Håkon Flatval

Denoising Algorithms for Ray Tracing in Real-Time Applications

June 2020

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Master's thesis

2020

Håkon Flatval

Master's thesis

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Denoising Algorithms for Ray Tracing in Real-Time Applications

Håkon Flatval

Computer Science
Submission date: June 2020
Supervisor: Theoharis Theoharis

Norwegian University of Science and Technology
Department of Computer Science

Håkon Flatval

Denoising Algorithms for Ray Tracing in Real-Time Applications

Master's Thesis in Computer Science

Supervisor: Theoharis Theoharis

Lier, June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Innovation and Creativity

Abstract

Ray tracing is a rendering technique that has until recently been considered too expensive for real-time applications. Advances in graphics hardware technology now allows the technique to be used in real time, but only with a limited number of *light rays* per pixel, resulting in noisy images. To produce convincing imagery with ray tracing in real-time, it is therefore necessary to use *denoising* algorithms.

This thesis explores the history and development of both ray tracing and denoising techniques that has lead to their applicability in the realm of real-time applications. Subsequently, it presents two state-of-the-art denoising algorithms in detail: Spatiotemporal Variance-Guided Filtering (SVGF) and Blockwise Multi-Order Feature Regression (BMFR).

Then two experiments are presented: The first compares the two algorithms in terms of performance and visual quality. The second experiment tests a proposed extension to BMFR to improve the result.

From the results, it is concluded that although both algorithms have their individual strengths and weaknesses, BMFR displays superior performance and better visual quality in many cases. The proposed extension to BMFR turned out to have little effect.

It is noted that modern denoising techniques have come a long way and can create convincing results. However, a number of visual quality issues must be adressed before ray tracing can take over as the preferred render technique in real-time applications.

Sammendrag

Ray tracing er en bildegjengivelsesteknikk som inntil nylig var avskrevet som for beregningstung for sanntidsapplikasjoner. Teknologiske fremskritt innen grafikkmaskinvare har gjort teknikken mulig å bruke i sanntid, men kun med et begrenset antall *lysstråler* per piksel, noe som resulterer i støyete bilder. For å produsere mer overbevisende avbildninger med ray tracing i sanntid, er det derfor nødvendig med *støyfjerningsalgoritmer*.

Denne oppgaven utforsker historien og utviklingen til både ray tracing og støyfjerningsteknikker som har gjort det mulig å bruke dem i sanntidsapplikasjoner. Deretter presenterer oppgaven to moderne støyfjerningsalgoritmer i detalj: Spatiotemporal Variance-Guided Filtering (SVGF) og Blockwise Multi-Order Feature Regression (BMFR).

Så presenteres to eksperimenter: Det første sammenligner de to algoritmene på både ytelse og visuell kvalitet. Det andre eksperimentet undersøker en foreslått utvidelse av BMFR for å øke kvaliteten på resultatbildene.

Fra disse resultatene konkluderes det med at selv om begge algoritmene har individuelle fordeler og ulemper, viser BMFR overlegen ytelse og bedre visuell kvalitet i mange tilfeller. Den foreslåtte utvidelsen av BMFR viste seg å gi liten effekt.

Til slutt nevnes det at moderne støyfjerningsteknikker har kommet langt, og kan allerede produsere overbevisende resultater. Likevel er det flere problemer med den visuelle kvaliteten som må imøtekommes før ray tracing kan ta over som den foretrukne bildegjengivelsesteknikken for sanntidsapplikasjoner.

Preface

I would like to direct a thanks to my parents for their never-ending support, no matter what I put my mind to.

I would also like to thank my supervisor Theoharis Theoharis, who has always been available to give me helpful advice and support in rough times during the work on this thesis.

Additionally, I would like to give huge thanks to Jan-Harald Fredriksen and Marius Bjorge at ARM's office in Trondheim, who originally brought my attention to this exciting topic, and who have provided me with useful guidance both before and during this endeavor.

The work associated with this document was performed within the period of time from January 2020 to June 2020. It has been a challenging semester for me, but undoubtedly also for the many other people around the globe who are affected by the ongoing pandemic. As I write the final words of this thesis, my thoughts go out to those who have lost their loved ones, to those who have met unforetold struggles the last few months, and to those who look into the future with deep concern.

Contents

Chapter 1

Introduction

1.1 Motivation: Real-Time Ray Tracing

Real-time computer graphics applications have relied on rasterization as the primary rendering technique for a long time. Rasterization is both fast and flexible, and many methods have been developed to approximate real-life effects like shadows, reflections and global illumination. However, the cost of approximating these effects accurately is high, and usually limited to static scenes.

Ray tracing is a family of more accurate rendering techniques which is often deployed in rendering industrial designs and movie productions. Ray tracing algorithms produce imagery by computing the paths of light rays to accurately estimate the illumination of the scene. The most common type of ray tracing algorithm is *path tracing*, where the key idea is to simulate many light rays (samples) stochastically to capture as many light paths in the scene as possible.

Ray tracing will often give more realistic-looking results than rasterization since effects like shadows, reflection and global illumination are automatically handled when simulating the light rays. However, ray tracing, and especially path tracing, has a high computational cost. This is because the number of samples required to capture the distribution of light accurately is very high, often several thousands of samples per pixel. Therefore, path tracing has long been considered infeasible, especially for real-time settings.

Recent developments in graphics hardware give hope that real-time path tracing is rapidly approaching. The catch is that even with such new hardware, the number of light rays that are feasible to simulate at real-time framerates is very low, usually only 1 *sample per pixel* (spp) every frame. Due to the stochastic nature of path tracing, such a low sample count will

produce unacceptable levels of noise in the image. Thus, to make path tracing feasible for real-time applications, techniques for removing the noise in the final image is needed.

1.2 Goal of Thesis

This thesis will explore the realm of denoising techniques targetting real-time path tracing. The objective of this document is stated through four research goals:

- Gain an understanding of the history and development of ray tracing and denoising techniques up to today
- Gain a thorough understanding of two specific state-of-the-art denoising techniques targetting real-time path tracing
- Compare and evaluate the two state-of-the-art denoising algorithms by conducting experiments investigating both performance and visual quality
- Provide a proposal for extending one of the denoising algorithms and evaluate the proposal through experiments

The structure of the thesis follows this list of goals. Chapter 2 will go through both the history of ray tracing techniques and denoising, highlighting some of the many scientific breakthroughs and contributions that have cumulated in the state-of-the-art methods. Chapter 3 will go through two hand-picked state-of-the-art denoising algorithms for real-time path tracing in detail. Chapters 4 and 5 will give an explanation of the experiments that have been conducted and their results, respectively. Lastly, chapter 6 will summarize the take-aways from the thesis, together with its limitations and ideas for future work.

Chapter 2

Related Work

This section will take a step back and try summarize the history behind ray tracing, including different ideas and techniques that have been invented and developed. It will point out some important steps up to today that has made ray tracing feasible for real-time applications.

2.1 Early Ray Tracing

The *ray casting* algorithm by Arthur Appel [5] was presented in 1968 and is recognized as the ancestor of modern ray tracing algorithms. Ray casting involves tracing one ray for each pixel and follow it until it collides with geometry in the scene. Then the algorithm traces a ray from the collision point to each light source to determine the illumination at that pixel. The illumination value is computed based on the whether the light sources were found to be reachable by the second batch of rays.

Appel's work targetted the 60's display technology, which had limited drawing capabilities compared to today's devices. Therefore, the goal fidelity by his method was not absolute realism, but a visualization that could conveye a reasonable sense of 3D geometry to the viewer. The approach mainly targetted industrial use, e.g. for visualizing machine parts.

Turner Whitted further elaborated on the concept of casting rays in 1979, and described what we today would recognize as traditional ray tracing [53]. In Whitted's algorithm, a ray from the camera that hits geometry in the scene, can recursively create up to three new rays of different types: A shadow ray, a refraction ray and a reflection ray. *Shadow rays* resemble the illumination-checking ray in Appel's algorithm – it is a ray that is sent towards a light source. If the light source is directly visible from the collision point, the point is illuminated by the light source. The exact illu-

mination value depends on the angle of the shadow ray to the surface and the intensity of the light source.

Refraction and *reflection* rays act just as the rays originating at the camera. Reflection rays represent light that gets reflected off the surface, while refraction rays represent light that travels through the material, piercing the surface. Both of these ray types are handled in the exact same way as the original ray, traced recursively away from the collision point and colliding with new surfaces. The illumination values computed from each of these paths can be combined with material parameters to find their impact on the final color that is propagated back to the viewer. Looking at it this way, one can consider computing illumination at a single pixel as traversing a tree of recursive rays and collision points, computing the final root value by a depth-first-like approach. The recursion can terminate when a certain depth is reached, or when the contribution to the pixel color is negligible.

Whitted only considers ideal reflection and refraction, leaving the ray direction unambiguously determined for each of the ray types. This gives realistic-looking effects for rendering perfectly smooth surfaces, but cannot faithfully render most realistic surfaces. This stems from the fact that real-life light is scattered probabilistically on surfaces, leaving many new rays that would need to be traced.

2.2 The Rendering Equation

James Kajiya introduced the rendering equation [31] in 1986. The rendering equation is a compact description of the amount of light that is transmitted *directcly* from one surface point x' to another point surface point x in the scene. It is of the form

$$I(x, x') = g(x, x') \left(\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right).$$

Here, $I(x, x')$ is the illuminaton term describing the amount of light passing in a straight line from point x' to x, $g(x, x')$ is a "geometry term" which e.g. will be 0 when there is an object directly between x' and x blocking the light path. $\epsilon(x, x')$ represents the light emitted from x' towards x and S is the union of all surface points in the scene. $\rho(x, x', x'')$ is dubbed the *unoccluded three point transport reflectance* from x'' to x through x' in the original paper, and represents how much of the light that is transported directly from x'' to x', is transported further from x' to x.

ρ is solely the property of the surface material at x', and is not directly dependent on the positions of the three points. Rather, it depends the

directions to x and x'' from x' relative to the surface orientation at x'. This function is what we call the bidirectional scattering distribution function (BSDF). It had been described earlier by Bartell et al. [7] in 1980.

BSDF is an important concept, because it fully captures how light interacts with a surface at a point. For a light ray incoming from a certain direction, the BSDF describes how much of the light is reflected back out in every direction from the collision point.

While the render equation is very general and compact, it imposes severe difficulties when used in computer graphics due to its integral nature. One could attempt to use Whitted's algorithm and, at each surface collision, sample several new rays from the BSDF to trace recursively. The most imminent problem is that the amount of rays grow exponentially with the recursion depth, so that the load quickly becomes intolerable for any setting. Furthermore, the BSDF can vary frequently across the same surface, making aliasing a problem – tracing a single ray for every pixel may lead to a final color that is not representative for all the surface area that the pixel covers.

2.3 Path Tracing

A solution to this computational problem, which is also discussed by Kajiya, is to switch to an entirely *Monte Carlo*-based ray tracing approach which has been known as *path tracing*. In short, path tracing creates many rays from different uniformly distributed positions within each pixel. When a ray collides with a surface, a single new ray is generated from the collision point, its direction sampled from the BSDF. By the Law of Large Numbers, such a simulation will generate an approximate numerical solution to the illumination problem, given that the number of rays is large enough. In contrast to Whitted's algorithm, the number of rays does not increase with depth. Intuitively, this is a reasonable sacrifice, as surfaces hit at higher recursion depths will contribute less to the final illumination value.

While path tracing virtually *guarantees* a realistic-looking result for a large enough ray count, the required computational cost can be infeasible for practical purposes. A typical render of a scene can require several thousand ray *samples per pixel* (spp) to arrive at a satisfactory result. This number grows with the complexity of the scene and its surfaces. Also, as the algorithm is stated here, every ray that does not eventually hit a light emitting surface contributes nothing to the final image, which is detrimental to complex scenes with small light sources. Therefore, it is common to combine ray tracing with the idea of Whitted's shadow rays. In practice, one can e.g. trace a shadow ray to a randomly chosen light source for every

ray-surface-collision point, and add the illumination if the light source is not occluded.

In the rest of this document, path tracing algorithms will be considered as a specific class of ray tracing. Consequently, statements made about ray tracing will also apply to path tracing – which will be the main subject of discussion throughout the thesis.

2.4 Path Tracing Effects

One of the great advantages of ray tracing, is that it generalizes well to naturally occuring phenomena. Cook et al. [19] described in 1984 how a path tracing algorithm could easily modified to cope for effects such as motion blur, depth of field, penumbra, translucency and glossiness. For instance, depth of field can be acheived by altering the distribution of ray directions and origins in the camera. Motion blur can be approximated by giving each ray a timestamp uniformly chosen from the exposure period. Every object in the scene is given a motion vector, and the ray-surface executions are then computed using the estimated object positions in the ray's timestamp. No other changes are needed for these two effects, the results are computed by relying on the same same Monte-Carlo approach as "vanilla" path tracing.

2.5 Challenges in Path Tracing and their Solutions

Ray tracing approaches give many advantages over more traditional rendering algorithms, but there are undoubtedly new challenges that comes with these opportunities. A number of algorithms have been developed to tackle different drawbacks one must expect when using path tracing. A few of them will be listed here, to show some of the variety of different problems that have arisen and the solutions that have emerged. Many of the topics here are also discussed in the survey by Christensen and Jarosz [16].

2.5.1 Bounding volume Hierarchies

The first and most glaring issue with ray tracing in general, is the immense computational load it is to trace many rays per pixel. It is therefore imperative that each ray is as cheap as possible. This is not trivial since, for every ray, one must find its first intersection with geometry in the scene. A straight-forward algorithm would be to check every ray with every primitive

in the scene and choose the closest hit, yielding a runtime linear in both the number of rays and the number of primitives in the scene, hardly feasible for any but the simplest ray tracing tasks.

