Utrecht University

THESIS

High-quality interactive path tracing

Student:
Mathijs LARDINOIJE

Project supervisor (first examiner):
dr. ing. J. BIKKER

Second examiner:
dr. A. VAXMAN

Daily supervisor:
Huub VAN SUMMEREN

ICA-3760219

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Master's programme Game and Media Technology
Department of Information and Computing Sciences

March 31, 2018

Abstract

Although still computationally expensive, path tracing is continually becoming more promising as a render technique for use in a production environment, due to the continuing increase in generally available compute power. We research the feasibility of path tracing in a scenario where high-quality images must be rendered at interactive rates, by integrating a path tracer in a product configurator framework and evaluating its performance. We evaluate the current state-of-the-art in path tracing postprocessing denoising filters, and propose our own novel filter aimed specifically at interactive path tracing, to reduce render times significantly.

Contents

Chapter 1

Introduction

Path tracing is a physically based render technique that has the capability of producing images of higher quality than traditional renderers based on rasterization. Compared to rasterizers however, path tracing requires at least an order of magnitude more compute power to render images of such quality. Path tracing is already used in many production environments where high-quality rendering has priority over short render times: Computer generated effects in movies are often rendered by means of path tracing, and some movies are even completely path traced. High-quality product renders, promoting a new car model for example, are often rendered by means of path tracing too. The aforementioned applications for path tracing share the fact that all images can be pre-rendered, tolerating long render times taking minutes or in some cases hours to finish. However, as compute power continually increases due to technological advances, these render times are continually becoming shorter. While high-quality real-time path tracing, applicable to for example games, is still likely to be impossible today, high-quality interactive path tracing could be feasible, or become feasible in the near future.

We refer to render times of ideally under a second, but possibly up to a couple of seconds, as being in the domain of interactive rendering. A practical use case requiring render times of this order in a production environment is the on-demand rendering of high-quality product images, as is needed in a product configurator. A product configurator is a service, usually made available through a website, where a user can customize a product they are interested in before buying. Arguably the most well-known example of this is the possibility to customize a car you are interested in on the maker's website, choosing your own desired exterior and interior colors and possible options and extras. However, product configurators are employed in many more markets, like for example in furniture or clothing.

A number of key aspects fundamental to product configurators are suited particularly well to high-quality interactive path tracing:

- High-quality rendering, path tracing's main strength, is of utmost importance, to provide the best possible impression of the product to a potential buyer.
- Short wait times are tolerable, allowing the use of interactive rendering.
- Pre-rendering is not feasible due to the amount of possible combinations of a customizable product, thus an on-demand solution is required.

To evaluate the feasibility of high-quality interactive path tracing, we integrate a path tracer in an existing product configurator framework, created by 3DIMERCE, and optimize it for interactive rendering by implementing state-of-the-art and novel postprocessing filter techniques to significantly decrease render times.

1.1 3DIMERCE

3DIMERCE is a small company located in Eindhoven, and was founded in 2001. Their main service is the conversion of real-life objects to 3D models of which then a cloud service is provided that hands out on-demand renders of these models.

To obtain accurate 3D models the client's objects are scanned with a 3D scanning tool, and the object's materials are scanned with a material scanning tool, to obtain material details such as textures and bumpmaps. Any customizable options of the objects are also digitalised. After the conversion process to 3D is completed, an API is given out through which clients can request renders of their objects. 3DIMERCE's servers handle the requests and reply with rendered images. In the requests, configuration options can be supplied, such as object color, background scene, camera location, and whether or not to add some accessories. Commonly the client owns a web application with some sort of product configurator, which uses 3DIMERCE's service for generating renders of the configured products (figure 1.1).

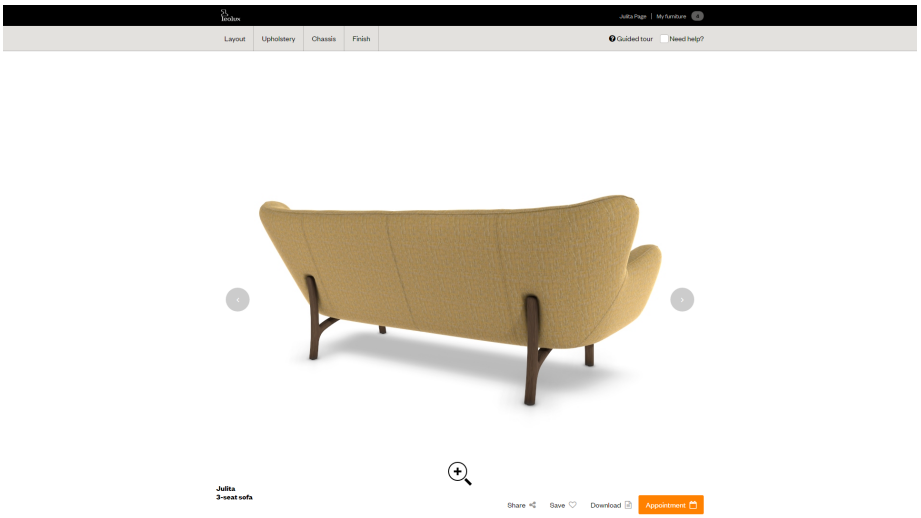

FIGURE 1.1: A product configurator with an image rendered by 3DIMERCE's service.

A lot of care is taken to make sure that the rendered images are of high quality, to keep customer satisfaction high and 3DIMERCE ahead of rivaling companies. Therefore, 3DIMERCE is always researching new techniques to increase image quality. Currently, they are using Unity3D's rasterization engine to render the images. Over the past few years 3DIMERCE has constantly been augmenting and upgrading this engine to keep increasing image quality. Because of this, 3DIMERCE's in-house version of the Unity engine has become quite cumbersome. It contains many post-processing effects that are being applied one after the other, and makes use of a single large shader for simulating many different effects.

3DIMERCE's latest effort in increasing image quality is to try and add global illumination to their renderer. However, this is proving to be difficult. Due to the high amount of effects already in use, finding or implementing a global illumination algorithm that is compatible with these effects is problematic. Because of the high number of effects the renderer is hard to maintain too. Also, most global illumination algorithms for rasterization engines are merely approximations of physically correct algorithms, resulting in inferior quality compared to path tracing.

Because of the problems mentioned above, 3DIMERCE wants to research whether switching from a rasterizer to a path tracer might be beneficial for them.

1.2 Objective and contribution

Our main objective is to find out whether high-quality interactive path tracing for use in a production environment is possible today, or in case it is not, how far off we currently are from this goal. Subsequently, we pose our main research question as:

How fast can high-quality images be rendered, by means of path tracing?

With the following remarks:

- The image should be of a visibly higher quality than its rasterization counterparts, and free of any noise, bias, or other types of glitches and errors associated with path tracing, to be deemed high-quality.
- The scenes to be rendered are production environment ready, portraying customizable high-polycount products rendered at production-level resolutions, as would be the case in a product configurator.
- We aim for interactive render times, which take ideally under a second, but can possibly take up to a couple of seconds.

To decrease required render times significantly, thus increasing the odds of a positive answer to the main research question, we extend the current path tracing algorithm with state-of-the-art and novel postprocessing filter techniques. We provide a scientific contribution by researching, implementing, and comparing such techniques.

1.3 Research methodology

We first integrate a path tracer in 3DIMERCE's framework, to enable the rendering of product configurator scenes of high-quality. Albeit with long render times, this gives us a frame of reference to improve upon with postprocessing filters. We choose to integrate a custom, highly-optimized and state-of-the-art path tracer, giving us the maximum currently possible path tracing performance. We have access to this path tracer's source code, which enables us to perform the modifications required for a successful integration.

Next, we implement a reference filter, based on the most promising state-of-the-art filtering technique for interactive path tracing, as concluded by our literature study. We then start iteratively researching and implementing other state-of-the-art and our own novel filtering techniques, to improve upon the reference filter implementation. The iterations are driven by actively targeting weaknesses of the current implementation, to try and end up with a more robust and better performing filter.

Finally, we perform both an absolute and relative evaluation of our final implementation, by measuring the time required for rendering high-quality images with and without our filter, and comparing our filter to the current state-of-the-art in postprocessing filters. We also compare the path traced renders against reference renders to assess the improvement in quality and ensure no glitches or errors are introduced in the path traced renders.

1.4 Test method and evaluation

We evaluate the work proposed in three areas: the absolute quality of the path traced images, the amount of time required to render these images (with and without our filter), and the performance of our filter compared to the state-of-the-art.

Quality We must consider two aspects when evaluating render quality: the correct presence of graphical effects caused by path tracing, and the absence of any noise, bias, glitches or errors caused by path tracing. We compare a reference image rendered by a rasterization engine and a path traced image of the same scene side by side. The scene to be rendered is production environment ready, portraying a high-polycount product using complex materials like translucent surfaces. In this evaluation, render time is irrelevant. Therefore the path tracer will be configured to keep rendering until all visible noise has disappeared, and no postprocessing filter will be applied to speed up render times.

For the graphical effects evaluation, we compiled a list of the most prominent graphical effects that should be present in the path traced image (table 1.1).

Effect	Rasterizer	Path tracer
Indirect lighting: Not directly lit surfaces receiving light reflected off other surfaces	Ambient lighting (physically incorrect)	Natural effect of path tracing (physically correct)
Color bleeding: Surfaces receiving colored light reflected from nearby surfaces	-	Natural effect of path tracing (physically correct)
Contact shadows: Darkening of creases and other areas occluded from light	Ambient occlusion (physically incorrect)	Natural effect of path tracing (physically correct)
Soft shadows: Shadows dissipating smoothly at their edges	-	Natural effect of path tracing (physically correct)
Reflections: Correct reflection of light by specular surfaces	Screen space reflections (physically incorrect)	Natural effect of ray tracing (physically correct)
Anti-aliasing: Removal of jagged edges	Supersampling	Pixel area random sampling
Translucency: Correct transmission of light through transparent surfaces	-	Natural effect of ray tracing (physically correct)

TABLE 1.1: A list of the required graphical effects that should be visible in the path traced image.

We perform a visual inspection to check for noise, bias, errors, and glitches in the path traced image. Noise should not be present as we will let the path tracer render until the image is fully converged. Bias shouldn't be present either, as this is usually introduced only after applying a postprocessing filter. The most important aspect to check will be any presence of errors or glitches, which can be caused by improper integration of the path tracer in the product configurator framework.

If all required graphical effects are present, and no other defects can be found in the path traced image, we deem the generated images to be of high quality.

Interactivity We evaluate the interactivity of the integrated path tracer by measuring the required time to render a high-quality image. We differentiate between an evaluation with applying a postprocessing filter and without, to be able to record the reduction in render times the postprocessing filter provides. For the evaluation without a postprocessing filter, we keep rendering until any visible noise has completely disappeared from the image. For the evalution with a postprocessing filter, we keep rendering until any visible bias has completely disappeared from the image. This differentiation is due to the inherent design of postprocessing filters, removing all noise in a render but doing so at the cost of introducing bias, usually in the form of blurring or general loss of detail. The scenes used for the evaluation will be of the same type as in the previous area of evaluation.

We plot graphs of the measured *mean squared error* (MSE) of filtered and unfiltered renders over time too, to provide scientifical measurements. This is a frequently used metric in path tracing to quantify the difference between two renders. We use it to measure the difference between a perfectly rendered noise-free image and the image to evaluate. The noise-free image is obtained once by instructing the path tracer to render without a time limit and without applying a filter, until the image is fully converged (no longer containing any noise). Given every RGB pixel in the reference image $y_1, \ldots, y_n$ and every RGB pixel in the image to evaluate $y'_1, \ldots, y'_n$, we calculate MSE as:

$$\frac{1}{n}\sum_{i=1}^{n}(|y'_iRED - y_iRED|^2 + |y'_iGREEN - y_iGREEN|^2 + |y'_iBLUE - y_iBLUE|^2)$$

Not only images with noise will be penalised by MSE, but also images that overblur, lose detail, or that contain other graphical glitches, as the MSE metric penalizes all pixels that diverge from the ground truth image, regardless of the type of diversion.

Filter performance We compare our novel filter to the state-of-the-art by measuring MSE on various scenes and sample counts, for all filters. To avoid integrating all filters in our path tracer, we compare on well-known scenes only which are commonly used in filtering literature. These scenes are integrated in the publicly available path tracer *pbrt*, for which implementations of all state-of-the-art filters exist. We use this path tracer to generate the filtered images for the state-of-the-art filters. We also plot the measured runtimes (overhead) of each filter, to evaluate filtering performance in relation to filter overhead.

1.5 Thesis structure

We start with a preliminary chapter (2) about path tracing, for readers who are not familiar with the technique. The following chapter (3), contains our literature study about postprocessing filters and its results. Next, we detail our own implementations of existing and novel filters (4), and share the results of our evaluation (5). The study of the product configurator framework, integration of the path tracer into the framework, and the objective, evaluation, and results for 3DIMERCE are placed in a separate chapter (6). We finish with a discussion about the work presented and its results, and discuss any possible future work, in the final chapter (7).

Chapter 2

Preliminaries

2.1 Path tracing

A path tracer is a physically based renderer, meaning the render process aims to simulate the way light is transported in real life. In nature, light sources emit photons that travel in straight lines until they hit a surface, which we model as rays in a path tracer. If a surface gets hit, it can (partially) absorb the ray's energy, and/or (partially) reflect the ray. For example, a diffuse red surface will absorb all energy of the incoming ray except the red wavelength's energy, which gets reflected in a random direction. Next, the reflected ray travels in a straight line again until it hits another surface, which will manipulate the ray again, and so on. Some of these rays might eventually hit our eyes, which enables us to see the world in front of us. A path tracer simulates this process by shooting rays in a reversed direction, starting from an artificial eye, and then letting the rays bounce around until they hit a light source. The light information obtained while tracing such a ray's path determines the ultimate colors in the finished render.

