Ragan-Kelley, Jonathan ; Adams, Andrew ; Paris, Sylvain ; Levoy, Marc ; Amarasinghe, Saman ; Durand, Frédo: Decoupling algorithms from schedules for easy optimization of image processing pipelines. In: *ACM Transactions on Graphics* 31 (2012), Nr. 4, S. 1–12

[RKZ12] Rousselle, Fabrice ; Knaus, Claude ; Zwicker, Matthias: Adaptive rendering with non-local means filtering. In: *ACM Transactions on Graphics* 31 (2012), Nr. 6, S. 195:1–195:11

[RMZ13] Rousselle, Fabrice ; Manzi, Marco ; Zwicker, Matthias: Robust Denoising using Feature and Color Information. In: *Computer Graphics Forum* 32 (2013), Nr. 7, S. 121–130

[SD11] Sen, Pradeep ; Darabi, Soheil: Compressive Rendering: A Rendering Application of Compressed Sensing. In: *IEEE Transactions on Visualization and Computer Graphics* 17 (2011), Nr. 4, S. 487–499

[She94] Shewchuk, Jonathan R.: *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* 1994

[Smi07] Smith, Julius O.: *Mathematics of the discrete Fourier transform (DFT): With audio applicaitons.* 2nd ed. 2007

[Ste81] Stein, Charles M.: Estimation of the Mean of a Multivariate Normal Distribution. In: *The Annals of Statistics* 9 (1981), Nr. 6, S. 1135–1151

[TM98] Tomasi, Carlo ; Manduchi, Roberto: Bilateral filtering for gray and color images. In: *IEEE 6th International Conference on Computer Vision*, 1998, S. 839–846

[Vea98] Veach, Eric: *Robust Monte Carlo Methods for Light Transport Simulation.* 1998

[VF08] Van den Berg, Ewout ; Friedlander, Michael P.: Probing the Pareto frontier for basis pursuit solutions. In: *SIAM Journal on Scientific Computing* 31 (2008), Nr. 2, S. 890–912

[ZJL+15] Zwicker, Matthias ; Jarosz, Wojciech ; Lehtinen, Jaakko ; Moon, Bochang ; Ramamoorthi, Ravi ; Rousselle, Fabrice ; Sen, Pradeep ; Soler, Cyril ; Yoon, Sung-Eui: Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. In: *Computer Graphics Forum* 34 (2015), Nr. 2, S. 667–681

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: ……

Bachelor ☐ Master ☐ Dissertation ☐

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....
Ort/Datum

.....
Unterschrift