A better approach is to utilize a datastructure to subdivide the scene space into smaller chunks. One such datastructure is the *Bounding Volume Hieranchy* (BVH). A BVH is a tree structure where each node is associated with a *bounding volume*, often an axis-aligned box, and where each node has the property that its bounding volume contains the bounding volumes of its children. The leaf nodes of the tree holds references to the geometry that is contained within their bounding volumes. Naturally, it is ensured that every piece of geometry in the scene is contained within the bounding volume of at least one leaf node. It is not required that the bounding volumes are disjoint.

A reasonable approach to constructing a BVH is to recursively split the scene in two along an axis-aligned plane that puts the primitives of the scene geometry into two evenly sized chunks. The recursion ends at a certain bounding volume size or geometric primitive count, and the remaining primitives are put into a single leaf node.

The speedup from a BVH comes from the fact that the ray tracer can discard large amounts of geometry at a time by checking for intersection with the bounding volume. Starting with the root node, if the ray intersects the bounding volume of a node, it recursively checks its children. This goes on until it reaches a leaf node, in which case the ray is checked against all the primitives associated with the node. The efficiency of this approach highly depends on the scene and the camera orientation, but a reasonable approximation is that using BVH reduces the algorithmic complexity to as little as $O(\log N)$ per ray, N being the number of primitives in the scene.

Lauterbach et al. [35] and Wald et al. [51] show that Bounding Volume Hierarchies can be used efficiently, even in dynamic scenes.

2.5.2 Sampling Patterns

As mentioned, ray generation in path tracing can happen through uniform sampling of points within the pixel in question. However, when the affordable number of samples per pixel is limited, it is more important that the generated rays covers the pixel sufficiently, rather than of being properly random. In general, one would want a *high-discrepancy* distribution, which is a distribution of samples that can be perceived as random, but where the samples are guaranteed to be somewhat evenly distributed.

One simple method to generate a high-discrepancy distribution is jitter [17]. N^2 jittered samples can be constructed by dividing the pixel into

$N \times N$ uniform tiles, and subsequently sample once uniformly at random within each tile. A variation of this is multijitter [13], where the pixel simultaneously is subdivided with N^2 rows and columns, each of which is only allowed to contain one sample.

In some cases, the number of required samples cannot be known prior to computation. An alternative is to compute a sequence of high-discrepancy samples of arbitrary length. Examples of such sequence generating algorithms are Halton sequences [29] and Sobol sequences [47].

2.5.3 Bidirectional Ray Tracing

Another problem is that path tracing may require many samples or large recursion depth to reach light sources that are small or not in direct line of sight from the majority of the scene surface. In the latter case, even shadow rays may not reach the light source.

A solution to this problem is *Bidirectional Ray Tracing*, proposed both by Lafortune and Willems [34] and Veach and Guibas [49] independently, in which rays are traced *both* from the light sources and the camera. By sampling rays from the light source, one can treat their collision points with the geometry as new light sources, which can be connected with the camera rays using shadow rays as before. In cases where indirect lighting is important, that is, where the original light source is largely hidden away, this may drastically increase the number of rays that can contribute to the light, thereby reducing noise while resulting in similar rendering times.

2.5.4 Memory Management

The final issue that is discussed here, is that of memory management. Ray tracing faces severe performance problems when faced with scenes that are too large to fit in working memory. In some traditional rendering methods, e.g. rasterization, one can trivially render large scenes by loading in one and one object and render them independently, only maintaining e.g. depth and color buffers representing what's already rendered.

Ray tracing algorithms, on the other hand, have unpredictable access patterns due to the stochastic nature of ray sampling, meaning loading objects in and out of memory may take up a significant portion of the run time if performed naively.

An early approach to this was Pharr and Hanrahan's *Geometry Caching* [41] in 1996. Their algorithm assumes that the world geometry has relatively low detail which is refined using displacement maps when rendering.

Pharr and Hanrahan's approach was to subdivide the scene into voxels, each voxel holding a reference to all the geometry it contained. When a ray travels through a voxel, all geometry within the voxel is subdivided and refined using the displacement maps, allowing for intersection testing with the ray. The voxels of refined geometry is maintained in a geometry cache, so that it does not have to be recomputed if it is to be intersected with new rays in the near future. This makes it possible to render complex scenes with a smaller memory footprint.

Further down the line, Pharr et al. introduced *Ray Reordering* [42] to produce more temporal locality in using their geometry cache. Each voxel is given a ray queue, which rays are pushed to as they arrive at the voxel. Then all rays in the queue is tested for intersection in the voxel simultaneously, ensuring good utilization of the voxel.

Disney also has a ray reordering algorithm in their Hyperion engine [23]. It works by sorting rays into batches based on origin and directions, which are then traced one by one. Their algorithm also sorts the collision points with scene geometry before looking up texture maps, in order to make the lookup of many texture samples as efficient as possible.

2.6 Ray Tracing in Movies

Path tracing was long considered too expensive to be used in a feature-length film. Many early computer generated effects, as well as the first completely computer-generated animated feature film *Toy Story* (1995, Pixar) used Reyes rendering [18]. The Reyes algorithm uses ray tracing sparsely, and only when deemed necessary.

An early movie production that was fully path traced, was the short film *Bunny* by Blue Sky studios in 1998. The first feature-length film came nearly a decade later with *Monster House* in 2006 [16]. Christensen et al. have expanded upon how they extended Pixar's rendering tool Renderman with ray tracing capabilities for the movie *Cars* from the same year [14]. The authors detail how ray tracing made effects like arbitrary reflections, motion blur and ambient occlusion more believable and straight-forward to create. However, ray tracing was only used supplementary to the Reyes algorithm, much due to the number of features the Reyes algorithm could still handle sufficiently and more efficiently.

Nowadays, Pixar, Disney, and a large part of the animated movie industry largely relies on path tracing as a primary method of rendering in feature-films [15], [11], [24]. A large part of this move in technique can be attributed to a large increase in hardware capabilities as well as algorithmic

improvements.

2.7 Towards Real-Time Path Tracing

Path tracing has long been considered a very expensive way of creating images, even for industry-scale film production. Still, work in the last few years have paved the way for path tracing in real-time settings.

An important step towards real-time ray tracing has been the utilization of GPU hardware to cope with the heaviest computations. A work back in 2006 by Friedrich et al. [26] explored what opportunities ray tracing could bring to game content. They show that ray tracing applications with interactive framerates exist at the time, but emphasize that future development in algorithms and parallel hardware will be necessary for the adaption of ray tracing in games.

GPU powered ray tracers were not able to outperform CPU implementations before 2007 by Popov et al. [43]. The difficulties with ray tracing on GPUs stems from issues with utilizing the parallelism with space subdivision data structures and thread divergence.

An important step towards the adaption of real-time ray tracing among developers, has been the introduction of uniform software APIs to take advantage of GPU accelerated ray tracing. An early example is NVIDIA's OptiX [40] released in 2010, that is tightly integrated to their CUDA API. While general-purpose real-time ray tracing still was years away, the specification of the API signifies the intention to entirely abstract away e.g. ray-geometry intersection algorithms and make ray tracing applications portable across GPU architectures.

More recently, similar APIs have also been introduced to already existing graphics APIs, specifically to DirectX (DirectX Raytracing [1]) and to Vulkan, where the latter was given the capabilities both in the form of an NVIDIA-specific extension [52] and, very recently, an official cross-platform Vulkan extension for ray tracing [32].

In 2018, NVIDIA released their first GPUs using their Turing architecture. These graphics cards have hardware cores dedicated to the task of ray tracing [39]. The release made real-time ray tracing hardware available for consumers, leading to the appearance of a large amount of video games using ray tracing [2], but usually only for certain special effects.

So far, NVIDIA is the only company to have released devices that target real-time ray tracing, but their competitor AMD has announced that their upcoming GPU architecture RDNA will hold similar capabilities [4]. It is tempting to predict that the increased competition will further accelerate

the development of hardware and algorithms to make fully ray traced real-time applications a reality in the near future.

2.8 Denoising

The most prominent problem with ray tracing, image quality-wise, is noise. Noise is an inevitable consequence of the probabilistic nature of path tracing. One important cause of visible noise is the lack of a sufficient number of samples. In real-time ray tracing, the number of samples per pixels is bound to be low, making the result unacceptably noisy. Therefore, denoising is an integral part of any real-time ray tracing application, but denoising algorithms are also put to use in offline settings, like movie production.

The survey given by Zwicker et al. [57] summarizes much of the denoising work that has been done since the start. Zwicker et al. divides the denoising techniques into two main categories: *A priori* and *A posteriori* techniques.

A priori techniques try to compute features of the motive, like spatial frequency or gradients, and use this information to guide sampling. In particular, one can compute spatial frequency in different regions of the image and apply Nyquist's sampling theorem to avoid aliasing. Durand et al. demonstrated this method by analyzing the frequency of the resulting image and were able to recontsruct the image from very sparse sampling with *bilateral filters* [22]. Bilateral filters will be discussed in section 3.2.1.

The rest of the methods discussed here, belong among the a posteriori methods. These methods sample first, and try to make a smooth image from the noisy samples, or use them to guide future sampling.

An early example of such an a posteriori method is Don Mitchell's method for avoiding aliasing with low sampling density [38]. Mitchell performs non-uniform sampling of the image, guided by a few initial low-density samplings to detect regions of high spatial frequency, calling for higher amounts of samples.

The denoising techniques are further divided into offline and real-time denoising methods, optimized for high counts of samples per pixel (spp) and low spp respectively.

2.8.1 Offline Denoising

Offline denoising is typical for movie production. The amount of noise in the images is usually close to negligible, but an important task of an offline denoiser is to ensure consistency across frames.

Denoising methods, both offline and real-time, can usually assume access to excess information from the rendering stage. This information may include world positions, surface normals, albedo and material properties. They are usually provided to the denoiser as images of the same dimensions as the noisy image, where each pixel contains e.g. a position or a normal. Saito and Takahashi introduced this concept as *G-Buffers* [45], while this document will refer to such images as *feature buffers*.

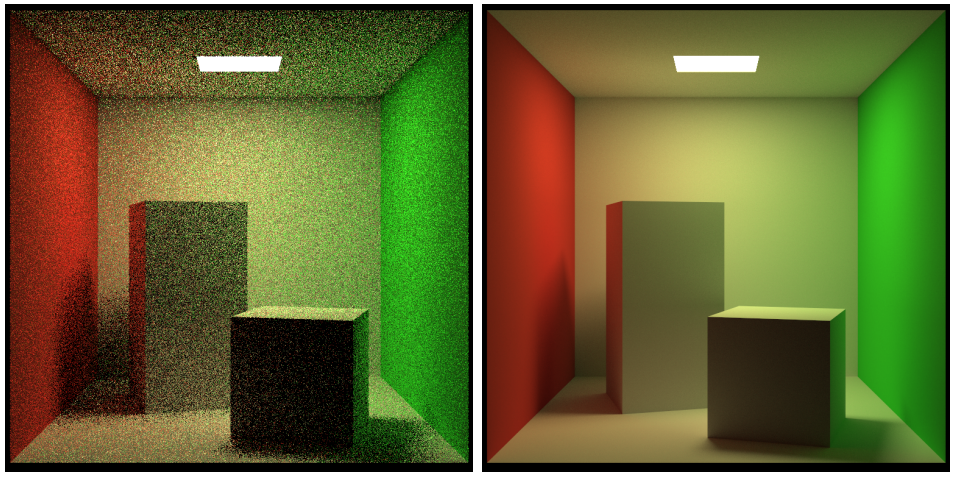
Non-linearly Weighted First-order Regression (NFOR) is a powerful denoiser by Bitterli et al. made for denoising ray tracing in an offline setting [9]. Their work performs an extensive analysis on previous methods, including bilateral filters and local zero- and first-order regression with feature buffers. Based on the analysis, they propose a method that uses a combination of several passes of local regression using NL-means [10] with weights computed from feature buffers. They also makes an estimate of an applicable kernel frequency bandwidth to optimally remove noise while preserving detail.

An example of industry use of offline denoising is given by Vicini et al. at Disney [50]. Their work specifically handles *deep images*, images where each pixel also holds information on occluded objects, thus preserving 3D information. As deep images is quite popular with animated movie production, it is expected that a denoiser can take advantage of them. Vicini et al. specifically show an existing denoising technique adapted to remove noise at different depths independently. They build their work on NL-Mean filters which also take features such as depth and surface normals into consideration.

2.8.2 Real-Time Denoising

Real-time denoisers have two main goals that differ from those of offline denoisers. First of all, they must work on very sparsely sampled images, usually 1 spp. Several works (e.g. Koskela et al. [33]) argue that the limit of 1 sample per pixel is likely to persist for a long time, considering that real-time applications like video games are seeing an ever-increasing demand in geometric level of detail and display resolution, making it unlikely that any immediate increase in computational power will benefit the sample count. The loss of detail at this level of sampling may be so severe, that these techniques are often said to *reconstruct* the image rather than remove noise from it.

The second goal for a real-time denoiser, in order to be fit for real-time applications, is that it must run in real-time. Needless to say, it is expected



(a) (b)

Figure 2.1: (a) A scene path traced with 1 sample per pixel, showing a large amount of noise. (b) The reference image, showing the same scene path traced with 1024 samples per pixel. The scene is the Cornell Box, first used in [28]

to be run along-side a real-time ray tracer, meaning it may have to operate under very strict conditions in terms of hardware resources and runtime.

Figure 2.1 illustrates the sort of situation that a real-time denoising algorithm must handle. It shows one image created with 1 spp, affordable in real-time, and the reference image, path traced with 1024 spp, where the noise is negligible to a human. Ideally, a real-time denoiser should take the first image as an input and give the second as an output. In practice, the denoiser must work with heuristics to approximate the reference image using a limited amount of information.

Another goal is to ensure *temporal stability*. This goal is shared with offline denoisers, but is a task that requires greater attention in the case real-time denoisers, since the light intensity from a single pixel, even with static camera and scene, can vary wildly across frames.