By simulating light transport as described above, a renderer can produce photorealistic images without relying on the addition of artificial effects. Effects commonly used in rasterization engines, such as *global illumination* and *soft shadows*, occur naturally when performing light transport simulation. Regarding global illumination for example (figure 2.1), one can perceive a visual effect known as *color bleeding*: the coloring of surfaces with reflected light from nearby surfaces. This effect is inherently visualized in a path tracer when a ray is reflected off a surface onto another one nearby. In a rasterization engine, a separate effect has to be implemented that simulates this behaviour.

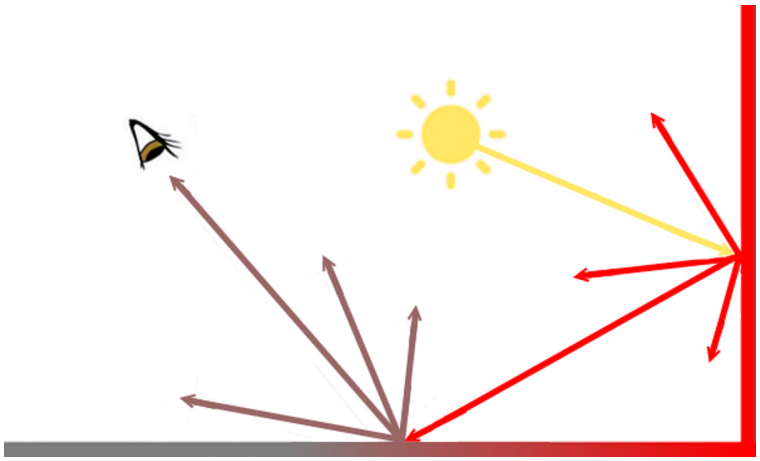

FIGURE 2.1: Color bleeding in a path tracer: the red surface bleeds its color on the nearby gray surface.

2.2 The rendering equation

Path tracing revolves around physically correct simulation of light transport. Generally, path tracers approximate light transport by solving the *rendering equation*, first introduced by Kajiya [Kaj86]. It is defined as:

$$I(x \leftarrow x') = G(x \leftarrow x')[E(x \leftarrow x') + \int_S R(x \leftarrow x' \leftarrow x'')I(x' \leftarrow x'')dx'']$$

Where:

- $I(x \leftarrow x')$ is all light passing from x' to x.
- $G(x \leftarrow x')$ describes the geometry factor between x and x', returning 0 if x and x' are not directly visible to eachother, otherwise returning a factor based on the distance and surface angle between x and x'.
- $E(x \leftarrow x')$ is the light directly emitted by x' to x, returning 0 unless x' is a light source.
- S is the domain of integration, containing all surface points of the entire scene. This can include the scene's hemisphere, if skybox lighting is implemented.
- $R(x \leftarrow x' \leftarrow x'')$ is the light reflected from x' to x, that originates from x'' and is calculated by evaluating $I(x' \leftarrow x'')$.

Solving this equation for every possible $I(x \leftarrow x')$, by iterating over every pixel to render for x, and iterating over every surface/hemisphere point in the scene for x', would result in the exact answer to the rendering equation, synthesizing a perfect physically based image. However, this is impossible, practically due to the computation time required and theoretically due to the fact that the amount of points on a surface is infinite, as points have no area. Therefore path tracers make use of *Monte Carlo integration* to approximate the answers, as described in the next section.

2.3 Monte Carlo integration

With *Monte Carlo integration* the integral over the set S in the calculation of $I(x \leftarrow x')$ is replaced by the sampling of a uniformly chosen subset $x'_1, \ldots, x'_N \in S$.

By stochastically sampling this subset we can approximate the integral's value:

$$\int_S \approx \frac{1}{N} \sum_{i=1}^{n} x'_i \in S$$

This holds true because the *law of large numbers* ensures that the Monte Carlo integration equals the integral over the set S, when $N = \infty$. As N increases, the variance in the approximation of the integral decreases. We only reach the integral's exact value after taking infinite samples.

Practically, this implies that S can be sampled by tracing a path if, everytime a surface is hit, a random direction within the cone of available directions for that surface is chosen. The cone's properties are dependent on the surface. For example, on a purely specular surface (a mirror) there is only one available direction in this cone: the direction with angle of reflection equal to angle of incidence (proportional to the surface). But for a purely diffuse surface, all directions contained in the surface's hemisphere are inside the cone of available directions.

Another aspect to consider is that the paths in the rendering equation are theoretically infinite, as it is an equation where $I(x \leftarrow x')$ is called recursively. To be able to still compute paths within finite time, a technique called *Russian Roulette* is employed. Every time a ray hits a surface, the ray is not allowed to bounce (i.e. evaluate another $I(x \leftarrow x')$) with probability p, terminating the traced path at that point and returning its results so far as a valid sample. Paths that are not terminated then have their sample weights scaled by $\frac{1}{p}$, to compensate for the fact that light energy has been removed from the equation by terminating some other paths. Eventually all paths reach a light source or get terminated.

Evaluating one sample of the render equation for each pixel, using random directions within the available cone of directions for each bounce, and terminating the path at each bounce with probability p, is the foundation of applying a render pass in a Monte Carlo path tracer. After each render pass the results of that pass are added to the rendered image. The final correct color for a pixel is the average of the colors that all traced paths through that pixel return. An example of this incremental rendering system, as used by Monte Carlo path tracers, is provided in figure 2.2.

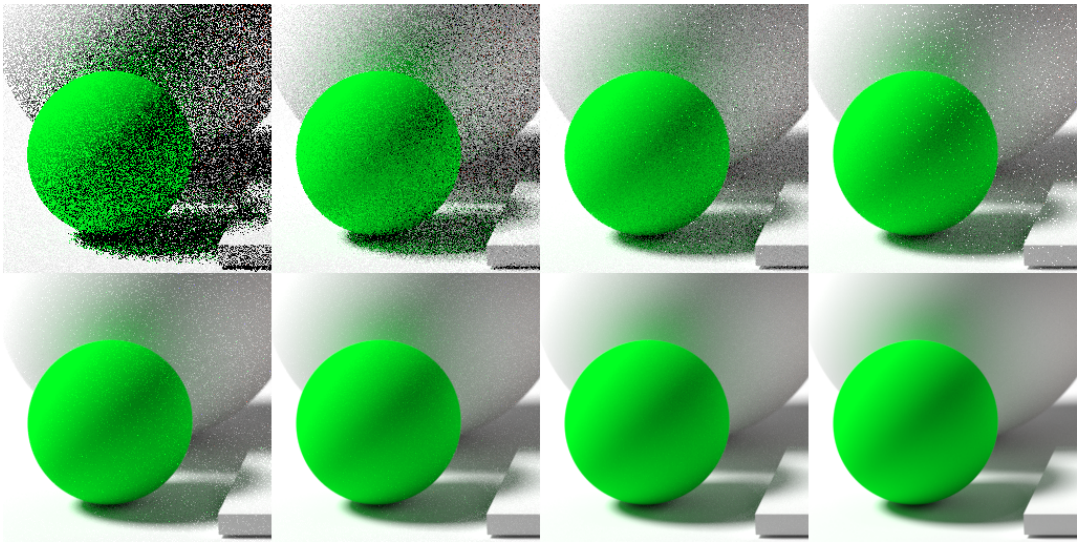

FIGURE 2.2: Incremental rendering with a path tracer. Noise is reduced with more render passes.

Chapter 3

Previous work

Monte Carlo path tracing is slow. Depending on the scene, thousands of samples per pixel are needed to obtain a perfect noise-free image. In turn, to calculate one sample of the render equation, multiple rays have to be traced. Therefore, to render an image containg a couple of million pixels, multiple billions of rays need to be traced. Currently, the fastest GPUs reach a throughput of about a hundred million to one billion rays per second on fully optimized Monte Carlo path tracers. Thus, to render a noise-free image, render times in the order of tens of seconds are needed. To increase path tracer performance as much as possible, and possibly enable interactive rendering, a denoising filter can be implemented. This filter receives a noisy image as input, and transforms it into a noise-free image, by analyzing the results so far and then filtering out the noise. For the denoising filter to successfully enable interactive rendering, it should be able to reduce the required sample count by at least an order of magnitude.

We first introduce the different types of denoising filters currently popular in literature, and explain which type of denoising filter we target and why. Next, we selected a list of the most promising state-of-the-art filters currently available in literature, and describe them in detail. Finally, we compare the chosen filters in the context of interactivity and reason how they can help us in implementing a filter suitable for interactive rendering.

3.1 Filtering

Denoising filters come in two variants: *post-processing* and *adaptive* filters.

Post-processing filters These are applied once on the finished render. One analysis is performed, making use of all the results so far, after which the image is filtered.

Adaptive filters These are applied iteratively, inbetween a predetermined number of render passes. They perform an analysis to estimate local variance (noisiness) of the render so far. Using this information, a sampling density is allocated to the pixels. During the successive render passes, the noisy pixels are then sampled more, to reduce the total amount of noise more efficiently than with uniform sampling. Often, every iteration the image is also filtered.

Another important aspect of filters to consider, is their knowledge of (aspects of) light transport, and their level of *generality* because of this. Filters can be divided again into two variants: *a priori* and *a posteriori* filters.

A priori These filters perform an informed analysis, where the logic used is based on light transport theory that has been acquired *in advance*. Often these filters focus their analysis on certain specific aspects of light transport, e.g. diffuse global illumination or soft shadows. This is done because the complexity of the used light transport analysis is often so high that only by reducing the analysis to one or a couple of aspects it can become computable within reasonable time. As these filters try to model light transport equations, often a lot of information is required for the analysis, such as geometric and material information of all samples. Filtering of the targeted aspects is often of good quality, however applying these filters to scenes that do not have the targeted aspects as prominent features gives mixed results.

A posteriori Filters that perform an uninformed analysis have no explicit knowledge of light transport theory. Instead of requiring a lot of information they usually only work on the information that is contained *afterwards* in the resulting image, e.g. the colors, normals, and textures of the final pixels or samples. Because of this, these filters are more general: They do not perform exceptionally well on certain light transport effects, but do however provide more consistent results. In comparison to *a priori* filters, these filters often only work in image-space.

Note that most adaptive filters are *a priori*, to estimate variance resulting from a certain light transport aspect, and that most post-processing filters are *a posteriori*, since both work best after image rendering. Combinations the other way around are not very uncommon however. Also, many adaptive filters perform post-processing like filtering steps in each iteration, to enhance their effectiveness.

We focus on *a posteriori* post-processing filters. Implementing a post-processing filter is a lot more practical than implementing an adaptive filter. Adaptive filters are tightly intertwined with the used renderer, being applied in between render passes and allocating sample budgets to the renderer. It is more straightforward to implement a filter that is only applied after the render process has finished. We consider *a posteriori* filters superior to their *a priori* counterpart because of their increased generality. Ideally, a filter should be able to denoise all thinkable material and lighting conditions successfully. *A priori* filters are limited in this generality, targeting one or more specific light transport effects. Moreover, the implementation is again more practical, since likely a lot less intrinsic information has to be extracted from the renderer in order for the filter to perform its analysis.

A short list of the most promising state-of-the-art *a posteriori* post-processing filters, accompanied with descriptions of their workings, effectiveness, and runtime, is provided in the next section.

3.2 STAR filter selection

Random parameter filtering (RPF) The first idea for denoising path traced images would be naïvely applying general denoising filters targeted at for example photography, but this has often proven ineffective. Sen and Darabi [SD12] noted that this is the case because these filters assume that the noise is uniformly divided over the image, while Monte Carlo renders generally have large local noise variations. Furthermore, they reason that noise is a result of the random parameters used in Monte Carlo sampling, e.g. the random selection of surface bounce directions. They propose an approach that filters samples rather than pixels, and assigns variable weights to a pixel's neighbouring samples for the filtering, depending on the amount of similarity between the samples. Sample similarity is calculated by comparing their random parameters: samples with parameters mostly randomized over the same domains are assigned higher weights since they are both trying to approximate the same integral to a certain degree. In contrast, samples with low random parameter similarity are assigned lower weights. To calculate random parameter similarity a lot of feature information is needed: the normal, position, and texture for the first and second scene intersection of every sample. This approach yields impressive results, especially for low sample counts. However, it is also slow to compute, in the order of hundreds of seconds.

Park et al. [Par+13] propose a more efficient method, that makes use of the same principle, but doesn't compare all samples between a pixel and its neighbours, to greatly reduce compute complexity. Instead, only the samples between roughly a tenth of all pixels are compared, and all other pixel samples are interpolated. This technique reduces computation time significantly, while retaining nearly identical denoising performance. Although still too slow for use in a product configurator, ideas of their algorithm could be incoporated in an interactive filter.

General image denoising (GID) Kalantari and Sen [KS13] decided to keep exploring the possibilities of general denoising filters and proposed a method that allows effective use of them on Monte Carlo renders. Their approach is also based on the notion that these filters assume uniformly divided noise, requiring adaptation to the noise variations in Monte Carlo renders. First, they create a histogram of pixel noise levels of the entire render, and choose a small set of noise levels that encompasses most of the render's noise densities. For each chosen noise level, a copy of the render is filtered with a filtering intensity appropriate for this noise level, resulting in a set of images filtered with different intensities. The final filtered render is created by choosing the best filtered result for each pixel individually.