An early idea for denoising in general, is the use of *temporal coherence* [6]. Sequential frames from any kind of movie or real-time application often holds many similarities, since the camera position and orientation changes slowly. Due to this temporal coherence, many of the ray traced samples from the previous frame can be used again in the current frame. This becomes particularily important in real-time ray tracing, as it drastically increases the effective number of samples per pixel for most frames. Also, human

perception is less vary of image noise shortly after the beginning of a clip or after a change of camera, giving the application some time to gather samples from a few frames to create a convincing result [44].

Guided filtering is a regression method for interactive denoising introduced in 2011 by Bauszat et al. [8]. The main idea is to approximate small blocks of the noisy image as a weighted sum of corresponding small blocks of feature buffers. For each pixel, a least-square linear solution is computed using statistical regression, which yields a set of coefficients for the best linear sum. Subsequently, the coefficients are interpolated at each pixel, and the final illumination value is computed as a linear sum of features at that pixel. The authors also separated direct and indirect illumination, as these two illumination types often depend on the camera movement in different ways.

Mara et al. introduced another real-time method in their paper *An Efficient Denoising Algorithm for Global Illumination* [36]. Their method assumes that the ray tracer traces one direct path to a surface through the center of each pixel, equivalently to traditional rasterization. Furthermore, the second bounce is simulated using two different paths, responding to the matte and glossy parts of the surface texture respectively. These indirect terms are temporarily and spatially filtered separately, The filtering accounts for the fact that matte reflections can tolerate much higher blurring than glossy reflections.

Dammertz et al. introduced a method based on edge-avoiding filtering [21]. This method borrows ideas from *À-trous* filtering and bilateral filtering. It will be described in detail in chapter 3 as part of the SVGF algorithm.

Another class of denoisers are those based on machine learning algorithms.

One such algorithm is NVIDIA's official OptiX denoiser by Chaitanya et al. [12]. The algorithm is based on a neural network that is trained on noisy images and corresponding reference (high spp) images. The network is of the recursive neural network (RNN) architecture, which makes it capable of taking advantage of temporal coherence. Although their achieved visual quality is good, the algorithm is slightly slower than what one would consider comfortable in a real-time application, with more than 50 ms of computation time per frame, putting this in the category of "interactive" denoisers.

Intel Open Image Denoise is an open source CPU-based denoiser that also relies on a neural network for reconstruction [30]. Its performance is heavily restricted by the CPU implementation, and does not achieve higher than interactive frame rates either.

Schied et al. [46] took the edge-avoiding approach of Dammertz et al. [21] further and combined it with spatiotemporal filtering in their algorithm *Spatiotemporal Variance-Guided Filtering* (SVGF). In addition to accumulating samples across several frames, Schied et al. continuously computes the variance of the intensity at each pixel, using this as a heuristic to guide the amount of blurring to conduct throughout the image.

Another method that was recently demonstrated by Koskela et al. to be suitable for real-time denoising, is *Blockwise Multi-Order Feature Regression* (BMFR) [33]. In their approach, they divide the image into equally-sized blocks and perform a regression step within each of these, similarily to Guided Filtering described above, but without interpolating the result at each pixel.

Both SVGF and BMFR will be explained in detail in chapter 3.

Besides Zwicker et al.'s survey, Alain Galvan has also given an extensive summary of the many flavors of modern ray tracing denoising techniques, covering many of the approaches mentioned here [27].

Chapter 3

Two Real-Time Denoising Algorithms

This thesis will now present and investigate properties of two modern real-time denoising algorithms. The chosen algorithms are Spatiotemporal Variance-Guided Filtering (SVGF) by Schied et al. [46] and Blockwise Multi-Order Feature Buffers by Koskela et al. [33].

The reasons for this choice of algorithms, is that they are both modern and fairly general denoisers that have both been considered state-of-the-art for denoising in real-time settings. Their inputs are relatively simple to obtain, and very similar between the two, making comparisons easy. Additionally, they both make few assumptions of the input data, meaning that they are both fit for general-purpose real-time applications.

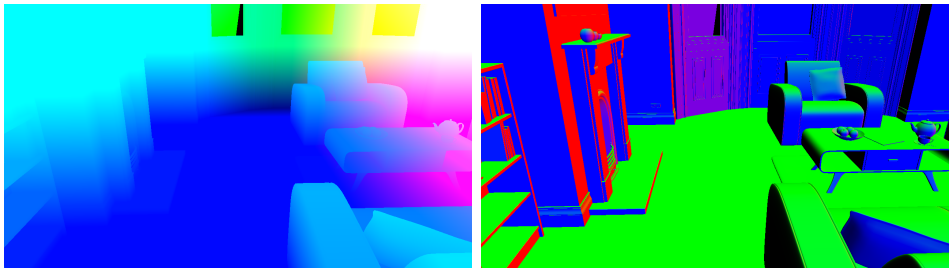
In addition, the two algorithms tackle the problem of denoising in two fundamentally different ways, making for an interesting comparison in terms of performance and output quality.

In this chapter, the two algorithms will be explained in detail. Before diving into algorithms themselves, a short recap on feature buffers is given. Feature buffers play a vital role in both SVGF and BMFR.

3.1 Feature Buffers

As written in section 2.8, feature buffers are images that contain extra information about the scene, like world positions and surface normals. An example of each of the two types is shown in figure 3.1. Other potential feature buffers include gradients, object IDs and buffers depicting motion.

Feature buffers are valuable because they give geometric information about the scene. This information can be used to deduce what areas of



(a) (b)

Figure 3.1: *Two examples of feature buffers: (a) World positions, (b) Surface normals. Each feature consists of three-dimensional values, which are visualized directly as RGB.*

the image may have similar intensities of incoming light. Also, they can usually be constructed cheaply either from the ray tracing engine itself or a rasterizer. For the purpose of this thesis, and as is common for denoising applications, all feature buffers are assumed to be noise free, meaning that there is no randomness involved in the rays' path from the camera into the scene; the rays form a regular rectangular grid. The feature buffers used in this work are all rasterized, which means they are automatically noise free.

3.2 Spatiotemporal Variance-Guided Filtering

At the heart of SVGF is the edge-avoiding À-Trous algorithm for denoising, as it was presented by Dammertz et al. [21]. Therefore, we will start off by describing their work.

3.2.1 Edge-Avoiding À-Trous Filtering

Edge-avoiding À-trous filtering combines ideas from two other filtering techniques: À-trous filtering and bilateral filtering.

À-Trous Filtering

À-trous ("with holes") filtering is a way of approximating large convolution kernels cheaply, avoiding the quadratic dependence on kernel size. In À-trous filtering, the same kernel is used several times iteratively, but with more space between its coefficients for each iteration. Consider the following simple blurring kernel:

$$h_0 = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}.$$

Subsequent kernels h_i are constructed from h_0 for $i \in [1...N-1]$ for a desired N, such that the number of zeros between two neighboring coefficients in the original kernel is always $2^i - 1$. In this example, we would construct h_1 as

$$h_1 = \begin{bmatrix} \frac{1}{16} & 0 & \frac{1}{8} & 0 & \frac{1}{16} \\ 0 & 0 & 0 & 0 & 0 \\ \frac{1}{8} & 0 & \frac{1}{4} & 0 & \frac{1}{8} \\ 0 & 0 & 0 & 0 & 0 \\ \frac{1}{16} & 0 & \frac{1}{8} & 0 & \frac{1}{16} \end{bmatrix}.$$

Similarily, h_2 would have three zeros between each coefficient from h_0, and so on. To perform À-trous filtering with this kernel, we compute $I_{i+1} = h_i * I_i$ for $i \in [0...N]$, where $*$ denotes the convolution operation and I_i are the intermediate convoluted images, I_0 being the input image and I_{N+1} the final result. The effect of the À-trous filtering operation, is approximately that of convolving by a larger kernel with a size on the order of magnitude of that of h_N. The cost is much smaller however, since the number of non-zero coefficients stays constant.

Bilateral Filtering

Bilateral filtering resembles traditional convolution, but adds another weight term to the input pixels which depends on the pixel at the center of the convolution mask. In notation, traditional convolution can be described as

$$R(x,y) = \sum_{(\Delta x, \Delta y) \in U} h(\Delta x, \Delta y) I(x + \Delta x, y + \Delta y)$$

where R is the resulting image, regarded as a two-dimensional function, h is the kernel, I is the input image and U is the set of pixel positions present in h. To be mathematically precise, this is the operation of *correlation* whereas convolution would require the kernel parameters to be negated. The distinction will be ignored here, and it is assumed that h is already flipped or symmetric. General bilateral filtering changes this formula to

$$R(x,y) = \sum_{(\Delta x, \Delta y) \in U} w(x, y, \Delta x, \Delta y) h(\Delta x, \Delta y) I(x + \Delta x, y + \Delta y),$$

where w in general will depend on I and possibly other inputs. A typical use case of bilateral filtering is *edge-preserving noise removal*. The goal is to blur out noise similarily to a Gaussian blur, but not across edges, preserving more of the sharpness in the image. For such a use case, h can be chosen to be the Gaussian kernel, while the weight function may be of the form

$$w(x, y, \Delta x, \Delta y) = \frac{1}{W(x, y)} e^{-\frac{|I(x,y) - I(x+\Delta x, y+\Delta y)|}{\sigma}},$$

where σ is a parameter to adjust the sensitivity and $W(x, y)$ is a normalization constant, chosen to be the sum of $w(x, y, \Delta x, \Delta y)h(\Delta x, \Delta y)$ across the kernel extent evaluated at the given (x, y). The general idea here is that pixels that are more similar to the pixel at the center are given more weight.

Edge-Avoiding À-Trous Filtering

Dammertz et al. combined these concepts with the use of feature buffers and applied them to denoising.

Their approach follows the idea of À-trous filtering, iteratively convolving with increasing mask sizes to smooth out the noise in the image. On top of this, a weight function is applied. The weight function makes use of the feature buffers to guide the smoothing.

The first À-trous kernel is based on the one-dimensional filter

$$g_0 = [\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16}],$$

leading to the 2-dimensional kernel $h_0(i, j) = g_0(i) \cdot g_0(j)$.

The weight function is divided into three different components. The three components account for illumination, normals and positions respectively, denoted by w_{rt}, w_n and w_p. All the three components share the form

$$w_i(x, y, \Delta x, \Delta y) = \exp\left(-\frac{||F_i(x, y) - F_i(x + \Delta x, y + \Delta y)||}{\sigma_i^2}\right),$$

where F_i is a feature buffer or noisy image, and σ_i is a sensitivity parameter. The final weight is computed as the product of these three, which subsequently is multiplied with the À-trous kernel weight. Dammertz et al. also ensures that the sensitivity parameter σ_{rt} for the noisy input image is halved for each iteration. This way, pixels that differ more from the center one is given even less weight, ensuring that the smoothing will focus on small differences in illumination on the later iterations of the algorithm.

3.2.2 The SVGF Algorithm

First, a couple of remarks about the treatment of light in SVGF:

Direct light and *indirect light* are treated separately. Direct light denotes light with only one bounce; the ray from the camera hits a surface and then a light source. Indirect light is all other light, arriving at the visible surface from other non-emitting surfaces in the scene. The separation makes SVGF more capable of handling e.g. mirror-like reflections correctly, because such reflections depend on camera movement differently than diffuse scattering from direct light does. Both input noise buffers and temporary buffers in the algorithm (like variance) have separate versions for direct and indirect light.

Furthermore, all light is filtered without albedo – the color of the directly visible texture. This is done in order to not conduct unnecessary blurring of textures. Instead, the albedo is multiplied with the computed light in the end. The albedo buffer is also noise-free and is constructed in the same manner as the other feature buffers. Note that this does not mean the filtered light is scalar, the light incident on a surface may have different values of the RGB channels.

Here follows a detailed description of the technique itself. The SVGF algorithm is divided into three stages:

- Reprojection
- Variance computation
- À-trous filtering

Each of these will be described in turn.

Reprojection

To get the most out of the traced ray samples, it is useful to reuse the samples of previous frames. This makes sense intuitively, since consecutive frames often display high grade of temporal coherence. Still, one must generally expect the camera to move a non-negligible amount between each frame, so that reusing earlier frames becomes non-trivial.

The process of reusing previous samples has been dubbed *reprojection*, and is used both by SVGF and BMFR.

Reprojection requires various inputs: First and foremost, one needs an image containing the light intensity at each pixel in the previous frame I_{n-1}. This will in general contain accumulated samples from earlier frames

and can be filtered – like in SVGF – or only be an accumulation of noisy samples – like in BMFR. Furthermore, the view-projection matrix $C = PV$ for the previous frame is required, with P being the projection matrix and V the view matrix for the previous camera orientation. Additionally, relevant feature buffers for current and previous frame are needed, including at the very least position buffers. The output of reprojection is the current accumulated buffer I_n.

For a given pixel coordinate (x, y) in the new accumulated buffer I_n, we will first compute the pixel coordinates (x', y') in the previous frame from which we can get the old samples. Using the world position buffer, let $\boldsymbol{p}$ be the world position on coordinate (x, y). We then perform the computation

$$\begin{bmatrix} \widehat{x'} \\ \widehat{y'} \end{bmatrix} = \left(\frac{(C\boldsymbol{p})_{xy}}{(C\boldsymbol{p})_w} \right) \cdot 0.5 + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

Here, $\widehat{x'}$ and $\widehat{y'}$ are the pixel positions in the previous frame, normalized to be between 0 and 1. $(C\boldsymbol{p})_{xy}$ and $(C\boldsymbol{p})_w$ denote the two-dimensional vector consisting of x- and y-coordinates of $C\boldsymbol{p}$, and the scalar w-component respectively. To get pixel coordinates, they are multiplied by the image resolution along the x- and y-axes respectively. Coordinates outside the previous frame are discarded.

To decide whether a sample value can be reused for the current frame, the feature buffers are consulted for more information. The world positions $\boldsymbol{p}$ at (x, y) in the current frame and $\boldsymbol{p'}$ at (x', y') in the previous are compared to see if the sample value originates from the same place. If $||\boldsymbol{p} - \boldsymbol{p'}||$ is higher than a set threshold, the previous samples are discarded. This happens e.g. if the origin of the previous sample is occluded in the current frame. If the positions correspond, other features like surface normals or object IDs can also be checked. SVGF and BMFR both check surface normals as well, which may be important because the amount of reflected light in a given direction is very dependent on the surface normal. If all checks pass, the sample values from the previous frame are stored in I_n at position (x, y).