Bauszat et al. [Bau+15] generalized this approach by not only using one filter with different intensities, but by also using different types of filters. First they apply all filters separately, then they estimate the per-pixel noise error of the filtered results, and finally they combine the results to one optimal image. Their per-pixel filter choosing algorithm also favors the filters chosen for neighbouring pixels, as continually using different filters for neighbouring pixels reduces smoothness of the final image.

As used by both methods, state-of-the-art general image denoising algorithms are the *Non-Local Means* (NL-means) filter by Buades, Coll, and Morel [BCM05], the *Block-Matching 3D* filter (BM3D) by Dabov et al. [Dab+06], and the *Bayesian Least Squares - Gaussian Scale Mixture* filter (BLS-GSM) by Portilla et al. [Por+03].

Both implementations run in the order of tens of seconds on the CPU, but might be suitable for interactive needs when implemented on a GPU.

Denoising using feature and color information (DFC) Where RPF uses features (normal, position, texture) as filter input, and GID uses pixel colors as input (since they make use of general image denoising algorithms), Rousselle, Manzi, and Zwicker [RMZ13] propose a new filter in which both types of input are used to denoise renders more robustly. They reason this could be beneficial since some image details are only captured well by features, while others are only captured well by pixel color. Their method applies three filters separately: one filtering on color information alone, one filtering on feature information alone, and one that uses both. For the color filtering the NL-means filter is used. For the feature filtering an approach similar to RPF is used, but instead of feature filtering per-sample, or feature filtering per-sample with interpolation, the filtering is applied per-pixel to reduce compute complexity even more. After the filtering, the three filtered renders are combined on a per-pixel basis, using noise estimation to determine the best candidate, similar to GID.

The authors of this approach implemented the filter on the GPU, where it took only a couple of seconds to execute. Optimizing this filter might render it fast enough to be used in the product configurator, when using a high-performance GPU.

Ray histogram fusion (RHF) Delbracio et al. [Del+14] reason that general image denoising algorithms, like in GID, often have trouble preserving image detail simply because only using pixel color information does not always provide sufficient information for high quality denoising. Using features on the other hand, like RPF, while yielding good denoising results, currently takes too much computation time and memory to be used in interactive applications. They propose a novel method, wherein instead of using pixel color information, they use sample color information. With this approach they aim to increase denoising quality over general image denoising algorithms, while still retaining the short computation time of such an algorithm. First, they create color histograms of each pixel, using the color samples of each pixel as input data. Next, they compare the pixel color histograms to neighbouring pixel color histograms. If two pixel's histograms are found to be similar, their samples get shared between the pixels, smoothing out noise. For comparing histograms, a chi-squared distance threshold is used. They apply their filter multiple times at different scales. When filtering at a higher scale, the image simply gets upsampled before filtering, i.e. a patch of pixels (with their accompanying samples) gets averaged into one pixel with accompanying sample set.

They implemented their filter on the CPU, where it achieved runtimes in the order of tens of seconds, reaching their goal in regard of achieving roughly the same level of overhead as denoising filters that make use of general image denoising algorithms.

Szeracki et al. [Sze+15] expanded upon this approach, by implementing an optimized GPU version of RHF. Their version runs in a couple of hundred milliseconds, proving that this filter is suitable for interactive applications.

Learning based filtering (LBF) Kalantari, Bako, and Sen [KBS15] conclude that all previous approaches tend to perform as promised in certain cases, but perform subpar in others. They reason this is the case because these approaches failed to model the relationship between Monte Carlo noise and filter parameters correctly enough to produce a filter that provides satisfactory results on a wide variety of scenes. Furthermore, they reason that modeling this relationship correctly is too complex to model explicitly. They propose the idea of using a supervised learning algorithm to learn this relationship. To do this, they train a nonlinear regression model, using a neural network. For the training they use a set of noisy renders, with accompanying converged renders as ground truth. From every render a large number of features (square patches of pixels) are extracted to train the neural network on. Using the ground truth features as reference, the neural network learns rules for modifying the noisy features to get as close as possible to the ground truth features as possible.

The results in their proposal prove that their neural network learned to filter efficiently, outputting high quality denoised images. They implemented the filter on the GPU, where it took some seconds to execute. If their current implementation leaves room for significant optimizations, runtimes might be able to be reduced enough to allow for interactive filtering, in conjuction with using a high-performance GPU. The learning process was implemented on the CPU, and took almost a day to complete. The obvious downside of LBF is that it has to be trained before being ready for use, and if the trainig set is not representative of the renders used for filtering, the filter won't work optimally.

First-order regression for denoising (FOR) Bitterli et al. [Bit+16] introduce a Nonlinearly weighted First-Order Regression model for filtering. They analysed state-of-the-art denoising algorithms, and based on these findings, proposed a model that should combine their strengths, while avoiding their weaknesses.

DFC and LBF use prefiltered features as input. The prefiltering is done by applying NL-means to the feature buffers. They found this to significantly improve denoising performance and therefore also incorporated it in their model.

All previously discussed filters use zero-order regression models, comparing single pixel or sample colors and features. Sometimes this results in artifacts, when a single large distance in a certain feature buffer prevents the pixels to be combined, like for example could happen on the constantly changing normal value of a curved surface. Using a first-order regression (used in some recent adaptive filters), linearly changing feature buffers can be captured in the regression, allowing pixels or samples to still be combined where doing so is desirable.

Zero-order filters RPF, DFC, and LBF use NL-means to compute filtering weights, whereas previous first-order filters do not. The zero-order filters still often provide exceptional denoising results, and they reason this is partly because of the effectiveness of using NL-means to compute filtering weights. Therefore they propose to compute the weights for their first-order regression with NL-means also.

DFC and GID apply several of the same filter passes with different parameters, and then combine these for optimal results. They find this method, collaborative filtering, to be very effective and implement it in their model.

Being based on the findings of previous filters, this filter provides exceptional denoising results. However incorporating all these 'features' also comes at a cost: their CPU implementation runs slow, roughly at the same speed as RPF, with a GPU implementation roughly halving the required computation time.

3.3 Summary

The discussed denoising filters are summarized in the table below. CPU runtimes are adjusted to represent the theoretical speed on a recent Intel CPU with 8 threads. GPU runtimes are adjusted to represent the theoretical speed on a Nvidia GTX 1080. Complexity refers to the amount of features considered when filtering. These features are all needed as input for the corresponding filters. Higher amounts of features needed for filtering result in a higher impact on path tracing performance, reducing the amount of samples that can be traced by the path tracer within a given amount of time. All filters have a computation time dependent on the number of pixels in the image. Filters using pixels as input have a constant computation time otherwise. Filters using samples as input have a computation time that is also dependent on the number of samples traced.

Filter	**CPU**	**GPU**	**Complexity**
RPF	100s	unknown	Samples: Colors, normals, positions, textures, 1^{st} bounce normals, 1^{st} bounce positions
GID	10s	unknown	Pixels: Colors
DFC	unknown	2s	Pixels: Colors, normals, textures, depth
RHF	50s	100ms	Samples: Colors
LBF	unknown	3s	Pixels: Colors, coordinates, normals, positions, textures, illumination, 1^{st} bounce textures
FOR	200s	100s	Pixels: Colors, coordinates, normals, depth, textures, illumination

All discussed filters promise high-quality denoising. These assumptions are however based on the examples given in their corresponding publications. It is not unthinkable that the examples given in these publications are chosen specifically to show the strengths of the proposed filters. Therefore the filters could perform worse in our application than they are implied to. The only way to confirm this is to implement or integrate (if a public implementation is available) a filter ourselves to study its performance.

RHF on the GPU is the only filter with a measured runtime that is suitable for use in an interactive path tracer. DFC and LBF need significant GPU optimizations to be suitable. GID is probable to reach desired runtimes after a GPU implementation: most of its runtime is used to execute general image denoising algorithms, of which efficient GPU implementations are often publicly available and documented. RPF and FOR are likely impossible to be made suitable for interactive filtering, but ideas from their methods could be leveraged to improve an interactive filter.

We implement RHF as a reference filter, being the only state-of-the-art filter we found to be suitable for interactive filtering. For the following iterative steps of improving upon this implementation, we consider all filters and their proposed methods, in an interactive context.

Chapter 4

Implementation

4.1 Ray Histogram Fusion

RHF is divided in two phases: the accumulation of sample colors in histograms during rendering, and the post-process merging of pixels with similar histograms.

Accumulating sample color histograms For each pixel an empty histogram is created. The histograms contain three sets of bins, one for each channel of the RGB color space. Each set of bins ranges from the lowest possible color value (0), to an arbitrarily chosen highest value (7.5). The number of bins and their sizes have to be chosen in advance. We used 20 bins per channel per pixel, and bin sizes increasing exponentially with an exponent of 2.2. Each sample is inserted in their corresponding histogram, by calculating for each channel in the sample to which bin it belongs, and then incrementing the counter of that bin by one.

Filtering This step is computed once as a post-processing filter. For each pixel, its histogram is compared to the histograms of neighbouring pixels within a prespecified window size. Histograms are compared using a *chi-squared distance threshold*. If this distance is below a set user parameter, the pixels will be merged. The rationale behind this, is that if two sample histograms are similar, the pixels geometric, texture, and shading properties are likely similar, warranting a merge for these pixels to increase the effective sample count, removing noise (figure 4.1).

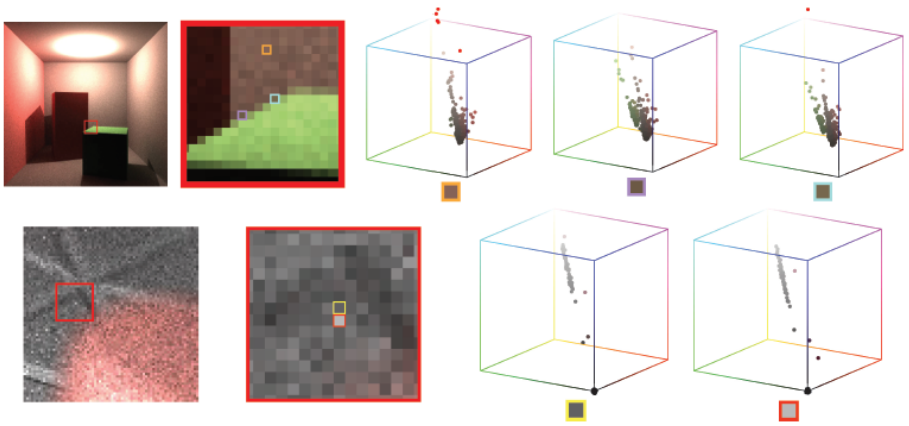

FIGURE 4.1:
Top: The first pixel's histogram distribution is unimodal. The other pixels lie on an edge and their histogram distributions are both bimodal, and therefore only these other two pixels should merge with each other. Bottom: These pixels differ considerably in color, but have similar histograms, implying they will converge to the same color, and thus should merge.
Source: [Del+14]

RHF implementation and results We fully implemented RHF, to check its denoising quality. We were pleased with the results for the most part. Unfortunately, in specific cases, the results were disappointing. In low color contrast areas of a render, for example on wooden surfaces with distinctive wood grain textures (figure 4.2), RHF merged too many pixels, blurring all fine details.

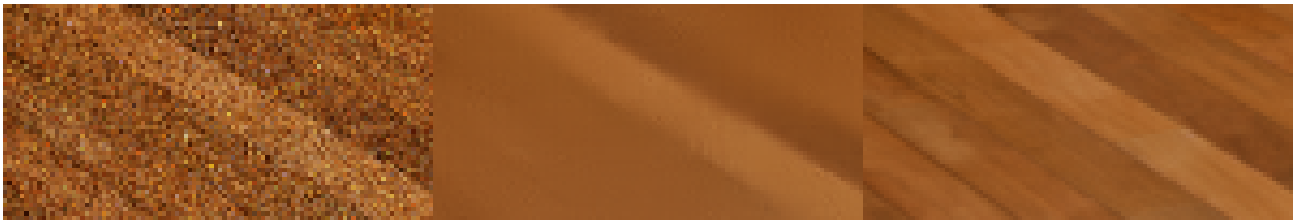

FIGURE 4.2: Left: 256spp input, Middle: RHF, Right: reference.

We concluded the reason for this failure case was the fact that most samples belonging to this surface were contained in the same bin, as their color values were very similar and the bins can only distinguish between 20 different color values per channel. Only after the bin count increased tenfold, the problem began to disappear. However, since the chi-squared distance compute time scales linearly with bin count, this solution is unfeasable due to the massively increased filter time.

We explored the option of using different bin sizes, decreasing bin size within the color regions where fine detail denoising was needed, effectively increasing the bin count only where it would improve denoising performance. The problem with this approach is however, that every render requires a different binning strategy, with bins concentrated in different color regions. Furthermore, the binning strategy must be known before rendering starts, as it is otherwise impossible to determine during rendering in which bin a sample should belong. After testing, we found automatically computing a succesful binning strategy beforehand to be too unreliable.

We found that where RHF failed to discriminate between two pixels where it should have, the texture color (albedo) or geometry normal values would often differ significantly between these two pixels. After incorporating distance calculation for these two features alongside the chi-squared metric, the problem was solved (figure 4.3). However, disabling the chi-squared metric and keeping the two features only to calculate pixel distance, we noticed only a marginal decrease in denoising performance, alongside a large improvement in filter computation time. This led us to believe RHF was too ineffective for our needs, and a fleshed out feature-based filter could provide far better denoising results. This led us to the development of our novel *Feature Distance Filter*, described in full detail in the next section.