In general, the coordinates x' and y' in the previous frame will not be integers. Therefore, the samples at the four neighboring pixels $(\lfloor x' \rfloor, \lfloor y' \rfloor)$, $(\lfloor x' \rfloor + 1, \lfloor y' \rfloor)$, $(\lfloor x' \rfloor, \lfloor y' \rfloor + 1)$ and $(\lfloor x' \rfloor + 1, \lfloor y' \rfloor + 1)$ are each evaluated and added together using bilinear weights, which are eventually renormalized to 1 to account for discarded samples.

During reprojection, SVGF and BMFR also uses motion buffers to account for dynamic scenes. Motion buffers hold a 2D motion vector on each pixel telling the change in screen-space position for thatpixel from the pre-

vious frame to the current, which is crucial for reprojecting samples from moving objects correctly.

Eventually, when the previous samples have been accumulated into the new frame, the new noisy input is also blended in. This is done with an exponentially moving average: $I'_n(x,y) = (1-\alpha)I_n(x,y) + \alpha J_n(x,y)$, where I'_n is the final output image and J_n is the noisy input for the current frame. α is a blending constant, typically chosen to be around 0.2. For the first few samples, it is common to instead use an arithmetic mean, weighing all samples, new and old, equally to avoid giving the first frames too much weight.

Variance Computation

As the name suggests, Spatiotemporal *Variance-Guided* Filtering uses variance of light on each pixel to control the À-trous filtering. The key idea is that pixels that display little variation should not need to be filtered as strongly as pixels with high amounts of variation. A buffer containing the *second moment* of the noisy input is maintained and reprojected along with the accumulated buffer. The second moment is the noisy input samples squared; $S_n(x,y) = J_n(x,y)^2$. This value is blended and accumulated just like the light samples are in reprojection.

The second moment buffer is used to compute an estimate of the variance using the well-known formula $\mathrm{Var}(X) = \mathrm{E}[X^2] - \mathrm{E}[X]^2$. Specifically, the variance at pixel (x,y) is $S'_n(x,y) - I'_n(x,y)^2$ where $S'_n(x,y)$ is the accumulated and blended second moment at the current frame.

When few samples have been accumulated at a given pixel, its variance is instead computed using its neighbors by a bilateral filter with feature weights. The temporal variance is thus only estimated by spatial variance until a more accurate estimate has accumulated.

À-trous Filtering

This step largely follows the procedure of edge-avoiding À-trous filtering as described earlier, with a few differences.

In the À-trous iterations, variance is computed together with and in the same way as the new light value. The variance comes into play in the component of the weight function that depends on incoming light: The sensitivity parameter σ_{rt} for light is multiplied by the standard deviation, the square root of the variance. To avoid problems with instability of the estimate, the variance is first blurred in a small spatial neighborhood.

Multiplying with the standard deviation serves the purpose of smoothing more aggressively in areas that display high variance. As the kernel size increases with the number of iterations, the variance at each pixel is expected to decrease, softening the blur effect.

Another modification to the edge-avoiding À-trous algorithm is this: Instead of feeding the bilateral weight functions with world position, they solely use the depth of the image. Furthermore, they account for the gradient of the depth image by multiplying σ_p, the world depth sensitivity parameter, by the dot-product of the depth gradient and the displacement vector from the kernel center. The depth gradient is computed in clip-space. This step ensures that scenes that contain details at many different scales will be handled correctly.

The output of the last À-trous iteration is the final illumination image. SVGF keeps the output of the *first* À-trous iteration, this is the image that will serve as input to the reprojection step in the next frame.

The two separate light buffers, for direct and indirect light, are combined and multiplied (modulated) with the albedo buffer to produce the image that is sent to the display.

3.3 Blockwise Multi-Order Feature Regression

BMFR takes a slightly different route to tackle the problem of denoising, namely by regression. The basics of the approach is that the frame is divided into blocks, and within each of these blocks, the algorithm tries to write the noisy buffer as a linear sum of features. The blocks will, for all intents and purposes, have the size 32×32. Before explaining BMFR step by step, a deep-dive into this regression step is given.

3.3.1 Features and Regression

The notion of "feature" in BMFR is slightly different from that of "feature buffers" in SVGF. Here, features are only scalar buffers. Thus each of the different components of e.g. world position is now its own buffer. In addition to the aforementioned feature buffers like positions and normals, features in BMFR can include any sort of information about the scene, including mesh IDs, material properties or gradients. Furthermore, the authors of BMFR add common feature components raised to a power, e.g. the squared x coordinate of the world position, as features, and also a constant buffer where every element is 1. The purpose of adding multiple new features is to increase the accuracy that can be achieved when approximating the noisy

buffer with features.

Let $\mathcal{F} = \{F_i \,|\, i \in [1, N] \subset \mathcal{Z}\}$ denote the set of N features, each of which is represented by a two-dimensional function. Let C denote the noisy accumulated buffer. Let Ω be the set of pixel coordinate pairs that are contained within some 32×32 block in the image. The result we want to achieve, is a least-squares approximation of C using the features in $\mathcal{F}$, restricted to the pixel coordinates residing in Ω. In notation we write

$$\boldsymbol{\alpha}' = \text{argmin}_{\boldsymbol{\alpha}} \sum_{(i,j)\in\Omega} \left(C(i,j) - \sum_{n\in[1,N]} \boldsymbol{\alpha}_n F_n(i,j) \right)^2$$

where $\boldsymbol{\alpha}'$ is the coefficients of the optimal linear combination within the block Ω and $\boldsymbol{\alpha}_n$ denotes the n'th component of $\boldsymbol{\alpha}$.

To solve this, Koskela et al. writes the buffers as a matrix by reshaping each block as a column of W elements. In this case, $W = 32 \cdot 32$. The columns vectors corresponding to each feature is then concatenated into a single $W \times N$ matrix T. Let T_i denote the i'th column of T. Similarily, let $\boldsymbol{c}$ be a column representing the noisy buffer within the block. Then we can restate the optimization objective as

$$\left(\boldsymbol{c} - \sum_{i\in[1,N]} \boldsymbol{\alpha}_i T_i \right)^2 \tag{3.1}$$

Now we concatenate the column $\boldsymbol{c}$ to T, giving us a new $W \times (N+1)$ matrix $\widehat{T}$. The matrix $\widehat{T}$ is factorized by QR-decomposition, $\widehat{T} = Q\widehat{R}$, where Q is a $W \times (N+1)$ matrix with orthonormal columns and $\widehat{R}$ is an $(N+1)\times(N+1)$ upper-triangular matrix. Let R be the left-most $(N+1)\times N$ matrix of $\widehat{R}$, and $\boldsymbol{r}$ the right-most $(N+1) \times 1$ column. Let R_i be the i'th column of R. Then we have $T_i = QR_i$ and $\boldsymbol{c} = Q\boldsymbol{r}$. Consequently, the optimization objective (3.1) becomes

$$\left(Q\boldsymbol{r} - \sum_{i\in[1,N]} \boldsymbol{\alpha}_i Q R_i\right)^2 =$$
$$\left(Q\left(\boldsymbol{r} - \sum_{i\in[1,N]} \boldsymbol{\alpha}_i R_i\right)\right)^2 =$$
$$\left(\boldsymbol{r} - \sum_{i\in[1,N]} \boldsymbol{\alpha}_i R_i\right)^T Q^T Q \left(\boldsymbol{r} - \sum_{i\in[1,N]} \boldsymbol{\alpha}_i R_i\right) =$$
$$\left(\boldsymbol{r} - \sum_{i\in[1,N]} \boldsymbol{\alpha}_i R_i\right)^2, \tag{3.2}$$

where the last step follows from the orthonormality of Q. Since the lower-most row of R is only zeros, the last component of $\boldsymbol{r}$ gives a lower bound on the error of this objective function. However, ignoring the last row, we have an upper-triangular $N \times N$ matrix R' and an $N \times 1$ column $\boldsymbol{r'}$. It is apparent that the minimization of objective expression (3.2) must be the solution to the equation $R'\boldsymbol{\alpha} = \boldsymbol{r'}$. Since R' is upper-triangular, the equation is easily solved for $\boldsymbol{\alpha}$.

To reiterate on the process once more: In order to find a least-squares approximation of C as a linear combination of buffers F_i, the pixel values within the block are ordered into columns, which are concatenated to a $W \times (N+1)$ matrix $\widehat{T}$. Then the QR-decomposition of $\widehat{T}$ is found – the authors suggest using the Householder's algorithm for this – and the coefficients $\boldsymbol{\alpha}$ of the linear combination can be found with a straight-forward back-substitution using R. Since Q is irrelevant for the computation, it does not have to be computed for the purpose of the algorithm.

Also, it is important that the matrix $\widehat{T}$ is not singular. To prevent this, the authors add zero-mean noise to all the feature buffers for the linear regression part. When computing the actual weighted sum later, the original noise-free feature buffers are used instead.

3.3.2 The BMFR algorithm

The description of BMFR is divided into five stages:

- Accumulate noisy samples

- Linear regression
- Compute weighted sum
- Accumulate filtered samples
- Temporal anti-aliasing

Accumulate noisy samples

This step largely follows the procedure of reprojection as described in section 3.2.2. The output of this stage is an accumulated noisy buffer which serves as input to the next stage in the pipeline, but it also, unlike in SVGF, serves as the accumulated buffer of reprojection in the next frame.

Linear Regression

The image is separated into regularly-sized blocks, and then the algorithm follows the procedure outlined in section 3.3.1. The noisy buffer to be approximated is the accumulated noisy output from the previous stage. To avoid problems with unbounded buffers, which is typically the case for the position buffer, a distinction is made between buffers that are to be scaled and buffers that are to be left unscaled. The scaling happens within each block separately and happens by computing the maximum and minimum of each scalable buffer and normalizing it to the range $[0, 1]$.

The result from this stage is the min-max values for each scalar buffer, which are reused in the next stage, in addition to the coefficients for the linear combination computed in the regression.

Compute Weighted Sum

Here, the actual linear combination is computed, and the output is a new frame approximating the accumulated noise with the feature buffers, using the corresponding coefficients within each block.

Accumulate Filtered Samples

At this point, the output buffer contains unacceptable "blocky" artifacts. This step is made to get rid of these artifacts. The preparations for this stage is done already at the separation of the image into blocks. The block grid is given a different offset every frame, circulating through a list of 16 different coordinate offset pairs, where each coordinate is between -16 and 15.

To take advantage of the variation in offsets, the weighted sums are also filtered together over time. This step largely performs the same reprojection operation as the first stage, but reuses information such as discarded samples and pixel location in previous frame, and it operates on a different accumulated buffer.

Due to the accumulation and blending in this stage, and with the offsets changing every frame, the blocky artifacts do not get the chance to form at any specific location on the screen over time, even if the camera is static.

Temporal Anti-Aliasing

The purpose of temporal anti-aliasing is to, once again, reuse information from the previous frames, but this time for the purpose of removing flickering ("fireflies") or excessive variance in the pixels over time. Such effects can be very distracting to a viewer.

In essence, the reprojection phase is repeated again, but this time without checking if the samples from the previous frame correspond to the same object in this frame. The final result is a bilinear sampling of the filtered buffer of the previous frame, clamped to a range of values based on values in a small local neighborhood around the new filtered pixel to avoid that it sticks out. Finally, it is blended together with the previous temporally anti-aliased buffer.

The authors note that this stage does not actually increase the score when evaluated with objective quality metrics, but gives "more visually pleasing results".

Chapter 4

Test Setup

To address the remaining research goals, two experiments will be conducted. The goal of the first experiment is evaluate and compare SVGF and BMFR with respect to performance and output quality. The second experiment will evaluate the potential improvement in quality resulting from a proposed extension to BMFR.

This section will describe the setup used for running and evaluating the two denoising algorithms. It will discuss the construction of inputs to the algorithms, the scenes that are used, details on the implementations, details on metrics and finally, a description of the two experiments.

4.1 Data Construction

4.1.1 Ray Tracer Engine

In order to test denoising algorithms, there is a need for ray tracing images with a suitable low sample count per pixel, and corresponding reference images to compare the result of the denoising. The reference images will, as is common in the literature, be created with the same engine, but with a much higher sample count per pixel, e.g. 1024 spp.

When choosing an engine to use in my experiments, finding a renderer that targets real-time applications is emphasized, to ensure that the rendering process holds a realistic level of simplifications and heuristics.

The chosen ray tracer engine is Will Usher's ChameleonRT project [48]. It implements a single interface a multitude of ray tracing backends, including Intel's Embree [3] and OSPRay [20], NVIDIA's OptiX [40], Microsoft's DirectX Ray Tracing [1] and Vulkan [32]. The project both contains a basic interface for rendering different kinds of scenes and supports several real-

time rendering frameworks. While having numerous backends is a notable freedom, it is unlikely to have noteworthy benefits to this work. The work presented here will restrict itself to the OptiX backend in ChameleonRT, since it is GPU-accelerated, making for relatively fast production of reference images, and has a focus on real-time applications.

4.1.2 Constructing Feature Buffers

In addition to a ray-traced input image, SVGF and BMFR requre feature buffers. Additionally, the feature buffers must be noise-free, as has already been mentioned.

ChameleonRT does not offer a unified interface for turning off such randomization for the initial rays. In terms of performance, this is also a task better suited to a rasterization engine. For rendering feature buffers, Sascha Willem's Vulkan-glTF-PBR project [54] has therefore been deployed. It is a Vulkan-based rasterizer with support for physically-based rendering (PBR). Full PBR shading is unnecessary for producing feature buffers, but because the fragment shader takes many different vertex attributes as input, it is easy to modify into displaying a variety of features of the scene.

4.1.3 Unified Data Construction Repository

The input data construction is handled by a unified code repository. The repository includes the modified versions of the ChameleonRT and Vulkan-glTF-PBR code repositories, plus a number of utilities. Links to the source code can be found in the appendix.

A few modifications were made to both Vulkan-glTF-PBR and Chameleon-RT to fit the requirements of the denoiser inputs. First and foremost, both repositories have been modified to write high dynamic range (HDR) images to disk. Both in the case of the feature buffers and the noisy input buffer, the output images should hold values on a floating point scale. This is important in case of rendering HDR content, and even more important for features in feature buffers that are not bounded, like world position. To support the required precision, images are written in OpenEXR format [25]. OpenEXR is a image file format that can store an arbitrary number of channels containing 16-bit floating point values.

Furthermore, both renderers are modified so that the camera follows a user-given path. The path is specified through a JSON file with "checkpoints" describing a camera direction and position along with an integer time stamp. Intermediate transformations are computed on each integer

time stamp, linearly interpolating position and view direction, and an output image is generated on each such transformation.

4.2 Scenes

In the experiments, three scenes are used. The scenes are Crytek Sponza, Living Room and San Miguel 2.0, all provided by Morgan McGuire's Computer Graphics Archive [37].

The Living Room and Sponza scenes are indoor scenes with a reasonably low polygon count, around what can be considered realistic for a real-time application. Sponza have ornaments, some smaller objects and hanging pieces of clothes that introduce some complexity to light paths. Living Room is slightly more complex, containing many smaller objects.

San Miguel is significantly more complex and contains many fine-grained details, including foilage, in the form of leaves, thin geometry in cloths and chairs, and reflective materials in cuttlery.

Each scene is given one path for the camera to follow. The paths each generate a series of 60 frames, where the camera is flying continuously through the scene. A usual target framerate for real-time applications is 60 frames per second, so that the movement captured in the 60 recorded frames are meant to represent one second worth of application time. Thus, the movement is reasonably limited and the imagery display large amounts of temporal coherence.

The screen resolution will be 1280×720 for all images, including noisy input, feature buffers and results.

Each scene is equipped with a single static rectangular light source.

4.3 Image Quality Evaluation

In addition to measuring the performance of each algorithm, the experiments will evaluate the quality of their outputs. The ideal metric for this cause is to find the resemblance between the output from the denoiser and the reference image *as perceived by a human*. Naturally, this is not a quality that is easily quantifiable, and can at best be approximated through conventional *similarity metrics*. Multiple different metrics are deployed for the job, in hope that they will complement each other's weaknesses: Root mean square error (RMSE), Structural Similarity (SSIM) [56], temporal error [46] and Video Multi-Method Assessment Fusion (VMAF) [55]. Each of these will be briefly explained in the following.

4.3.1 Root Mean Square Error

The simplest metric is the root mean square error (RMSE). In this scenario, it compares the output from the denoising procedure and the reference image to produce the metric E_{RMS} as follows:

$$E_{\mathrm{RMS}}(\mathbf{I}, \mathbf{R}) = \sqrt{\frac{\sum_{x,y \in \Omega} \left(\mathbf{I}(x,y) - \mathbf{R}(x,y)\right)^2}{N}},$$

where $\mathbf{I}$ and $\mathbf{R}$ are the denoised image and the reference image respectively, regarded as functions from pixel coordinates to a vector of e.g. RGB values or a scalar value like luminance. Ω is the set of pixel coordinates within $\mathbf{I}$ and $\mathbf{R}$, which are assumed to be of equal size. N is the number of pixels in each image, or $|\Omega|$.

E_{RMS} is a very simple metric, and one cannot in general assume that a higher E_{RMS} means that the images are perceived as more different that if E_{RMS} was lower. For example, changes in some areas of the image will affect perception more than others, and the human visual system puts emphasis on relations between neighborhoods of pixels rather than individual pixel values, something this metric does not reflect. It will be included nevertheless, as it is trivial to compute and may highlight interesting characteristics of different algorithms that are easy to overlook otherwise.

RMSE will give a score of 0 if and only if the two inputs are equal. The maximal score is the dynamic range of the pixel values. For the purpose of this thesis, the dynamic range for result images will be 1.0.

4.3.2 Structural Similarity

Structural Similarity (SSIM) was introduced by Wang et al. [56] and is a method for comparing patterns within two pictures. The metric is split into three components: *Luminance*, *contrast* and *structure*. The SSIM score will be computed separately for square windows of size 11×11 throughout the whole image space, and finally averaged to give a single score for the similarity between the two metrics. To avoid that the score for the images is affected by the block structure of the windows, the contributions for the different pixels in a window will be weighted by a normalized Gaussian weighting function with standard deviation 1.5 pixels centered in the middle of the window.

Now follows a description of the different components of the metric as proposed by [56]. Let any two corresponding 11×11 windows in the two pictures be denoted by $\mathbf{I}$ and $\mathbf{R}$, with $\mathbf{I}(i)$ and $\mathbf{R}(i)$ being pixel at index

i within the two windows, by an arbitrary but consistent ordering. w_i will denote the Gaussian weight assigned to that same pixel, and N is the number of pixel in each window, which is always 121 in this case. The mean and standard deviation for window $\mathbf{I}$ are defined as

$$\mu_I = \sum_{i=1}^{N} w_i \mathbf{I}(i)$$

$$\sigma_I = \sqrt{\sum_{i=1}^{N} w_i (\mathbf{I}(i) - \mu_I)^2}$$

respectively, and μ_R and σ_R are defined analogously. The covariance between the two windows is defined as

$$\sigma_{IR} = \sum_{i=1}^{N} w_i (\mathbf{I}(i) - \mu_I)(\mathbf{R}(i) - \mu_R)$$

With these definitions in place, one can compute the different components of the metric. The luminance part is computed as

$$l(\mathbf{I}, \mathbf{R}) = \frac{2\mu_I \mu_R + C_1}{\mu_I^2 + \mu_R^2 + C_1}$$

where C_1 is a constant added for improved stability. In work, as suggested by Wang et al., C_1 is defined by $C_1 = (K_1 L)^2$ where L is the dynamic range for the pixel values (e.g. 255 for 8-bit colors) and $K_1 = 0.02$. In essence, this part indicates how well the overall light intensities in the two windows correlate.

The contrast part of the metric is computed as

$$c(\mathbf{I}, \mathbf{R}) = \frac{2\sigma_I \sigma_R + C_2}{\sigma_I^2 + \sigma_R^2 + C_2},$$

where $C_2 = (K_2 L)^2$, $K_2 = 0.03$. The structure of this expression is identical to the one for luminance, and indicates how well the intensity ranges of the two windows correlate.

Finally, the structure part of the metric is computed as

$$s(\mathbf{I}, \mathbf{R}) = \frac{\sigma_{RI} + C_3}{\sigma_R \sigma_I + C_3},$$

where $C_3 = C_2/2$. This expression closely resembles the expression for statistical correlation, and its purpose is exactly that, indicating how well

the overall pixel intensities correlate throughout the image. This is the only part of the metric that is not restricted to the value range $[0, 1]$, but spans all of $[-1, 1]$.

To conclude, the *Structure Similarity* between the two corresponding windows $\mathbf{I}$ and $\mathbf{R}$ is simply

$$\mathbf{S}(\mathbf{I}, \mathbf{R}) = l(\mathbf{I}, \mathbf{R}) \cdot c(\mathbf{I}, \mathbf{R}) \cdot s(\mathbf{I}, \mathbf{R}).$$

By inspection of each individual part, it can be inferred that this metric only gives 1 as a result if the two windows matches completely, and that it is symmetric in its arguments.

Finally, the score for the entire image will be the average of the scores across all windows. Wang et al. propose using windows with a distance of 1 pixel from one another, meaning that the expression will be evaluated for *every possible* 11×11 window in the image before eventually computing the average.

4.3.3 Temporal Error

Schied et al. [46] constructed a metric to measure the *temporal stability* of their algorithm. Here, temporal stability refers to how consistently a reconstruction algorithm outputs the same light intensity over time in areas where the intensity should be constant, such as purely diffuse materials in constant lighting, or a static scenes viewed with a static camera. Their metric has later been referred to as the *temporal error* metric. The metric measures the average luminance of the absolute difference of subsequent frames, where both the camera and the scene are static. It shares elements with RMSE, including the fact that a score of 0 is the ideal case and a score of 1 is the worst possible.

4.3.4 Video Multi-Method Assessment Fusion

Video Multi-Method Assessment Fusion (VMAF) is a video similarity metric that aims to accurately measure the similarity as perceived by humans. VMAF is an open-source effort initiated by Netflix, primarily meant for evaluating the perceivable effect of video stream compression. Besides video streaming, it is also considered useful for evaluating quality of image reconstruction, and contrary to the other metrics discussed here, it is built on data directly related to humans' subjective perception.

Instead of constructing a new algorithm by hand, the authors of VMAF combine several other similarity metrics by giving them individual weights.

The weights are found through machine learning, using a Support Vector Machine (SVM) regressor.

The VMAF model used here is 0.6.1. The SVM is trained by the VMAF authors on a dataset of uncompressed and corresponding compressed video streams, annotated with subjective similarity ratings made manually. VMAF outputs a score between 0 and 100, where 100 indicates perfect equality and 0 indicates no similarity. Because of the machine learning aspect of VMAF, its score should not be interpreted as exact. For instance, two identical image sequences have been found to give scores around 98.

4.4 Implementations

BMFR is evaluated using the implementation published by the original authors. The algorithm runs on the GPU through OpenCL. A few minor changes to the code has been made for the sake of usability, but the algorithm itself remains untouched.

SVGF is evaluated using an implementation made by myself, also using OpenCL. This SVGF implementation closely resembles the sample program published by the authors of the original algorithm, which was made with Microsoft's DirectX Ray Tracing and HLSL. This implementation simplifies the original SVGF algorithm in several aspects, to boil down the input requirements to match those of BMFR, easing the test process:

First, the algorithm does not take into account moving objects in the scene, which makes its capabilities more similar to the BMFR implementation, which does not consider moving objects either. Adding support for dynamic scenes can be done, also for BMFR, by adding supporrt for motion vectors.

Secondly, and with practical consequences for the comparison, the implementation does not separate the light into direct and indirect light. This means that the implemented algorithm could be faster than the authors' sample, but the difference is thought to be minimal, as the separate operations can be easily parallelized on vector arithmetic hardware. It is expected that it will have some implications on the resulting quality.

Performance evaluation is done for each stage of the algorithms separately. Timing is done using OpenCL-native timers to ensure that the times are as precise as possible.

Links to the source code can be found in the appendix.

The experiments were conducted on a system with 12-core Intel Core i7-6800K@3.40GHz and an NVIDIA TITAN X (Pascal) graphics card.

4.5 Experiments

For this thesis, two experiments have been conducted. The first experiment are evaluations and comparisons between the two algorithms BMFR [33] and SVGF [46], both in terms of speed and output quality. The second experiment concerns a modification of the BMFR algorithm. The goal of the experiment is to analyze the impact of the modifications on output quality.

The experiments will be described in the following.

4.5.1 Evaluating and Comparing the Algorithms

In this experiment, the goal is to evaluate and compare the effectiveness of the two algorithms quantitively and qualitatively, in terms of computation time and the resulting image quality. SVGF and BMFR are run wtih the same set of noisy sample buffers and feature buffers.

The runtime for each scene combined with each algorithm will be reported. It is not expected that the variation in computational cost between scenes will be tremendous, as the scene complexity does not directly influence the denoisers' problem size. It can still lead to some variations however, especially since SVGF will do extra computations to compensate for regions with low sample counts.

Subsequently, comparisons between the output of two algorithms are made on the basis of the quality metrics described earlier. The resulting images will also undergo manual inspection and a qualitative discussion will be given. Note that a similar comparison has been conducted earlier by the authors of BMFR, Koskela et al., in [33], where they used SVGF as one of the baselines for BMFR.

4.5.2 Exploring Feature Buffers for BMFR

The second experiment is of a more exploring nature, and aims to determine whether another choice of input buffers for the BMFR algorithm could yield better output quality.

The original authors deviced a separate algorithm to decide the final choice of feature buffers: First, the authors made a collection of scalar feature buffers, each containing a single dimension from original feature buffers, such as positions or surface normals. In addition, different orders of these scalars were added, e.g. the squared x-component of the world position, together with a feature buffer only containing ones. This initial set of buffers is called the *feature pool*.

The feature-choosing algorithm initializes an empty list of "active buffers", which is iteratively extended with new feature bufers. In each iteration, each feature in the feature pool that is not already appended to the list of active buffers, is evaluated as one of the feature buffers in BMFR together with the chosen active buffers. After running BMFR with each of these unchosen buffers, the buffer that had the most positive impact on the objective output quality is pushed on to the list of active buffers. The algorithm continues until the number of active buffers reaches a preset maximum.

While this greedy approach is not guaranteed to find the optimal collection of feature buffers, it is likely to be a reasonable heuristic, and it will be the algorithm of choice in the search for a better collection of feature buffers.

There is one aspect with which extra care must be taken. The authors of BMFR reported that they add all components of a buffer at the same time. That is, their greedy algorithm adds e.g. all world position at the same time, instead of adding the three components separately. This is to avoid that the camera orientations in the image sequence that are used in choosing feature buffers, influences the choice too much. Adding new features in that manner with the new proposed buffer pool will be hard, since each feature is in general a function of several components. Therefore, each scalar feature will be treated entirely separately, which makes testing the final choice on other scenes and camera orientations important.

The pre-determined feature maps in the original work included world positions, surface normals and position gradients. The position gradients were found to contribute little to the output quality, and so the final 10 feature buffers were the full set of coordinates from world positions and surface normals, as well as the squared world positions and the constant 1.

The variation in the feature pool for the sake of this experiment will be more extensive than that of the original work. In addition to different integer powers of scalar components from the feature buffers, the following will be included in the initial feature collection:

- Features raised to non-integral powers, namely the square root of the absolute value of the feature.
- Products of different feature components, as was already suggested in the original BMFR article.

The base features will still be the same as in the original work: World positions and surface normals.