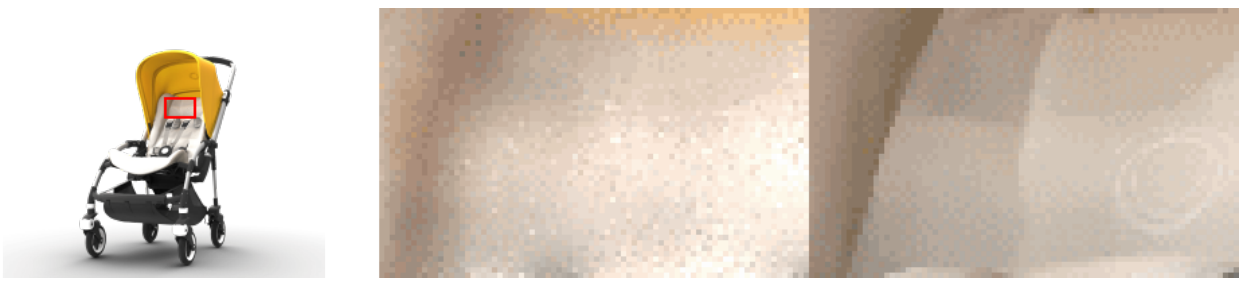

FIGURE 4.3:
Middle: RHF overblurs low-contrast details.
Right: Detail is preserved when leveraging texture and normal features.

4.2 Feature Distance Filter

FDF is our novel filter implementation, and greatly based on the notion that filtering based on feature buffers as input is very effective [RMZ13], [KBS15], [Bit+16]. As input for our filter we make use of three feature buffers: texture features, normal features, and our own novel shading features. The shading feature buffer is noisy, and therefore we prefilter it using non-local means (NL-means) [BCM05], based on the prefiltering step in the work by Bitterli et al. [Bit+16]. We improve the prefiltering by extending the algorithm to become adaptive in strength based on the noisy input image's sample count, and by reducing computation complexity while still retaining largely the same denoising quality. Using these three features, we filter the noisy image by means of our own variant of the bilateral filter based on the work by Rousselle, Manzi, and Zwicker [RMZ13]. To our knowledge, we are also the first to implement a feature-based filter that ignores the input image's color feature buffer completely, proving it is unnecessary as a filter weight. A schematic overview of the filter's workings is shown in figure 4.4.

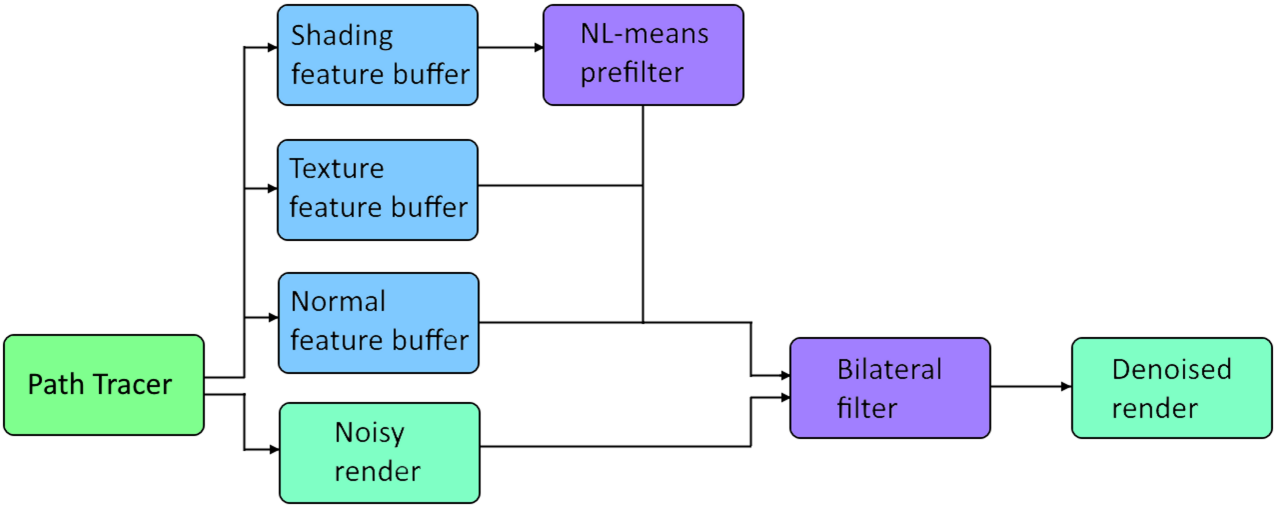

FIGURE 4.4: A schematic representation of FDF.

Texture and normal feature buffers We modified the path tracer to allocate three feature buffers before rendering: For the texture and normal features, we allocate two RGB color space buffers, and for the shading features we allocate a single-channel grayscale buffer. While rendering, the feature buffers' values are generated by the path tracer. For each color sample the path tracer calculates for a pixel, we accumulate corresponding feature values in the feature buffers. For the texture feature buffer we accumulate the unshaded albedo value of the surface that was hit by the primary ray of each sample. We do the same for the normal feature buffer, but use the world-space normal value of the surface. Since world-space normal axis component values fall in the range $[-1, 1]$, we apply $x = (x + 1)/2$ to each axis component to map to the visible RGB color range of $[0, 1]$. Primary rays are always traced through the area of the pixel they are supposed to render; their directions are not dependent on random bounces. Therefore these buffers are not noisy, nor aliased, thus not needing any prefiltering. An example of the texture and normal feature buffers is provided in figure 4.5.

FIGURE 4.5: From left to right: 800ms noisy input, texture feature buffer, normal feature buffer, and denoised shading feature buffer.

Using only texture and normal features as weights for a bilateral filter provides surprisingly good denoising results. However, the denoised image still loses detail in image regions where differences in reference image color are not captured well by the features. This mostly manifests itself in lack of (correct) lighting effects, such as shadows, color bleeding, global illumination, ambient occlusion, and translucency. An example of this lack of lighting detail is shown in figure 4.6. Most filters try solving this by utlizing the (noisy) input color buffer in some form for filtering [Bau+15], [Del+14], [KBS15]. Rousselle, Manzi, and Zwicker [RMZ13] propose the visibility buffer, which discriminates whether pixels receive direct lighting or not. Bitterli et al. [Bit+16] apply a first-order model to the input to try and solve this problem. All options however, have shortcomings and weaknesses in one or more areas, and in our opinion, perfectly reconstructing all lighting effects in all possible situations is the most difficult aspect of creating a denoising filter.

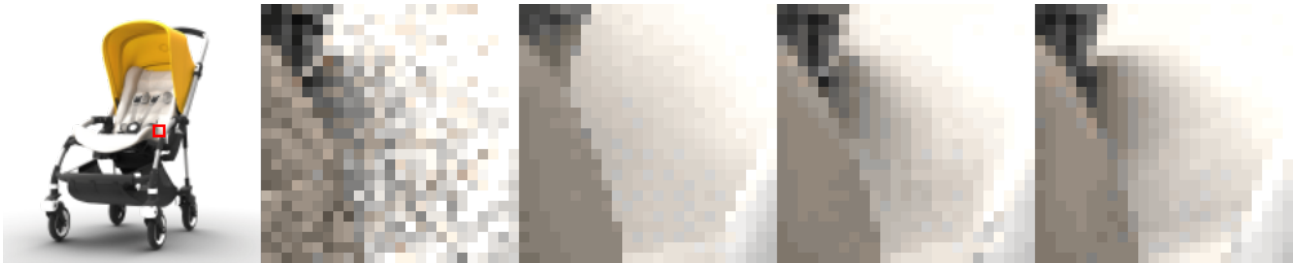

FIGURE 4.6: From left to right: 800ms noisy input, denoised using texture and normal feature buffers only, denoised with the addition of our shading feature buffer, reference.

Shading feature buffer To try and solve the problem of denoising while preserving lighting details, we propose a new method utilizing our novel shading feature buffer. Our reasoning is that, having noted that denoising using feature buffers is an effective strategy for preserving detail, this denoising approach could also be successfully applied to preserve lighting detail, if a feature buffer could be generated that captures lighting information well. With the shading feature buffer, we try to achieve this goal. Figure 4.6 shows an example of the type of detail our shading feature buffer can preserve, and figure 4.5 shows an example of a (denoised) shading feature buffer.

Generating the shading feature buffer is more complicated than generating the texture and normal feature buffers. Where these buffers take values from each sample's primary ray hit, we take values from the endpoint of each sample's fully traced lightpath for the shading feature buffer. Every lightpath ends by one of two possible means: either the path gets terminated by *russian roulette*, having bounced around the scene for too long, or it reaches a light source and terminates there. The light source can be a distinct light (e.g. a lightbulb), or when a lightpath reaches the skybox of the scene, this skybox is used as a lightsource. If a lightpath gets terminated, a value of 0 is accumulated to the shading feature buffer. If a lightpath reaches a lightsource, the intensity of the lightsource is accumulated to the shading feature buffer. Using this method, we can successfully approximate the amount of light that reaches each pixel, regardless of whether the light reached that pixel directly, or indirectly via for example color bleeding or through a translucent material.

Downside of this approach however, is that the feature buffer is noisy. Because the value accumulation relies on the outcome of random bounces and *russian roulette*, each sample generates a different lightpath, resulting in a different value added to the feature buffer. Therefore, we have to denoise the shading feature buffer before it can be used as a weight for the bilateral filter. Otherwise, the noise present in the shading feature buffer would be propagated through the bilateral filter weights to the final denoised image. We denoise the shading feature buffer using a modified version of NL-means. An example of the denoising is shown in figure 4.7.

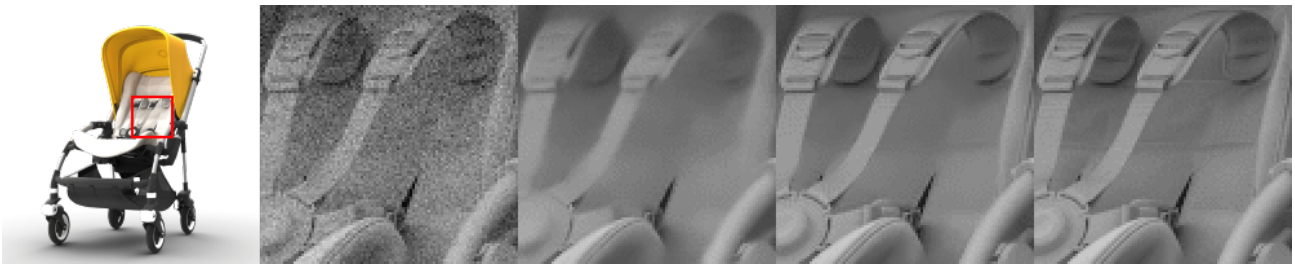

FIGURE 4.7: Shading feature buffers from left to right: 800ms noisy input, 800ms denoised, 8.000ms denoised, 80.000ms denoised.

NL-means denoising Given the noisy shading feature buffer, which is a single-channel greyscale image, a denoising algorithm is needed that works exclusively in image space. Therefore we look in the class of general image denoising algorithms. In recent work, the most effective general image denoising algorithm for monte carlo images has already been elected: [RMZ13], [KBS15], and [Bit+16] all use NL-means in some form for prefiltering, and [Bau+15] even use NL-means as a main filtering step. Therefore, we chose to use this algorithm for prefiltering too.

NL-means, proposed by Buades, Coll, and Morel [BCM05] in 2005, is still state-of-the-art for general image denoising today. It computes the final output for a pixel as the weighted sum of a set of pixels contained in the region situated around the input pixel. This also applies to bilateral filters, but with the difference that with NL-means, the weight calculation between two pixels is not only determined by these two pixels, but by two patches of pixels, centered around the two pixels to be compared. By comparing patches instead of pixels, the influence of a single pixel's noise component greatly diminishes, allowing for much improved denoising performance. An example of NL-means is shown in figure 4.8.

NL-means computes the filtered value $f(p)$ of pixel p as the weighted average of pixels in a square neighbourhood $N(p)$ centered at p of search window size $(2r+1)^2$, with weight $w(p,q)$ as the weight between pixel p and pixel q.

$$f(p) = \frac{1}{W(p)} \sum_{q \in N(p)} q * w(p,q)$$

Where $W(p)$ is a normalization term, scaling the final value of p by the total amount of weight.

$$W(p) = \sum_{q \in N(p)} w(p,q)$$

The weight $w(p,q)$ is calculated as the average weight between all equal-offset pairs of pixels contained in the patches $P(p)$ and $P(q)$ centered at pixels p and q of patch size $(2s+1)^2$, with $O(n)$ denoting the offsets for all pixels within a patch, relative to the pixel centered at that patch.

$$w(p,q) = \frac{1}{(2s+1)^2} \sum_{n \in O(n)} d(p+n, q+n)$$

Where $d(p+n, q+n)$ is the squared difference in color value v (or grayscale value) between the pixels at $p+n$ and $q+n$, transformed to return a weight of 1 for a distance of 0, and exponentially decreasing in weight for a linear increase in distance.

Even though patchwise filtering greatly reduces bias that is introduced by comparing distances between noisy pixels, every pixelwise comparison in itself is still significantly biased, due to the noise component in both pixels. To cancel out the noise contribution as much as possible, a variance term is subtracted from the calculated distance. The original NL-means algorithm targets general image denoising, where the pixel variance (noise component) is uniform. Assuming uniform variance σ, the distance metric can be improved by substracting $2\sigma^2$ from the pixel values' squared difference.

In Monte Carlo path traced images however, variance varies greatly between pixels, depending on a multitude of different scene properties like geometry shapes, light locations, and material properties. Therefore the distance metric between pixels $p+n$ and $q+n$ can be further improved by canceling out the variance values per pixel, which gives us our final distance metric implementation:

$$d(p+n, q+n) = \exp\left(-\frac{(v(p+n) - v(q+n))^2}{\epsilon + Var(p+n) + Var(q+n)}\right)$$

Where $Var(p+n)$ and $Var(q+n)$ are the variance of pixels $p+n$ and $q+n$, and ϵ is an extremely small constant value that prevents any possible division by 0.

To obtain variance values for all pixels, we allocate a second single-channel grayscale buffer, which we accumulate parallel with the shading features buffer. Only for the variance buffer, we accumulate squared values of the traced light intensity. We can then calculate $Var(p)$, knowing that variance is the average of the squared differences from the mean:

$$Var(p) = (p_v/spp) - (p_s/spp)^2$$

Where p_v is the variance buffer value of p, p_s is the shading buffer value of p, and spp is the number of samples the buffer accumulated per pixel.

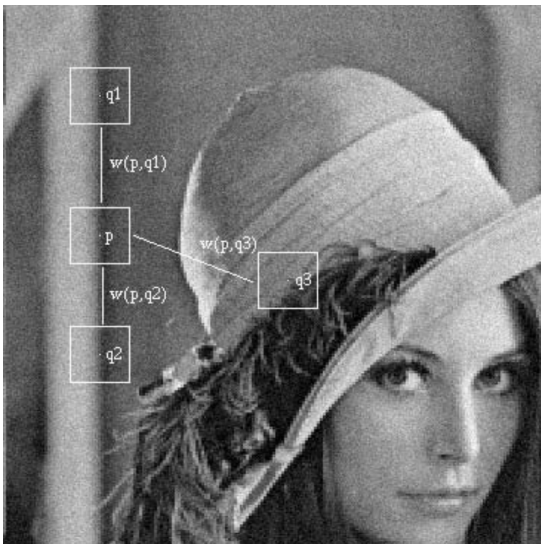

FIGURE 4.8: NL-means patchwise denoising: weights $w(p, q1)$ and $w(p, q2)$ for similar patches are high, allowing the pixels centered at patches $q1$ and $q2$ to contribute significantly to the denoising of the pixel centered at p, while weight $w(p, q3)$ is low, ensuring that any contribution made by the pixel centered at $q3$ will be very small or even non-existent. Source: [BCM05]

For best performance, we implemented NL-means on the GPU. Since the GPU excels in handling massively parallel workloads, and NL-means has to perform the exact same instructions for every pixel in the shading feature buffer, the algorithm is a good fit for the GPU, speeding up computation time considerably relative to a CPU implementation. We also made a number of performance to quality considerations:

- For the window size r, we used a value of 5, and for the patch size s, we used a value of 2. Usually recommended values are around 10-20 for the window size and 5-7 for the patch size. Altough larger values increase the denoising quality of NL-means, we found 5 and 2 to provide the best computation time tradeoff, while still providing good denoising results. The computational complexity of NL-means is $(2r + 1)^2 * (2s + 1)^2 * n$ (with n being the number of pixels in the image); reductions in either window or patch size parameters influence the algorithm's runtime significantly.

- For the shading feature buffer, we opted for a single-channel grayscale buffer instead of a RGB buffer, storing the light's intensity value only, instead of the light's full color value. We found this simplification to induce an unnoticeable difference in denoising quality, while speeding up the NL-means algorithm 3x. A distance calculation now has to consider one channel only, instead of repeating the calculation process for all three RGB channels.

- We chose not to incorporate some more complicated variance estimation methods after observing only limited quality improvements, at the cost of significantly reduced performance. These are: using two independent variance and shading buffers to cross-filter one with the variance values of the other and vice versa to further reduce variance bias, and using a dual-parameter variance cancellation term that clamps variance to the lowest between pixels p and q to prevent overblurring. Both are explained in more detail in [Bit+16].

As the input image's sample count increases, the amount of noise present in the accompanying shading feature buffer decreases. To prevent NL-means from overblurring on higher sample count images, we divide the variance estimate by the sample count. This results in a linear reduction of the filters power in relation to sample count, which we found to be the optimal relation. The effect this power reduction has on denoising is visible in figure 4.7.

Weighted bilateral filter The main and final step of FDF employs a weighted bilateral filter. A weighted bilateral filter is essentially a simpler version of the NL-means filter: they are identical in functioning, except that with the weighted bilateral filter, the weighting function $w(p, q)$ works purely on a per-pixel basis, instead of taking patches of pixels into account like NL-means does. The weights for our bilateral filter are the three feature buffers: texture feature buffer t, normal feature buffer n, and shading feature buffer s. We calculate the distance between pixels p and q as the largest distance of the three feature distances, and apply the same exponential function as with NL-means to transform the distance to a weight.

$$w(p, q) = \exp(-\max(d(p_t, q_t), d(p_n, q_n), d(p_s, q_s)))$$

The distance function $d(p_f, q_f)$, with feature buffer f takes the largest distance of the feature buffer's channels c first, and then normalizes the distance using a user-specified *maximum feature distance* parameter f_{max}.

$$d(p_f, q_f) = \frac{\max\limits_{i \in [1...c]} (|p_f^i - q_f^i|)}{f_{max}}$$

For the texture and normal feature buffers, c describes a three-channel RGB buffer, and for the shading feature buffer, c describes a single-channel grayscale buffer. The maximum feature distance parameters f_{max} were chosen empirically: we found $t_{max} = 0.03$, $n_{max} = 0.06$, and $s_{max} = \frac{5}{spp}$ to work best. The sample count variable spp is employed for the same reason as why it is employed in the NL-means prefilter: to extend the filter to adapt to variable sample counts. As the sample count increases, the measured shading feature buffer distances increase too. On noisy images with a low sample count, this extension allows FDF to still remove all visible noise, albeit at a loss of fine detail. On relatively converged images with a high sample count however, the increased shading feature buffer distances preserve the image's fine detail, while still allowing just enough leeway to remove the weaker noise left-over in a higher sample count image. An example of this is shown in figure 4.9.

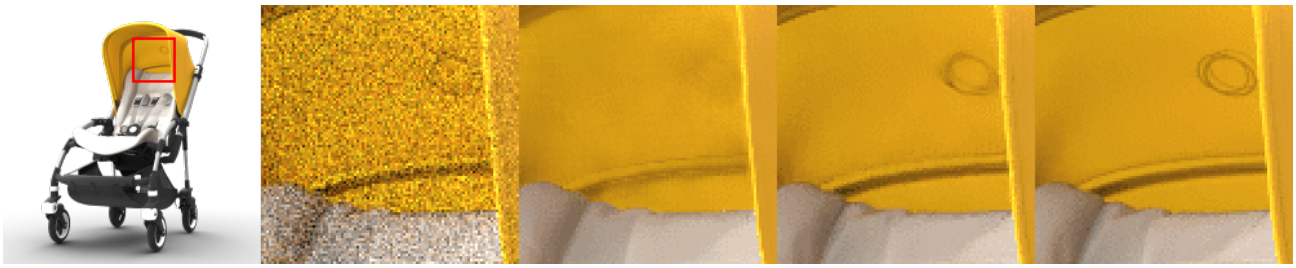

FIGURE 4.9: Denoising of a very difficult translucent material, with a logo on the other side of the material. As sample counts increase, noise decreases, and FDF becomes able to discriminate the logo's fine details. From left to right: 800ms noisy input, 800ms FDF denoised, 8.000ms FDF denoised, 80.000ms FDF denoised.

We implemented the weighted bilateral filter on the GPU too, for the same reasons as we did with NL-means. For the window size for the weighted bilateral filter, we used a value of 8. Here, values of 10-20 are recommended again, however here too the algorithm's total runtime depends on the window size. Because the per-pixel evaluations of the weighted bilateral filter require less computation time than the patchwise evaluation of NL-means, we could afford to use a higher value of 8 instead of the lower NL-means value of 5.

Computational and memory overhead We measured runtimes and memory usage of FDF on a machine equipped with a Intel Core i7-7700K, Nvidia GTX 1080, and 32GB of RAM. FDF's runtimes scale almost perfectly linear with the amount of pixels in the image to be denoised. A graph of runtime against pixel count is displayed in figure 4.10. Important datapoints have been highlighted: the commonly used resolutions 720p and 1080p, and a typical product configurator resolution: 800x800.

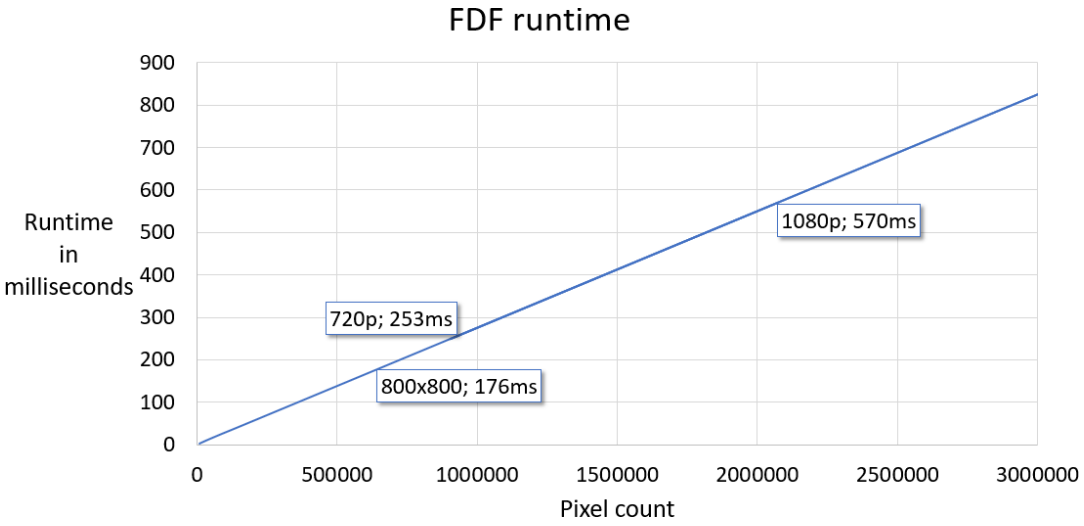

FIGURE 4.10: FDF's runtime in milliseconds.

Runtime is dominated by NL-means, making up 90% of the total time, of which 80% is spent evaluating distance metrics. Only 10% is used by the bilateral filter.

FDF requires 9 32-bit floats per pixel to be allocated in GPU memory (2 RGB feature buffers + single-channel NL-means shading input, output, and variance), thus FDF's memory overhead is 36 bytes per pixel, visualized in figure 4.11.

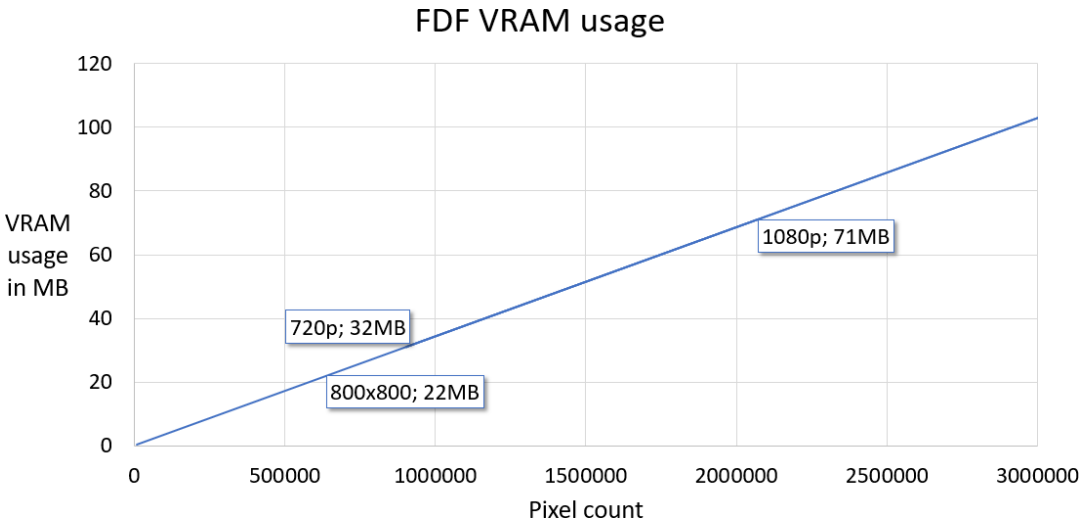

FIGURE 4.11: FDF's VRAM usage.

Chapter 5

Results

First we evaluate the absolute quality of the renders generated by the path tracer, integrated in the product configurator framework. Then, we evaluate the amount of time needed to render images of this level of quality, with and without a postprocessing filter. Finally, we evaluate the denoising performance and overhead of FDF, compared to the current state-of-the-art.

We use three product configurator scenes for the quality and interactivity evaluation. The scenes differ in complexity and material types, allowing the results to more accurately reflect the range of quality and performance achievable in a wide variety of scenes. The first scene is a worst-case scene of a high-complexity Bugaboo Bee^5 stroller using translucent materials for its canopy, which is typically difficult to path trace. The second scene is a comparable scene of the same product, but without a translucent canopy, to measure performance on high-complexity scenes without translucent materials. The third scene uses a low-complexity model of a Prominent Pure luxury armchair, to measure performance on a more common scene of average difficulty. Path traced renders of all three scenes are shown in figure 5.1.

FIGURE 5.1: Bugaboo Bee^5 transparent, opaque, and Prominent Pure.