It was considered to add sums and differences between e.g. two different components of the world position coordinates as its own feature, but the

idea was halted due to two theoretical observation: They are linear functions of several buffers, and thus can't possibly lead to a better approximation than those produced if the two buffers were added separately because the linear regression step already computes the optimal linear combination of all buffers in the pool.

Nevertheless, it was hypothesized that one feature that is the linear combinations of two other features could replace the two features and let the algorithm produce similar results with one less buffer, resulting in slightly cheaper computations. However, for the geometric features discussed here, a linear combination of any two of them would be possible to reproduce with the two separate features by a linear transformation of the scene, like a rotation. Thus, whether the new buffer could reproduce the same quality when replacing the other two buffers, would be entirely dependent on the scene and camera orientation, which is far from ideal for a general-purpose denoising algorithm.

The greedy feature-choice algorithm also requires an objective metric to evaluate feature buffer candidates. In this experiment, VMAF is used in evaluation. Due to its inaccuracies it is not ideal, but it is considered the best at capturing details that are important to human perception.

Chapter 5

Results and Discussion

The results of the comparison and extension experiments are presented sequentially. The results will be discussed both from a qualitative and quantitative perspective.

5.1 Comparison between SVGF and BMFR

5.1.1 Performance

Runtimes for each of the two methods run on the three different scenes are shown in table 5.1. The times are averages over a sequence of 60 consequtive images. For each image sequence, the average runtime is shown together with the minimum and maximum time for a single frame. Each frame has a resolution of 1280×720. All measurements are given in milliseconds.

We can first note that both algorithms clearly meet the goal of a real-time execution time. With a target framerate of 60 frames per second

Scene	SVGF (ms)			BMFR (ms)		
	min	mean	max	min	mean	max
Sponza	5.208	5.865	8.109	2.079	2.245	2.471
Living Room	4.467	5.208	8.101	2.096	2.299	2.515
San Miguel	5.319	5.908	8.139	2.083	2.282	2.502

Table 5.1: *Run times for the BMFR and SVGF algorithms on three different scenes. The reported numbers are the minimum, average and maximum time used per frame in milliseconds, for runs on 60-frame sequences.*

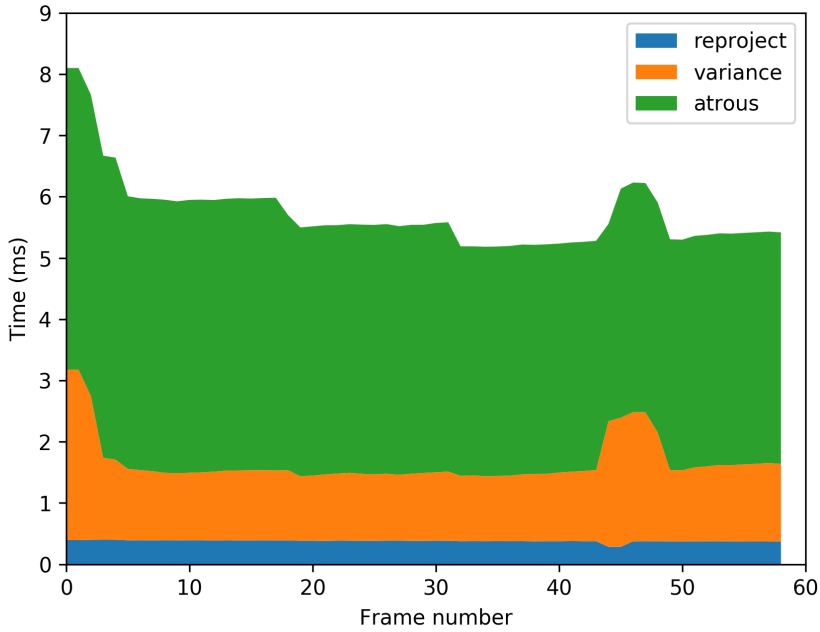


Figure 5.1: *Runtime for the SVGF algorithm, distributed on the different stages of the algorithm. The scene in this run was Sponza.*

(fps), the algorithms must use less than 16 milliseconds per frame, which is definitely the case. Secondly, there is a significant difference in execution time between the two algorithms. The BMFR algorithm consistently falls under 2.6 ms execution time, whereas SVGF averages between 5 and 6 ms.

Additionally, the execution time of SVGF shows greater variation, as the maximum run times are much higher than the average; 43.9% above the mean score, averaged over all three scenes, as opposed to an average of 10% above the mean score in the BMFR algorithm.

Figures 5.1 and 5.2 show the runtime of SVGF and BMFR respectively, distributed on the different stages of the algorithm, as a function of frame number. These numbers were generated from the path on the Sponza scene.

Focusing on the runtimes for SVGF, the figure gives more context to the great difference between maximum and average run times. The above-average runtimes are mostly restricted to the first frames. Furthermore, the run time difference mainly stems from the `variance` stage, which is reasonable, as the algorithm computes a local average if the number of

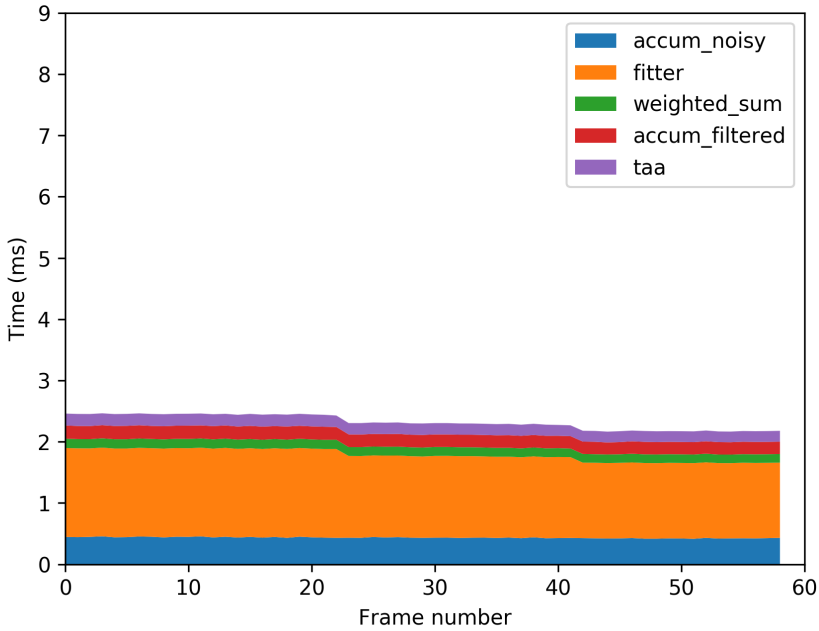


Figure 5.2: *Runtime for the BMFR algorithm, distributed on the different stages of the algorithm. The scene in this run was Sponza.*

samples is found to be too small. As the frames go by and the average number of accumulated samples increase across the image, this spike in runtime diminishes. Another, smaller spike in the same stage can be seen near the end of the run, probably due to less temporal coherence between the frames.

While a severe performance variation may be detrimental to some real-time applications such as certain video games, the above result shows that none-average computation times are restricted to the beginning of an image sequence, and are otherwise rather small. At the beginning of an image sequence or at the change of camera, a user must also adapt to the change of view, likely giving them little chance of noticing a small performance drop, especially since the drop would only be noticable during the five first frames or in an application that preferrably runs at 60 frames per second.

It can also be seen from figure 5.1 that the `atrous` stage is the most computationally heavy part of the algorithm. This is not surprising, as this stage performs four iterations with the À-Trous compute kernel. This suggests that lowering the number of iterations with the À-Trous kernel would be a reasonable optimization, but it will still lack severely in performance compared to BMFR.

Shifting our attention to the performance of BMFR and figure 5.2, we notice that the overall performance is very stable, in addition to being lower than that of SVGF. We see clearly that the `fitter` stage, where the linear regression fitting the feature data to the noisy data is performed, is the most computation-heavy stage of the algorithm. This stage alone has a $O(N^2)$ dependency on the number of feature buffers, suggesting that reducing the number of feature buffers is an effective optimization performance-wise.

5.1.2 Image Quality

Table 5.2 summarizes the quantitative image quality of the output from the two algorithms. It shows that BMFR outperforms SVGF on the Sponza and Living Room scenes while SVGF makes a better job than BMFR on the more complex scene San Miguel. Interestingly, BMFR seems to have a significant drop in quality on the San Miguel scene when evaluated with VMAF, while the two other metrics show a less significant change.

Figure 5.3 shows one of the last images in the image sequence from the scene San Miguel. The figure shows, from top to bottom, the image reconstructed with SVGF and BMFR and, lastly, the reference image. The

Scene	SVGF			BMFR		
	RMSE	SSIM	VMAF	RMSE	SSIM	VMAF
Sponza	0.031	0.888	53.54	0.029	0.889	59.26
Living Room	0.045	0.841	54.49	0.035	0.872	66.51
San Miguel	0.033	0.867	51.58	0.037	0.834	36.30

Table 5.2: *Average score on different image stream metrics for each combination of scene and algorithm, run on a 60-frame image sequence. The range of each metric is as follows – RMSE: [0, 1], lower is better; SSIM: [-1, 1], higher is better; VMAF: [0, 100], higher is better.*

crop-outs on the right put additional emphasize on some of the key differences between the results of the two algorithms.

For context, the camera is turning towards the left in this part of the sequence, meaning that the left part of the image will have a lack of accumulated samples. The left-most crop shows that this has the strongest effect on the BMFR algorithm, which shows significant artifacts even though the image region has had 4-5 frames of accumulated samples. What makes this scenario particularily bad, is that this part of the scene receives little light intensity overall, creating potentially large relative variation in illumination across frames. This seems to make a bad fit for BMFR, perhaps because it relies on blocks having a somewhat consistent illumination over time when filtering.

The second crop shows a table with fine-detailed objects. It can seen that BMFR picks up more of the geometric details on the table cloth. However, it seems to blend the light reflection off the bottle and glasses too much, and loses more detail than SVGF. SVGF probably outperforms BMFR because it takes the variation of the pixel over time into account, which makes it better equipped to handle specular reflections, which are highly dependent on viewpoint. In this case, the SVGF algorithm would likely perform even better if the separation of direct and indirect light had been implemented.

As of now, both implementations show significant differences from the reference. It is evident that the fine details of the objects are hard to reproduce for the two algorithms.

In figure 5.4, we see an image from the middle of the sequence from the scene Living Room. The order is the same as before – from top to bottom: SVGF, BMFR and reference. The left-most crop-out magnifies the corner of the stove by the three algorithms. It is evident here that SVGF has a harder time separating the lighting from the two different sides of the corner, over-blurring across the edge. BMFR seems to tackle this *corner case*

Figure 5.3: *One of the final images from San Miguel. From top to bottom: SVGF, BMFR, reference.*

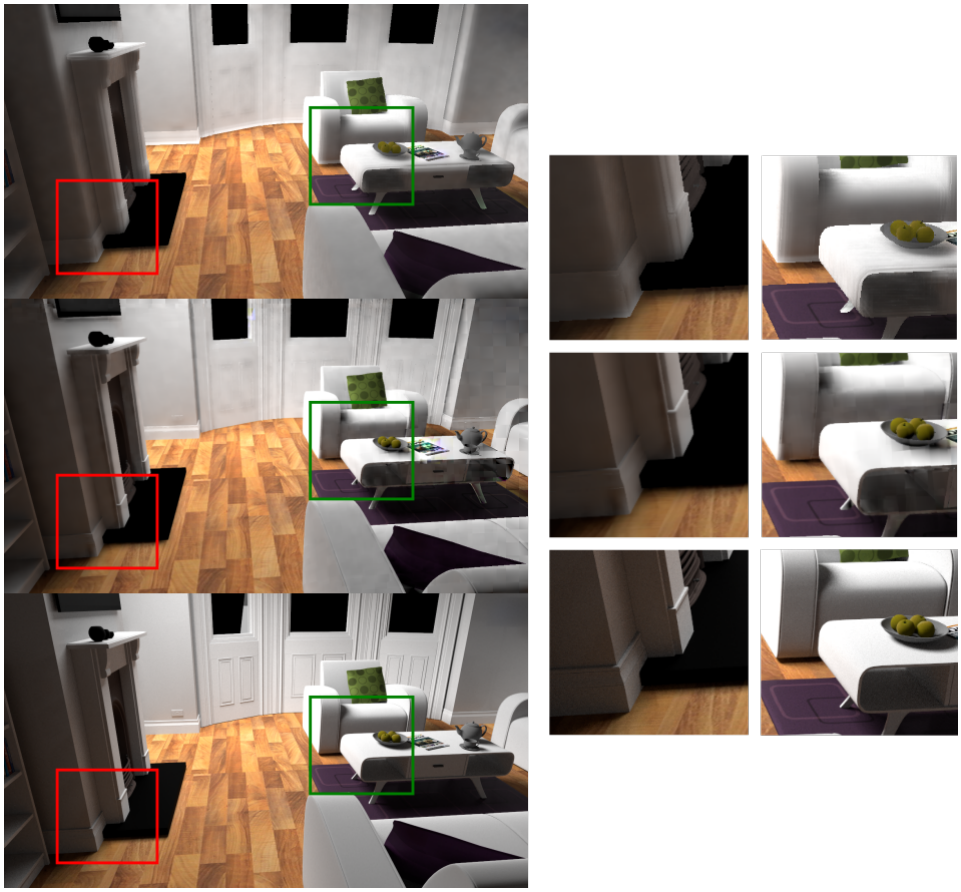


Figure 5.4: *An image from the scene Living Room. From top to bottom: SVGF, BMFR, reference.*

better. Its biggest artifact in this crop-out stems from an uneven shadow, and perhaps a too high emphasis on geometric detail when computing the final illumination.

The second crop shows a more difficult case for both algorithms: The dark region inside the living room table. SVGF again seems to over-blur the shadow, leaving little detail in the reconstruction of the region. BMFR, on the other hand, shows significant blocking artifacts, especially near the edge of the table. In this frame, the end of the table has been in the field of view for 6 frames, the camera moving towards the right, meaning the algorithms have had some time to accumulate samples. Although BMFR seems to reconstruct more details from the object, the artifacts it produces is arguably more distracting than those of SVGF.