5.1 Quality

On the next page, we show the quality evaluation on the most difficult scene, checking correct presence of all the effects targeted in evalution table 1.1. For each effect we have described where and how it is visualized in the path traced render. As can be seen, we can successfully render the image, visualizing all requested graphical effects, and showing no defects or glitches. Also, no significant differences between the rasterizer and path tracer are present regarding material types, textures, and scene setup, showing that our path tracer integration is working correctly. The other scenes performed the same (5.1). We can therefore conclude that our integrated path tracer can successfully render images of high-quality.

Effect	**Example of visualization**	**Path tracer evaluation**
Indirect lighting	Underside of seat support frame no longer uniformly lit by ambient lighting	Underside seat support is now darker, since less light rays reach there
Color bleeding	Parts of the metal bars being lightly colored yellow due to color bleeding from the yellow canopy	Left upper metal bar shows yellow reflections
Contact shadows	Realistic darkening between two close surfaces, not approximated by ambient occlusion	Darkening on canopy near right upper metal bar's extension button
Soft shadows	Proper soft shadow on the ground cast by the stroller instead of the current blurred hard shadow	Smoothly dissipating, wide-range, soft shadow on the ground
Reflections	Correct reflections of the metal bars and chrome bolts instead of a screen space approximation	Chrome bolts visualize skybox and object reflections correctly
Anti-aliasing	No visible aliasing in the image due to pixel area random sampling	No aliasing, and of a better quality than the rasterization method
Translucency	Yellow light cast on the seat, transmitted through the yellow canopy	Upper part of white seat is lightly colored by a yellow glow

TABLE 5.1: Table of the requested effects, visualization examples, and visualization evaluation for the path tracer.

FIGURE 5.2: Left: Rasterizer, Right: Path Tracer.

5.2 Interactivity

We evaluate the amount of time required to render the aforementioned scenes at a resolution of 800x800 pixels to high quality (table 5.2). All measurements were taken on a machine equipped with a Intel Core i7-7700K, Nvidia GTX 1080, and 32GB of RAM. We found this evaluation to be impossible to compute exactly: the MSE at which a render becomes noise-free (for unfiltered renders) or bias-free (for filtered renders) to a human differs for every scene, and depends on whether the render was filtered not. Therefore we measured render times by human evaluation. The point at which we accepted a render was professionally determined by 3DIMERCE's 3D visual expert Huub van Summeren, aligning the point of acceptance with the quality requirements for using renders in a production environment.

Scene	Unfiltered	Filtered
Bee[5] transparent	38.4s	12.8s
Bee[5] opaque	38.4s	9.6s
Pure	19.2s	4.8s

TABLE 5.2: The required render times for achieving high-quality.

The results are far under performance target: For filtered renders, the render times are about 2-16x too slow to be marked as interactive. We also plotted graphs displaying the filter's MSE improvements over time in figure 5.3.

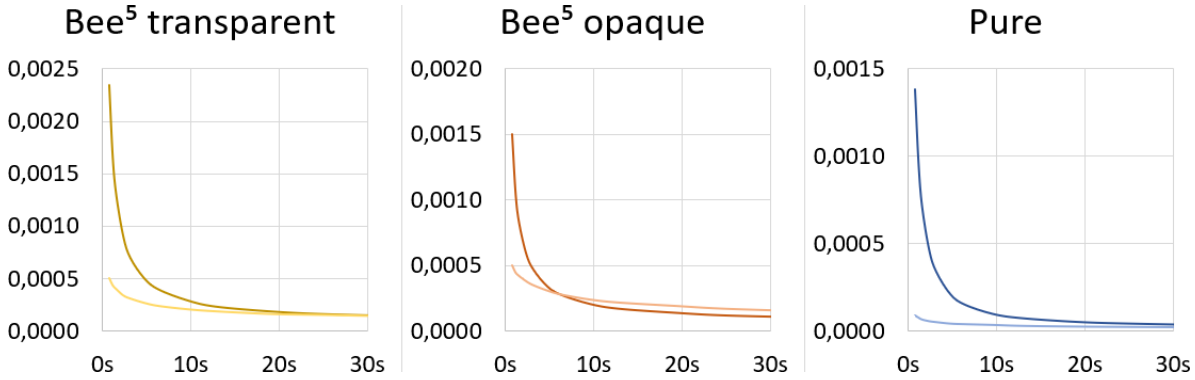

FIGURE 5.3: Dark colors: unfiltered MSE, light colors: filtered MSE.

Around the 1 second mark, we achieve significant MSE improvements. As time progresses, the relative improvement decreases: as the error of the unfiltered render decreases, less opportunity is left for the filter to improve upon its input. We achieve a larger improvement on the Pure scene due to its reduced complexity, containing less areas of fine detail for bias to appear. On the opaque Bee[5] scene, we lose quality starting at 6.4s, according to the MSE metric. However, the filtered render still converts all noise to bias. A human evaluation could find the filtered render to be of better quality (figure 5.4).

FIGURE 5.4: Opaque Bee[5] 6.4s equal MSE: input, filtered, reference.

5.3 Filter performance

We compare FDF's denoising performance against the current state-of-the-art in postprocessing denoising filters aimed at non-realtime reconstruction (16 to 1024 samples per pixel as input): the original *Non-Local Means* algorithm by Buades, Coll, and Morel [BCM05] applied to the pixel color output buffer on multiple scales (NLM), *Ray Histogram Fusion* by Delbracio et al. [Del+14] (RHF), *Robust Denoising Using Feature and Color Information* by Rousselle, Manzi, and Zwicker [RMZ13] (DFC), *Learning Based Filter* by Kalantari, Bako, and Sen [KBS15] (LBF), *Weighted Local Regression* by Moon, Carr, and Yoon [MCY14] (WLR), and *Nonlinearly Weighted First-Order Regression* by Bitterli et al. [Bit+16] (FOR).

We tested all filters on the sample counts 16, 64, 256, and 1024 on two scenes: San Miguel by Guillermo M Leal LLaguno, and Bedroom by SlykDrako. We found integrations of all state-of-the-art filters for the open source path tracer *pbrt*. San Miguel is included in pbrt, and a port of Bedroom to pbrt was made available by Benedikt Bitterli. Currently, FDF is only integrated in our own path tracer. We imported San Miguel and Bedroom into our path tracer to compare our results against the state-of-the-art filters. Due to differences between pbrt and our path tracer, small discrepancies regarding materials, objects, and tone mapping exist between the scenes. However, we did manage to replicate the lighting setup of pbrt exactly in our path tracer, allowing for a fair comparison. The exact scene setups used are displayed in figure 5.5.

FIGURE 5.5: Top: San Miguel, bottom: Bedroom, left: ours, right: pbrt.

5.6 contains the results for the San Miguel scene. All filtered images only contain bias and no noise, allowing use of the MSE metric for a reliable comparison.

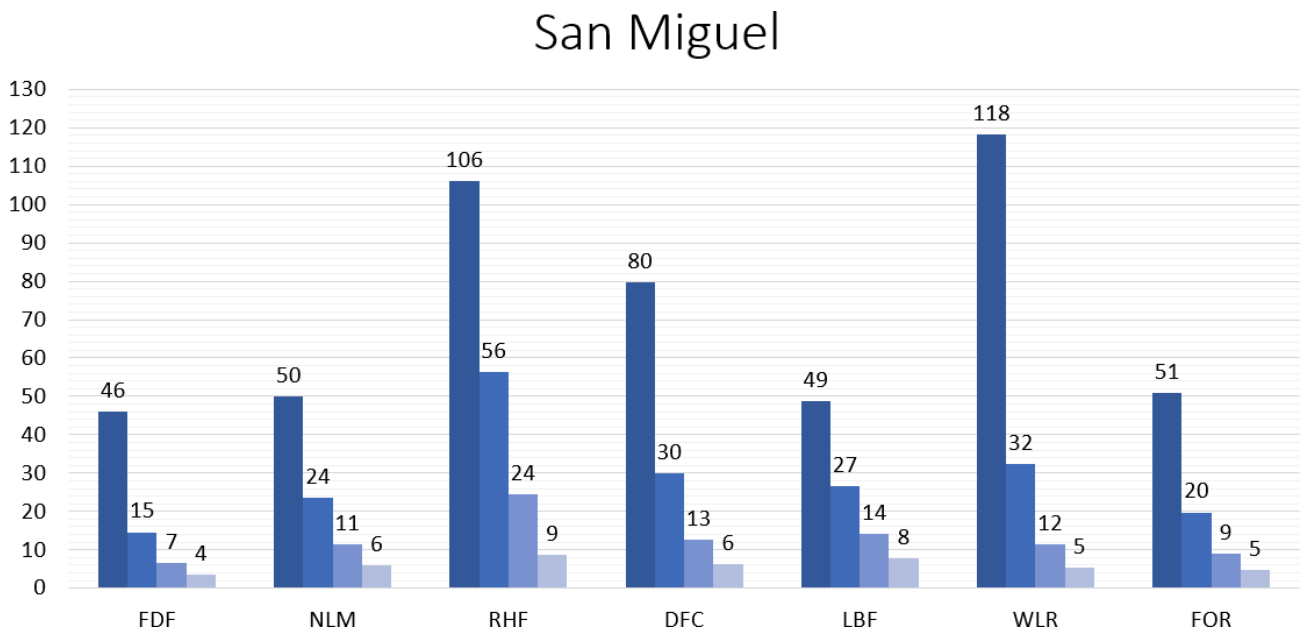

FIGURE 5.6: MSE of filtered renders of San Miguel.
Actual MSE values are multiplied by a factor 10.000 to increase readability.
Values from left to right represent sample counts 16, 64, 256, and 1024.

FDF achieves excellent results here, managing the best score on all sample counts, albeit with a tight margin in some cases.

The same test is repeated for the Bedroom scene, displayed in figure 5.7.

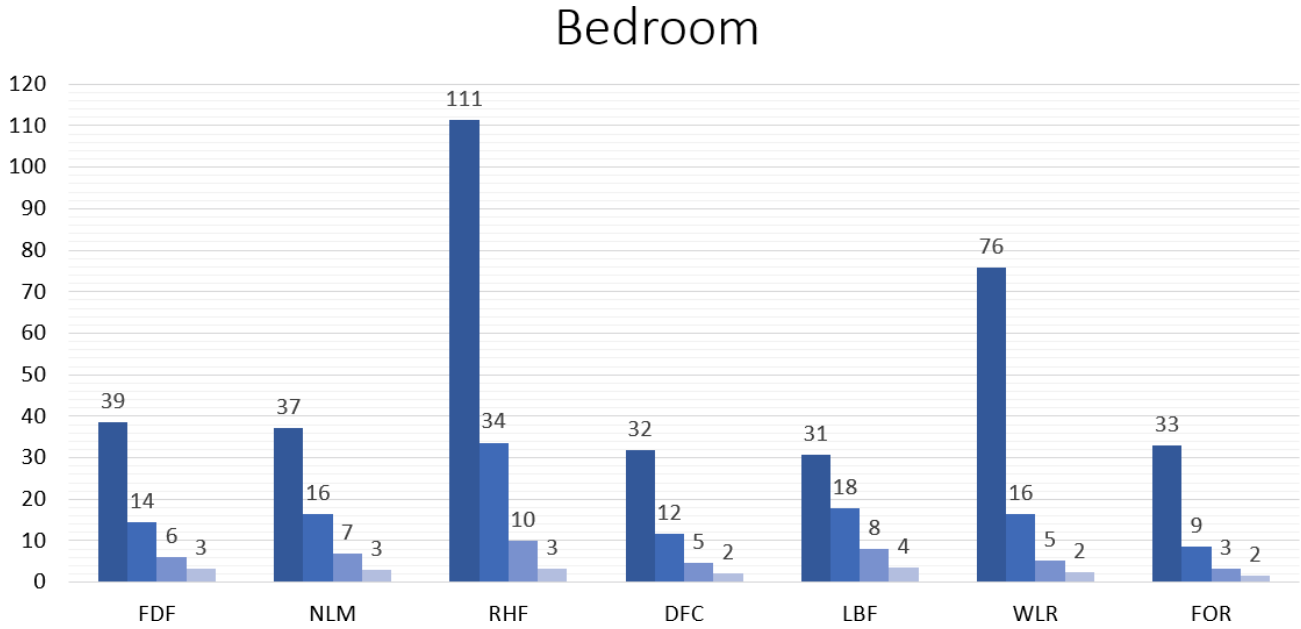

FIGURE 5.7: MSE of filtered renders of Bedroom. The same format applies as in figure 5.6.

Here, FDF only manages a mediocre result, scoring a fourth place on average. The gap between FDF and better scoring filters however, is not extreme.

The results above show that FDF can compete with the current state-of-the-art in terms of absolute denoising performance.

Next, we measured runtimes of all filters, displayed in figure 5.8. All implementations make use of the GPU. The test system is the same as before: Intel Core i7-7700K, Nvidia GTX 1080, and 32GB of RAM. The runtimes are of a single full denoising pass on a 720p image.

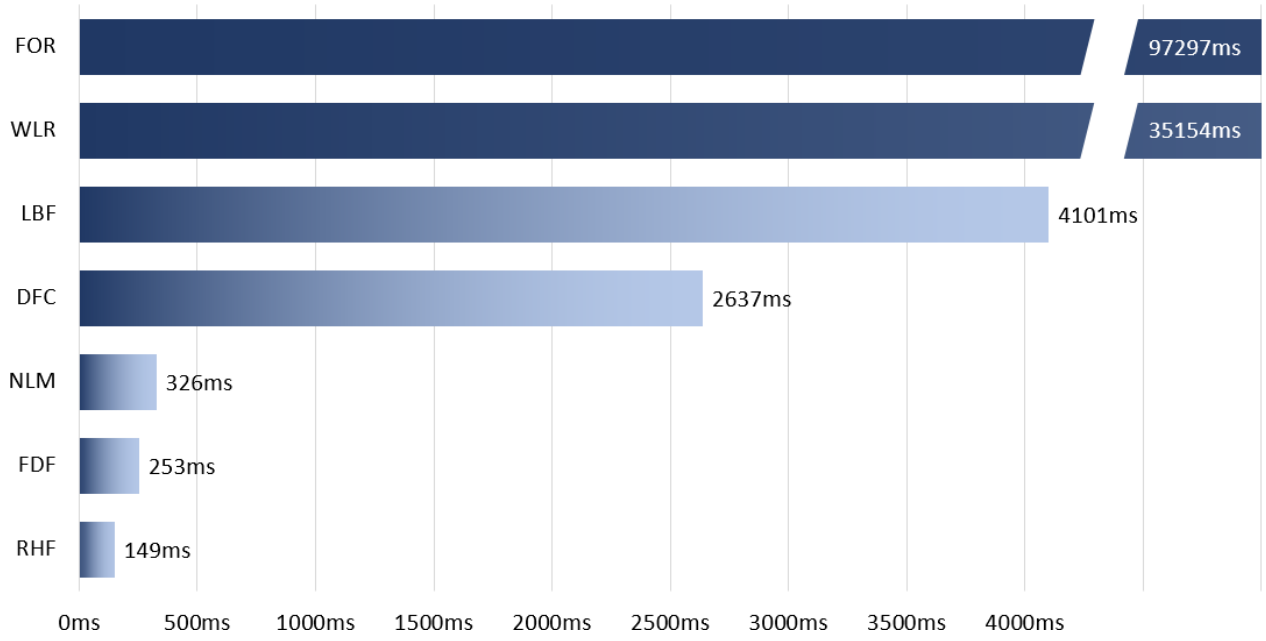

FIGURE 5.8: Runtimes of all filters in milliseconds. FOR and WLR contain a break, since their runtimes do not fit in the chart.

Here, FDF achieves outstanding results, managing a second place, receiving competition from NLM and RHF only.

Summary Purely from a denoising point of view, FDF, NLM, LBF, FOR, and marginally DFC score decent to good results. RHF and WLR stand out with subpar results. From a runtime point of view, FOR and WLR run for an extremely long amount of time, making them only suitable for multi-minute rendering. The same, altough to a lesser extent, applies to LBF and DFC, making these filters only suitable for non-interactive rendering. LBF also needs to be trained once before use, which took approximately 4.5 hours on our system. The results of training can be saved for reuse though. This leaves FDF with real competition from NLM and RHF only. During our own implementation, we have found RHF to perform subpar, which the state-of-the-art comparison tests confirms, showing RHF to perform significantly worse than other filters, especially on lower sample counts typically associated with interactive rendering. NLM finally, performs consistently slightly worse than FDF, while also requiring slightly more compute time.

Putting all of the above in perspective, we find our filter to be the best performer in this evaluation, when runtimes and denoising performance are both taken into account.

For visual reference, we compiled a set of images to compare the denoising performance of the various filters in figure 5.9. Note that FDF's images look slightly different due to small scene differences caused by different conversion and importing methods between pbrt and our own path tracer.

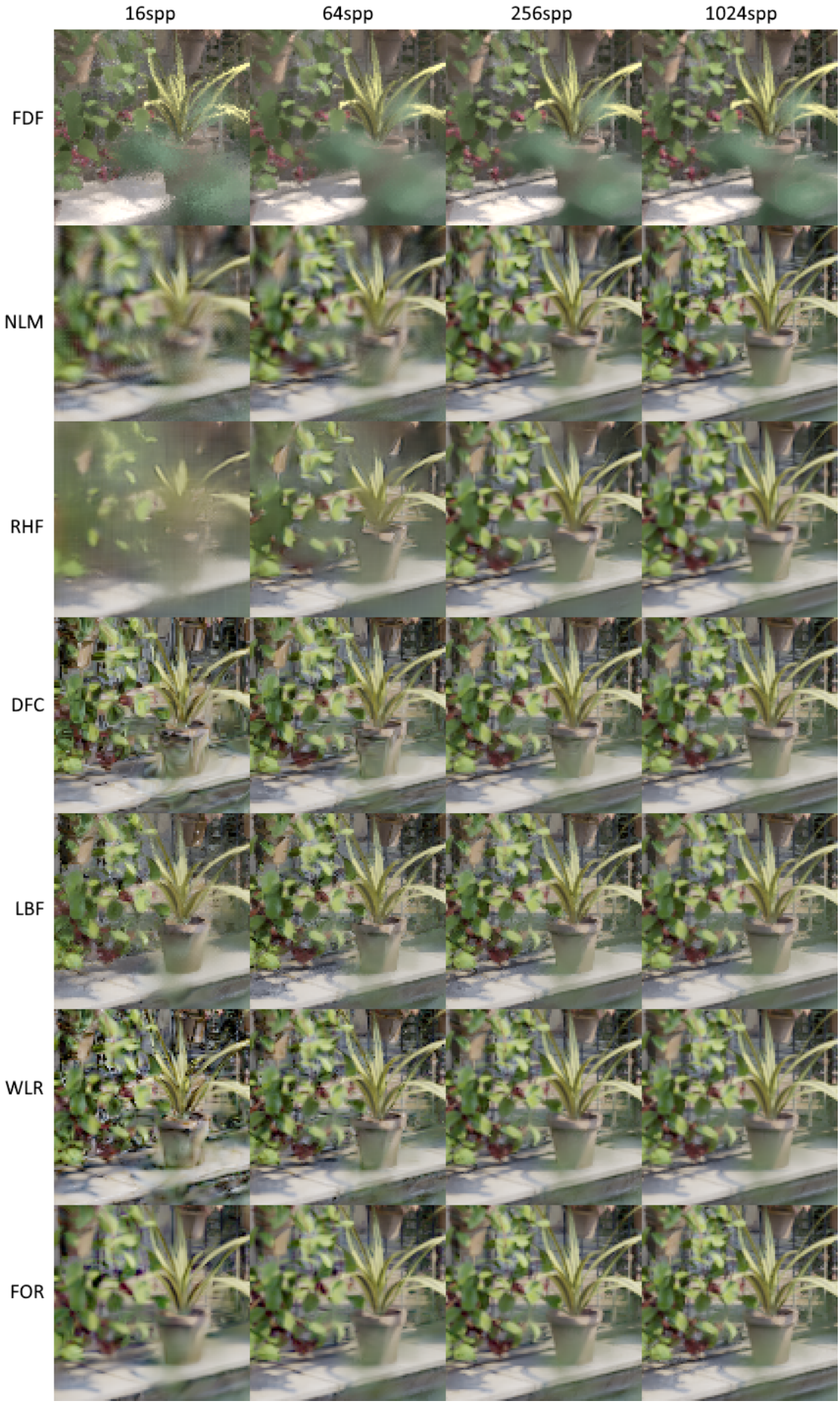

FIGURE 5.9: Comparison between all tested filters.

Chapter 6

3DIMERCE

This chapter contains all work and research specific to 3DIMERCE. First, we share a study of their product configurator framework. Then, we describe how we implemented the integration of our path tracer in this framework. Finally, we show the results of our evaluation, regarding the feasibility of using a path tracer in their product configurator framework in a production environment.

6.1 The Pancakes framework

The Pancakes framework is 3DIMERCE's internal name for the software developed in-house that their servers are running, compiled by Unity3D.

Each running Pancakes instance constantly listens for incoming messages, which can be sent by clients to request a render. Each message contains a multitude of details describing the render request: what object to render, the location and rotation of the object, the location and rotation of the camera, the resolution to render at, what background scene to use, what additional parts to add to the object, and what colors and materials to use for the object and the parts. First, all information is parsed by Pancakes. Then, an ingame Unity3D scene is arranged according to the message's information by spawning, translating, rotating, and texturing the specified objects, cameras, and background props. As soon as everything is in place, the game engine renders one frame of the scene to a texture, and closes the scene again. Finally, the texture is converted to the requested file format (usually .png or .jpg) and sent back to the client. A schematic representation of the pipeline is shown in figure 6.1.

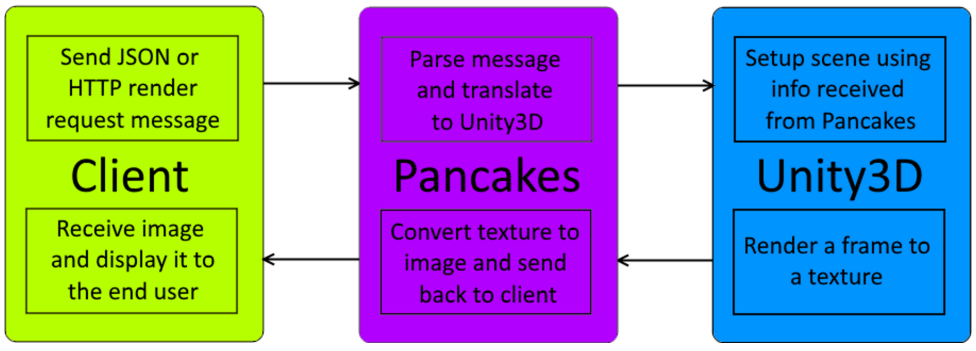

FIGURE 6.1: A schematic representation of the Pancakes framework's main use case.

In our integration, the path tracer can setup a scene configured by Pancakes, render the scene to a texture, and send it back to Pancakes: it replaces the blue Unity3D block in the schematic. Pancakes is modified to send scene info to and read a render texture back from the path tracer instead of Unity3D.

6.2 Path tracer integration

The path tracer and Pancakes are two standalone applications. When integrating the path tracer into Pancakes, it is important to keep this independency. This ensures both applications can still run without the need for one another, and that development of one application can happen without dependency on the other.

First, we explain how Pancakes can send a request to the path tracer, and how the path tracer can respond to this request. Then, we detail the process of exporting scene information in Pancakes, and importing scene information in the path tracer.

Sending a request The path tracer is a C++ Microsoft Visual Studio project, which compiles to an executable, created inside its project folder. The executable has to stay within the project folder because it depends on a considerable amount of resources, also located inside the project folder. If a user has both the path tracer and Pancakes installed on his/her machine, the most practical way for Pancakes to run the path tracer is by running the executable inside the path tracer's project folder.

To inform Pancakes of this folder's location, we added a new option to the configuration file of Pancakes, containing the location of this folder. We also added a new option to the Pancakes request messages, enabling them to ask for a path traced image. This option is accompanied by a parameter, describing the amount of time in milliseconds that the path tracer is allowed to spend on rendering the image. Pancakes' pipeline was edited to recognize this option, and forward the render request to the path tracer instead of its internal renderer.

We converted the path tracer from a Windows Forms application to a console application, allowing it to receive commandline arguments. Pancakes is modified to be able to run the path tracer's executable and send along commandline arguments that contain the scene information needed by the path tracer to know what to render.

Replying to a request The path tracer continually renders to a window, and provides no ways of exporting what it renders. Moreso, the rendering happens on the GPU, and therefore the render data resides there too. Also, these renders only contain RGB information.
To send a render back to Pancakes, we first copy the render data on the GPU back to the CPU, where we store it in an array that resides in constant memory. For Pancakes renders, alpha channel information is also needed to enable the rendering of images with transparency. Therefore, we modified the path tracer's render target to contain four values (RGBA) per pixel instead of three (RGB), and incorporated rendering of the alpha channel in the path tracing algorithm.

When the path tracer has completed a render, it writes the memory address of the array back to the commandline. Pancakes, having started the path tracer from the commandline, reads this memory address back. Then, Pancakes reads the array from memory using the received memory address, and uses this data to fill a new Unity3D render texture. This render texture is then passed back to the regular Pancakes render pipeline, at the same location where the internal Unity3D renderer would have returned a render texture containing a screenshot of the Unity3D scene. Pancakes then continues handling the render request like any other request, converting the texture to an image and sending it back to the client.

Pancakes: scene exporting The various scene configurations that Pancakes uses are stored in the project. Before sending a render request off to the path tracer, Pancakes has to gather all required scene information from the requested Unity3D scene, which is listed below. For each item in the list, a different commandline argument is used, that was designed specifically for this item.

- **Resolution:** contains a *width* and *height* parameter describing the requested render's resolution.
- **Camera:** contains the render camera's *world position*, *rotation*, and *field of view*.
- **Lights:** Unity3D contains three types of lights: *point lights*, *spot lights*, and *directional lights*. Support for these light types has been added to the path tracer, to allow accurate scene recreation. For each light in the scene, a new commandline argument is sent, containing the *light type*, *light color* (including intensity), *position*, *rotation*, and *angle*. However, this only happens if the light is configured in Unity3D to cast hard shadows. In Unity3D lights can cast hard, soft, or no shadows. Only the casting of hard shadows is physically correct for these light types. The path tracer is a physically based renderer, and can therefore only render hard shadows for these light types. To avoid bad looking hard shadows in the path traced renders where the 3DIMERCE scene designers intended soft or no shadows, Pancakes only sends lights that are configured to cast hard shadows to the path tracer.
- **Environment:** The environment for a Pancakes render consists of a skybox and an invisible ground plane, to cast shadows on. The *skybox path* is sent, along with an *intensity*. If Pancakes had to skip lights in the previous step, the skybox intensity gets an additional increase to compensate for the missing lights, ensuring the scene stays bright enough. The skybox is only used for lighting, and should not be visible in the final render. For the visible background, a *background color* is sent. If the alpha channel of this color equals zero, the background of the image will be transparent. The size and shape of the invisible ground plane is also configurable, limiting the area on the ground that can receive shadows. For this, a *shadow mask texture path* is sent.
- **Materials:** All materials in Pancakes are based on the **Hubershader**, a custom version of the internal Unity3D shader. All shader information needed by the path tracer to correctly replicate a material is directly read from the shader file by Pancakes and then sent to the path tracer. This information entails: *name*, *color*, *specularity*, *glossiness*, *translucency*, *scale*, and a number of textures: *diffuse map*, *specular map*, *glossiness map*, and up to three *normal maps*. For each texture, their associated *intensity*, *texture scale*, and *texture offset* are also sent along.
- **Meshes:** The model(s) in a render usually consist of multiple meshes, to allow optional and customizable parts for a model. For each mesh, Pancakes calculates the world space location and rotation, and then sends the needed data: *mesh file path*, *material name*, *scale*, *position*, and *rotation*.