The bowl of apples atop the table also poses a challenge, since the object contains small-scale details that are much smaller than the scale of the room. SVGF is not able to reproduce the shadows from on the apples correctly. BMFR seems to handle this situation much better, although the shadow from the bowl on the table shows some slightly blocky artifacts.

In figure 5.5, we see the first image in the Sponza scene image sequence, thus it is taken at a point where neither algorithm has had the time to accumulate samples. While it is unrealistic that a human would perceive artifacts in the first image of a camera sequence at a framerate of 60 fps, it highlights some of the characteristics that were observed in the two other scenes.

The first crop-out shows the edge of the shadow from the arc. In this case, SVGF produces an image that resembles the reference very closely. BMFR, however, shows a significant amount of artifacts, both at the edge of the shadow and its interior. As the frames go by, the new samples will be blended in and smooth out the shadow. This, together with the image from the Living Room scene and the left-most crop-out from the San-Miguel scene, shows that SVGF can perform perceivably better than BMFR in cases where the sample count is low, like in early frames or dark areas.

The second crop-out reiterates another aspect we have already observed. The edge between the two walls is blurred out by the SVGF algorithm. The BMFR algorithm also has difficulties in this scenario, producing a small hint of blocky artifacts on the same edge.

In general, it seems that the BMFR algorithm has a problem with reproducing shadows on smooth surfaces, where the shape of the shadow has no correlation with the shape of the surface. This is a natural consequence

Figure 5.5: *The first image from the scene Sponza. From top to bottom: SVGF, BMFR, Reference.*

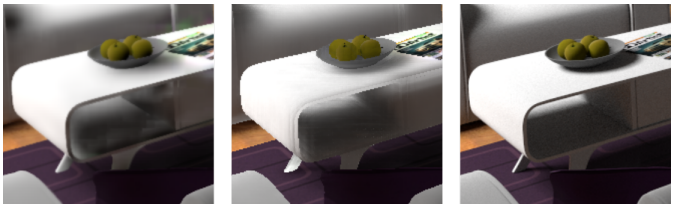


Figure 5.6: *Detail from the last image of the Living Room image sequence. From left to right: BMFR, SVGF and Reference. The table has been within field of view for about 30 frames. BMFR still shows off significant amounts of artifacts, while SVGF has converged at a slightly inaccurate but closer lightsetting.*

of the linear-regression computation that lies in the heart of BMFR: Since the algorithm only tries to approximate the light on each pixel as a linear sum of scene features, it cannot hope to recreate the illumination faithfully when there are no features that correlate with the shape of the shadow. Provided the number of samples is stable over time however, the flaw seems to perish gradually, although often leaving an unsharp shadow.

SVGF seems to struggle more with over-blurring across edges than BMFR in general. This can very well be a consequence of flawed parameter tuning. Nevertheless, it has a clear advantage in settings where the number of samples is restricted, e.g. as shown on the table in the Living Room scene. To further investigate this, a crop-out of just the table is shown in figure 5.6. In this image, the table has been in the field of view for about 30 frames. Still, BMFR has not converged, and shows clearly visible artifacts in the result. Following the above train of thought, this might be attributed to the instability in the number of samples over time.

To summarize the image quality metrics, figure 5.7 shows the VMAF score for the three scenes as a function of frame number. As expected, both algorithms makes a short jump early in the image sequence, as they gather more samples and stabilize on a reconstructed image.

In the graph corresponding to the Living Room scene, SVGF shows a significant drop in VMAF score at frame 17. The sudden drop is highly unexpected, provided no new significant number of artifacts is shown in this frame compared to either of its neighbors. In addition, neither SSIM nor RMSE has a spike at this point. This frame could highlight a limitation to the VMAF algorithm, or rather, a limitation to the use of a video stream evaluation algorithm applied to image denoising. It therefore suggests that one should be careful about relying on VMAF as a ground truth to human

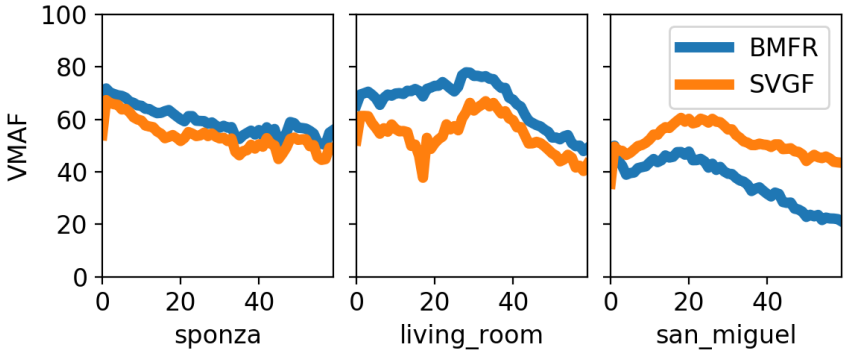


Figure 5.7: *VMAF scores for the three different scenes as a function of frame number.*

perception. Nevertheless, it seems to be the most accurate predictor that is available to this project, and so it will still be used as the main image quality estimator in these experiments, bearing its limitations in mind.

Lastly, BMFR is seen to do consistently better than SVGF on the scenes Sponza and Living Room, but consistently worse on San Miguel. A theory was already established earlier, considering that San Miguel is a dark scene with a general lack of samples, in addition to the fact that the reflections off of specular materials are handled slightly better by SVGF than BMFR. Furthermore, San Miguel contains a high density of foliage and thin structures. The differences are not easy to spot, but figure 5.3 shows a loss of detail in the leaves of the tree, as well as tiny blurs on the chairs in the reconstruction by BMFR. It is clear that BMFR has important weaknesses that should not be overlooked.

5.1.3 Temporal Error

As previously described, temporal stability will be evaluated by running the algorithms on a static scene with a static camera and compare the average luminance difference per pixel. Figure 5.8 shows the behavior of the temporal error metric for the two algorithms, in addition to its behavior on a reference image sequence.

As expected, both algorithms need time in the beginning to stabilize, resulting in a high initial temporal error. After a short while, perhaps surprisingly, both algorithms appear to produce results that are just as stable, or more stable, than the reference. The cause of this could be that both algorithms rely heavily on the samples from the previous frames to produce the next. Additionally, both algorithms take advantage of the neighborhoods of each pixel to smooth out the color. Consequently, the same pixel in consecutive frames is less susceptible to temporal variation in the noisy input.

On the contrary, the reference images are produced entirely independently from each other. In addition, the reference images are produced with a sample count of 4096 per pixel, which may not be enough to stabilize some of the darker regions of the scene.

Furthermore, BMFR outperforms SVGF drastically in this experiment. This is hardly a fair comparison, since SVGF does not contain a temporal anti-aliasing (TAA) stage. It is reasonable to believe that SVGF would outperform the reference and behave similarily to BMFR if TAA was implemented.

As was already mentioned, the authors of BMFR [33] commented that the use of TAA in their algorithm *reduced* the score on the other image

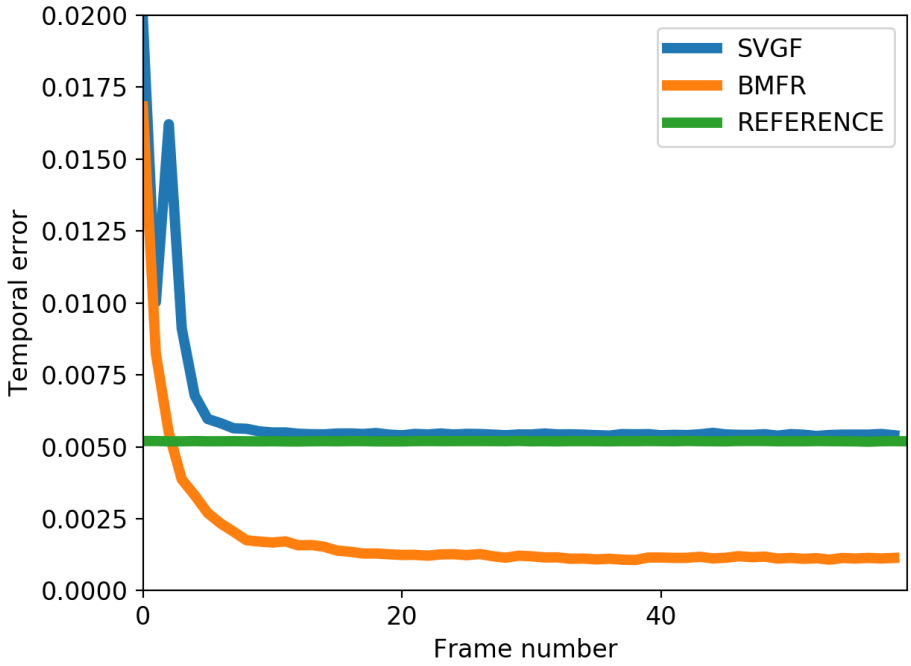


Figure 5.8: *Temporal error scores for the two algorithm and the reference. Lower score is better. The reference image sequence was made with 4096 spp.*

1.0	$n.x$	$n.y$	$n.z$	$n.x^2$
$n.y^2$	$n.z^2$	$\sqrt{\lvert n.x \rvert}$	$\sqrt{\lvert n.y \rvert}$	$\sqrt{\lvert n.z \rvert}$
$n.x^3$	$n.y^3$	$n.z^3$	$w.x$	$w.y$
$w.z$	$w.x^2$	$w.y^2$	$w.z^2$	$\sqrt{\lvert w.x \rvert}$
$\sqrt{\lvert w.y \rvert}$	$\sqrt{\lvert w.z \rvert}$	$w.x^3$	$w.y^3$	$w.z^3$
$w.x \cdot n.x$	$w.y \cdot n.x$	$w.z \cdot n.x$	$w.x \cdot n.y$	$w.y \cdot n.y$
$w.z \cdot n.y$	$w.x \cdot n.z$	$w.y \cdot n.z$	$w.z \cdot n.z$	$w.x \cdot w.y$
$w.y \cdot w.z$	$w.x \cdot w.z$	$w.x - w.y$	$w.y - w.z$	$w.x - w.z$

Table 5.3: *The new feature buffer pool*

quality metrics with dynamic camera, but they decided to include the TAA stage because they found it to give better subjective visual quality.

5.2 Re-Evaluating the Choice of Features in BMFR

The new feature pool from which a new set of features for BMFR will be chosen, is shown in table 5.3. Here, world position is denoted by w and normals by n. Referencing a specific component of a vector is done through a programming-esque dot-notation.

With the features in this new feature pool, the greedy algorithm for feature selection described in section 4.5.2 was run. Table 5.4 shows both the original and the new chosen features, in the order they are chosen by the greedy algorithm. The order of the original features is only shown as a comparison for the interested reader. The order of the original features was determined using a feature pool consisting of these features only.

Figure 5.9 shows the VMAF score of BMFR as each of the ten original feature buffers are added to the mix. The scene used here is Sponza, so that the topmost line is the same as the topmost line in the left graph of figure 5.7. It is immediately apparent that the last few buffers only contribute marginally to the resulting quality. Therefore, shaving off a few of these feature buffers would result in a small quality decrease, but a more significant decrease in computation time.

Figure 5.10 shows VMAF scores with the new choice of feature buffers, using the same image sequence as figure 5.9. On first sight, there is little

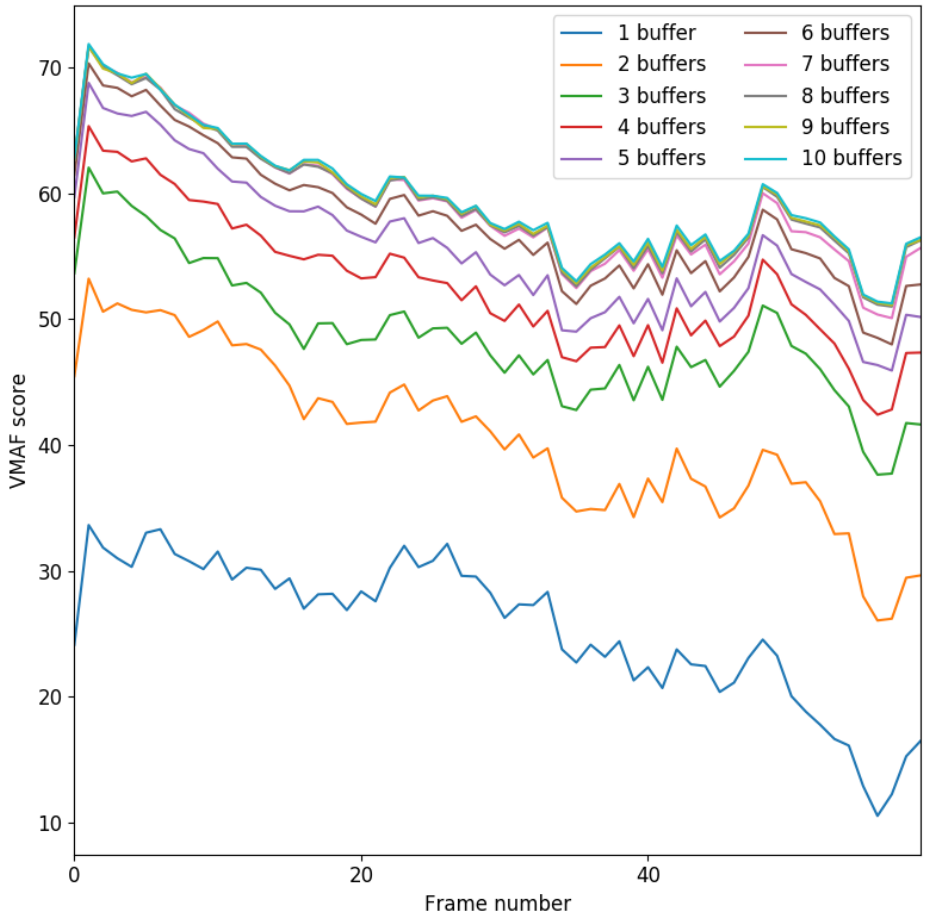


Figure 5.9: *Old choice of feature buffers: Progressive increase in image quality of BMFR output as the number of feature buffers grows, using the feature buffers from the original article. The algorithm is run on the Sponza image sequence.*