Path tracer: scene importing The path tracer is a standalone application that starts rendering a predetermined scene on startup. We modified it to read any commandline arguments passed on startup. If commandline arguments are present, it switches into commandline mode, parses the arguments, and configures a scene accordingly. The most prevalent modifications to the path tracer that allow it to configure a scene to Pancakes' standards are listed below.

- **Resolution:** The path tracer could only use a single precompiled resolution. We modified this to allow dynamic resolution, depending on the render request.

- **Environment:** To visualize the background color, all rays that hit the skybox are separated in two distinct types: primary rays, directly generated from the camera, and extended rays, bounced from an object. The primary rays receive the requested background color for the render, and the extended rays receive the skybox color to shade the surface they bounced from.
  Rendering an invisible ground plane that can receive shadows is physically incorrect and therefore requires some substantial hacking: If a primary ray hits the ground plane, a check is performed to determine if it falls within the texture mask. If it doesn't, we render the background color. If it does, we let the ray bounce, and check what the ray hits next. If it hits an object, we shade the ground plane with the received color, which gets visualized in the form of shadow on the ground plane. If it hits the skybox, we render the background color again. Rays hitting an object first and then bouncing onto the ground plane receive a special treatment too: Rays falling outside of the texture mask get forwarded to the skybox as extended rays. Rays within the texture mask get separated into diffuse rays and specular rays, depending on the last type of bounce off the object. Specular rays also get forwarded to the skybox as extended rays. This allows visualization of skybox reflections on specular surfaces. Otherwise, the ground plane would become visible within the reflections of these specular surfaces. Diffuse rays get colored by the background color, visualizing a global illumination effect on the underside of the object.

- **Materials:** In Pancakes, a material has one layer, and both specularity and glossiness parameters. In the path tracer, a material has two layers, both having only a specularity parameter. Through testing, we found that always setting one layer fully diffuse, matching the other layer's specularity with the Pancakes glossiness parameter, and setting the balance between the two layers proportional to the Pancakes specularity parameter, to give the most satisfactory results.
  Support for the various material textures, with alpha, scales, and offsets, has also been added to the path tracer, as well as support for partially translucent materials, to cover Pancakes' translucency parameter.

- **Meshes:** All Pancakes meshes are of the *Filmbox (.fbx)* format, developed by Autodesk. A novel importer using Autodesk's *FBX SDK* has been implemented, to import the Pancakes meshes into the path tracer.

6.3 Objective and results

3DIMERCE's wants to know whether using a path tracer in their product configurator solution is feasible. The limiting factor is the render time required. They impose a strict requirement on the render time: Rendering the Bugaboo Bee[5] scene, with a resolution of 800x800 pixels, using a Nvidia GTX 1080, should not take more than 800 milliseconds. Referencing the results in table 5.2, we are 48x off performance target without, and 16x with employing our filter. Thus we can conclude that using a path tracer is not feasible for 3DIMERCE, at least until a combination of advances in hardware performance and significant optimizations in path tracing and filtering result in a 16x speedup over what can be achieved today.

In 800ms, we do still achieve reasonable quality when employing our filter, but not high enough for the render to be accepted as high-quality, as visible bias is still left-over. Figure 6.2 shows an example of the reached and desired image quality: here the quality of shadows after filtering on lower sample counts is insufficient.

FIGURE 6.2: From left to right: 800ms noisy input, 800ms denoised (spotty shadows), 9600ms denoised (acceptable quality), reference.

Our filter performs well at the 800ms mark, improving measured MSE by 3-16x (table 6.1). FDF, taking about 200ms to execute, was designed to perform optimally around the 800ms mark, since rendering images within this timeframe is the ultimate goal of high-quality interactive rendering. As it turns out, to achieve images of sufficient quality, the filter is applied after 5-15s, where it is relatively less effective: a more complicated filter requiring 1-3s to execute could probably have achieved a significantly larger speedup than FDF around this timeframe.

Scene	**Unfiltered**	**Filtered**	**Factor**
Bee[5] transparent	0.002344	0.000510	4.6x
Bee[5] opaque	0.001503	0.000505	3.0x
Pure	0.001383	0.000087	15.9x

TABLE 6.1: FDF's MSE improvement on 800ms renders.

Besides increasing compute power, we believe the results can be improved by optimizing the meshes of the models used in the renders. The Bugaboo Bee[5] has a very high polycount of over 2 million polygons. The level of detail of such a high polycount is too precise to be visible in a 800x800 render. Reducing the polycount could significantly improve performance.

Chapter 7

Discussion and future work

Main research question Our main research question was related to finding out whether high-quality interactive path tracing for use in a production environment is possible today, or in case it is not, how far off we currently are from this goal. We formulated the question as:

How fast can high-quality images be rendered, by means of path tracing?

Where the images are production environment ready, portraying customizable and complex models of products at useable resolutions, and are of high quality, containing no noise, bias, errors, or glitches, and visualizing all graphical effects associated with path tracing where applicable. The answer to this question, on the specific scenes and hardware we used for evaluation, is **roughly 5-15 seconds**. Interactive render times are ideally under a second, but up to a couple of seconds is acceptable. Thus, we currently are **about 2-16 times** off from this goal, depending on scene complexity and the precise amount of allowed render time.

Discussion Given that we are still about 2-16x off target, we can estimate the amount of time it will take for advances in hardware to catch up to this deficiency. For this, we reference *Moore's Law*, which states that roughly every two years the number of transistors on integrated circuit chips doubles, and therefore in a best-case scenario performance too. This means that we might be able to just start being able to render interactively in as little as two years, and are likely to interactively render complex scenes in under a second in less than a decade.

The premise of this research project was that enabling high-quality interactive path tracing was likely to be possible today, with the help of a postprocessing filter specifically tailored for interactive rendering. In hindsight, we believe this premise to have been overly optimistic. Our novel filter FDF achieves comparable denoising performance to the current state-of-the-art, while requiring only a couple of hundred milliseconds to execute, allowing it to be used for interactive rendering, while the current state-of-the-art could require multiple minutes of compute time. However, our filter still needed roughly 10x more path tracing samples to be able to denoise renders without visible bias than could be traced in under a second. Thus, for high-quality interactive path tracing to be feasible today, we would have had to propose a postprocessing filter that would improve over current state-of-the-art tenfold in performance, while still needing only a couple of hundred milliseconds to execute.

Scientific contribution We gave a scientific contribution related to postprocessing denoising filters by researching current filters, implementing novel techniques, and comparing between state-of-the-art and novel filters, summarized below:

We thoroughly examined *Ray Histogram Fusion* by implementing our own version and found weaknesses in the algorithm, causing overblurring in low-contrast areas, killing expectations of RHF being suitable for interactive rendering.

To combat RHF's problems we went with a feature-based filtering approach, of which we had to implement our own version *Feature Distance Filter*, since the currently available feature-based filters required computation times at least in the order of seconds. The main shortcoming of FDF was the overblurring of lighting effects like shadows, translucency, and color bleeding. Pixels are discriminated based on their distances to each other in feature buffers, and we could not find usage of a feature buffer directly targeting lighting effects in the literature.

To address this shortcoming, we proposed a novel feature buffer, the *shading feature buffer*. It is created by accumulating the amount of light that reaches each pixel while sampling. This resulted in a noisy buffer, which we opted to denoise using NL-means. NL-means buffer denoising has been implemented before in literature, and we implemented our own variant, keeping computation time in mind, to create a faster variant with relatively little denoising quality loss.

We compared our novel filter FDF to the current set of state-of-the-art filters on two representative scenes and found it to achieve adequate denoising quality, performing best overall in one scene and performing averagely in the other. We compared runtimes and found our filter to be second fastest, 100ms behind the fastest filter RHF, achieving 10-100x faster runtimes than other filters in the comparison. In our opinion we have proposed a new filter that achieves the best tradeoff between denoising quality and runtime so far, advancing progress in the field of postprocessing denoising filters for path traced images.

FDF's limitations Currently, FDF's main limitation is the slight overblurring and underblurring of certain areas. Ocassionally, on edges and complex areas of a scene, FDF underblurs and leaves some residual noise, because all neighbouring pixels are discriminated by one or more feature buffers, preventing the current pixel from merging with any other pixels, leaving it noisy. Relaxing feature buffer distance thresholds however poses no solution, as this increases overblurring in FDF: In areas of extremely fine and low-contrast detail, FDF incorrectly overblurs, since the feature buffer distance thresholds are too great to discriminate any neighbouring pixels, pushing the current pixel to merge with all similar neighbours.

FDF future work Future work would be mainly focused around solving FDF's main limitation. It would be interesting to research any possible adaptive per-pixel feature buffer distance threshold techniques. This technique should relax distance thresholds on noisy pixels (e.g. on edges) to ensure they always get blurred enough to look smooth, while it should tighten distance thresholds in cases where there are many possible pixels to merge with, only merging with extremely similar pixels to preserve fine detail. Another approach would be simply optimizing the complete algorithm, to reduce computation time. This would allow for an increase in both the NL-means and bilateral filter's search window size, increasing the amount of possible pixels to merge with, which in turn could allow for smaller feature buffer distance thresholds, without inducing more underblurring.

Chapter 8

Acknowledgements

I wish to thank various people for their contribution to this project; Jacco Bikker, for introducing me to 3DIMERCE, and providing an expert opinion and key insights in various problems encountered during the making of this work. Huub van Summeren, for always helping me out at 3DIMERCE, and providing a pleasant and stimulating working environment. Kevin van Mastrigt and Olaf Schalk, for helping me work out various programming issues and bugs. And Tessa Klunder, for her love and support.

Bibliography

[Bau+15] Pablo Bauszat et al. “General and robust error estimation and reconstruction for monte carlo rendering”. In: *Computer Graphics Forum*. Vol. 34. 2. Wiley Online Library. 2015, pp. 597–608.

[BCM05] Antoni Buades, Bartomeu Coll, and J-M Morel. “A non-local algorithm for image denoising”. In: *Computer Vision and Pattern Recognition. IEEE Computer Society Conference on*. Vol. 2. IEEE. 2005, pp. 60–65.

[Bit+16] Benedikt Bitterli et al. “Nonlinearly Weighted First-order Regression for Denoising Monte Carlo Renderings”. In: *Computer Graphics Forum*. Vol. 35. 4. Wiley Online Library. 2016, pp. 107–117.

[Dab+06] Kostadin Dabov et al. “Image denoising with block-matching and 3D filtering”. In: *Electronic Imaging 2006*. International Society for Optics and Photonics. 2006, pp. 606414–606414.

[Del+14] Mauricio Delbracio et al. “Boosting monte carlo rendering by ray histogram fusion.” In: *ACM Trans. Graph.* 33.1 (2014), pp. 8–1.

[Kaj86] James T Kajiya. “The rendering equation”. In: *ACM Siggraph Computer Graphics*. Vol. 20. 4. ACM. 1986, pp. 143–150.

[KBS15] Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. “A machine learning approach for filtering Monte Carlo noise.” In: *ACM Trans. Graph.* 34.4 (2015), p. 122.

[KS13] Nima Khademi Kalantari and Pradeep Sen. “Removing the noise in Monte Carlo rendering with general image denoising algorithms”. In: *Computer Graphics Forum*. Vol. 32. 2pt1. 2013, pp. 93–102.

[MCY14] Bochang Moon, Nathan Carr, and Sung-Eui Yoon. “Adaptive rendering based on weighted local regression”. In: *ACM Transactions on Graphics (TOG)* 33.5 (2014), p. 170.

[Par+13] Hyosub Park et al. “P-RPF: Pixel-based random parameter filtering for Monte Carlo rendering”. In: *Computer-Aided Design and Computer Graphics (CAD/Graphics), 2013 International Conference on*. IEEE. 2013, pp. 123–130.

[Por+03] Javier Portilla et al. “Image denoising using scale mixtures of Gaussians in the wavelet domain”. In: *IEEE Transactions on Image processing* 12.11 (2003), pp. 1338–1351.

[RMZ13] Fabrice Rousselle, Marco Manzi, and Matthias Zwicker. “Robust denoising using feature and color information”. In: *Computer Graphics Forum*. Vol. 32. 7. Wiley Online Library. 2013, pp. 121–130.

[SD12] Pradeep Sen and Soheil Darabi. “On filtering the noise from the random parameters in Monte Carlo rendering.” In: *ACM Trans. Graph.* 31.3 (2012).

[Sze+15] Sebastian Szeracki et al. “Boosting histogram-based denoising methods with gpu optimizations”. In: *Workshop Virtuelle Realität und Augmented Reality der GI-Fachgruppe VR/AR*. 2015.