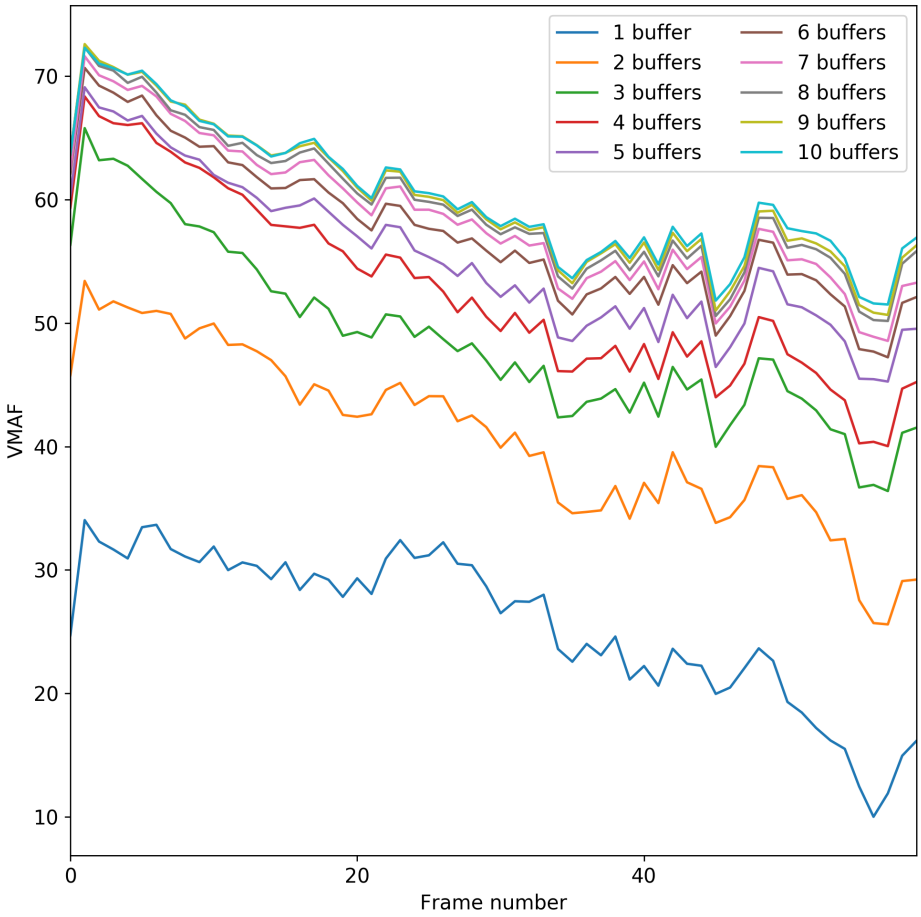


Figure 5.10: *New choice of feature buffers; VMAF score for each frame, for the incremental list of feature buffers*

Original buffers	New buffers
1.0	1.0
$w.x^2$	$w.x^2$
$n.y$	$\sqrt{\lvert n.x \rvert}$
$w.z^2$	$w.z^2$
$n.x$	$w.x \cdot n.y$
$w.y^2$	$w.y^2$
$n.z$	$n.x$
$w.x$	$n.z$
$w.y$	$\sqrt{\lvert n.y \rvert}$
$w.z$	$w.x \cdot w.y$

Table 5.4: *The original and new feature buffers, given in the order they were chosen by the greedy algorithm. The "old buffer pool" from which the original buffers are drawn, consists solely of the buffers shown here.*

Scene	BMFR with Old Features			BMFR with New Features		
	RMSE	SSIM	VMAF	RMSE	SSIM	VMAF
Sponza	0.029	0.889	59.26	0.028	0.890	60.25
Living Room	0.035	0.872	66.51	0.035	0.873	67.37
San Miguel	0.037	0.834	36.30	0.037	0.834	36.45

Table 5.5: *Average scores on one image sequence from each scene, comparing the new and old sets of feature buffers in BMFR.*

evidence of large improvements over the original choice of buffers. In fact, there is little evidence of any changes at all.

Figure 5.11 highlights the situation. Surprisingly, the new, larger feature buffer pool does not strictly outperform the old one. In fact, it seems to give worse results when the number of feature buffers are between 4 and 8. This is unexpected, since the new feature buffer pool contains all the feature buffers in the original selection.

The only explanation of this seemingly paradoxal result, is that the greedy optimization strategy has found a suboptimal solution for 4-8 buffers. The extra features in the feature pool lead to a non-optimal order of buffer selection. Nevertheless, the final score when all ten buffers are added, shows a tiny improvement over that of the original ones.

Table 5.5 shows the RMSE, SSIM and VMAF scores for both the new

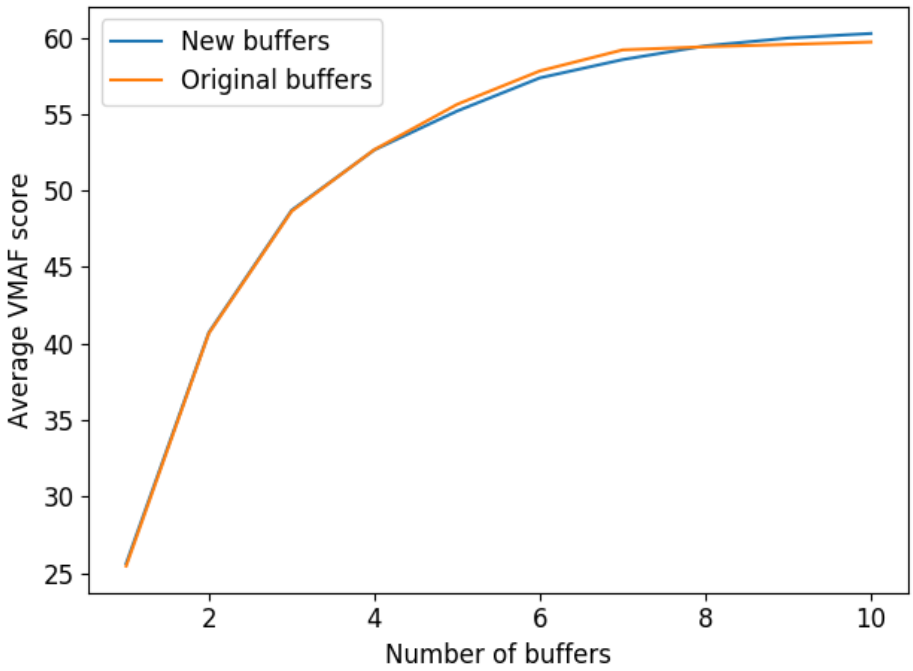


Figure 5.11: *Average VMAF score as more buffers are added, for both original feature buffers and the new feature buffer pool.*

and the old choices of feature buffers.

The results show that the new choice of feature buffers performs just as well or slightly better than the old ones, even on the scenes not used in the choice process. This comes as a surprise, as the chosen features are clearly not agnostic of camera orientation, as the original ones were. The camera directions in the scenes have significant components along both the x- and z-axes, meaning that BMFR with these features – though not indifferent to camera orientation – generalizes surprisingly well to other camera directions than those of the test scene.

However, all of the image sequences treat the positive y-axis as the up-direction of the camera, and neither camera direction has a significant component along the y-axis. Therefore, it would be unwise to put this choice of feature buffers to use before properly investigating whether it is fit for a broader range of camera orientations.

Chapter 6

Conclusions, Limitations and Future Work

To conclude the thesis, this chapter will summarize the key aspects of this work, along with some take-aways based on the research goals of section 1.2.

Chapter 2 gave an introduction to ray tracing and the problem of noise, along with the history of contributions leading up to today's state of the art within both ray tracing techniques and denoising algorithms.

Chapter 3 presented two different state-of-the-art algorithms, namely Spatiotemporal Variance-Guided Filtering and Blockwise Multi-Order Feature Regression. The chapter gave a detailed walk-through of the key steps in each algorithm, giving a deep view into different modern solutions to the denoising problem.

Lastly, chapter 4 described the different experiments that were conducted to compare the two denoising algorithms to one another. It also presented a proposed extension to BMFR and how the proposed modification would be evaluated. Subsequently, chapter 5 presented the results from the conducted experiments and discussing some of the patterns that were observed.

Based on the results from chapter 5, the following section will attempt to draw several conclusions.

6.1 Conclusions

In the first experiment, the performance of SVGF and BMFR were compared to each other both in terms of performance and visual quality. None of the algorithms seems to be undividedly advantageous over the other.

BMFR had a superior performance footprint in terms of both run time and stability, as well as better objective visual quality on the majority of the scenes, but SVGF displays better behavior in areas with few samples and with reflective materials and foilage.

In the subsequent experiment, an extension to BMFR involving new choices of feature buffers was evaluated using a visual quality metric. With the proposed feature buffers, the greedy feature-picking algorithm was unable to choose features to significantly increase the output quality of BMFR. In fact, the greedy algorithm would choose features that proved to be *worse* than the original buffers, for some numbers of features. Therefore, it might be the case that the algorithm is generally unable to benefit from more non-linear functions of the same feature buffers, and that new, independent features may give better results.

As a side-note, a mildly surprising result was that the features chosen based on a single scene and camera path generalized tolerably well to other scenes and camera orientations, even though the features were not orientation agnostic. All in all, this seems to suggest that having varied information about the scene in the feature buffers is more important than that it is homogenous in every coordinate component.

These observations may have implications for the work to improve BMFR through a better selection of feature buffers.

6.2 Limitations

Nothing is perfect, and that goes for the experiments conducted in this thesis as well. This section will point out some of the most prominent limitations of the presented work.

The most glaring issue with the comparison experiments, is that the BMFR implementation was built and fine-tuned by the original authors themselves. The SVGF implementation, although made to resemble a reference provided by the original authors, contains several simplifications as described in section 4.4. Furthermore, even though the algorithm has been tested with different choices of parameters for tuning, it is likely that a set of parameters that performs better in general across scenes could be found.

It was not tested whether having more than ten feature buffers in BMFR would improve quality further. Seeing the convergence rates of figures 5.9 and 5.10, it is tempting to assume the difference would have been insignificant. Although there is room to add more buffers due to BMFR's performance, the number of buffers would be limited by the already-expensive fitter stage's quadratic dependency on this number.

The number of scenes and the variety of materials and light settings has also been limited. Each scene has only been equipped with a single rectangular light source. A scene for real use would be expected to have many and more general light sources. Each scene was also investigated with only a single camera path, which further limits the variation in the data.

6.3 Future Work

It is apparent that some issues need to be fixed before path tracing can fully replace the many years of work and experience that lies behind traditional real-time rendering techniques. Real-time denoisers have come a long way and are able to reconstruct realistic-looking scenery, even with the very unreliable base of just one sample per pixel.

The two different algorithms that were examined here have been shown to display different strengths and weaknesses. Future algorithms should do their best to unite the strengths of each of the two approaches while adressing their weaknesses.

One thing is certain, however. Practical real-time path tracing is closer to becoming a reality than ever before.

Appendix

Source Code

The unified data construction repository can be found at `https://github.com/TheVaffel/denoiser_data_constructor`.

It includes the modified repositories for production of ray traced images and rasterized feature buffers as *Git submodules*. They can also be accessed directly through `https://github.com/TheVaffel/ChameleonRT` and `https://github.com/TheVaffel/Vulkan-glTF-PBR` respectively.

The self-made implementation of SVGF can be found at `https://github.com/TheVaffel/spatiotemporal-variance-guided-filtering`.

The modified implementation of BMFR can be found at `https://github.com/TheVaffel/bmfr`.

Bibliography

[1] D3D Team, Microsoft . Announcing microsoft directx raytracing! https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/, 2018. Accessed 2020-06-10.

[2] Martindale, Jon. Here are all the games that support nvidia's rtx ray tracing. https://www.digitaltrends.com/computing/games-support-nvidia-ray-tracing/, 2020. Accessed 2020-04-23.

[3] Attila T. Áfra, Ingo Wald, Carsten Benthin, and Sven Woop. Embree ray tracing kernels: Overview and new features. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, pages 52:1–52:2, New York, NY, USA, 2016. ACM.

[4] AMD. Amd details strategy to deliver best-in-class growth and strong shareholder returns at 2020 financial analyst day. https://www.amd.com/en/press-releases/2020-03-05-amd-details-strategy-to-deliver-best-class\protect\discretionary{\char\hyphenchar\font}{}{}-growth-and-strong-shareholder, 2020. Accessed 2020-06-10.

[5] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 37–45, New York, NY, USA, 1968. Association for Computing Machinery.

[6] Sig Badt. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4(3):123–132, May 1988.

[7] F O Bartell, E. L. Dereniak, and W. L Wolfe. The Theory And Measurement Of Bidirectional Reflectance Distribution Function (Brdf) And Bidirectional Transmittance Distribution Function (BTDF). In Gary H. Hunt, editor, *Radiation Scattering in Optical Systems*, volume

0257, pages 154 – 160. International Society for Optics and Photonics, SPIE, 1981.

[8] Pablo Bauszat, Martin Eisemann, and Marcus Magnor. Guided image filtering for interactive high-quality global illumination. *Computer Graphics Forum*, 30(4):1361–1368, 2011.

[9] Benedikt Bitterli, Fabrice Rousselle, Bochang Moon, José A. Iglesias-Guitián, David Adler, Kenny Mitchell, Wojciech Jarosz, and Jan Novák. Nonlinearly weighted first-order regression for denoising Monte Carlo renderings. *Computer Graphics Forum (Proceedings of EGSR)*, 35(4):107–117, June 2016.

[10] A. Buades, B. Coll, and J. . Morel. A non-local algorithm for image denoising. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 60–65 vol. 2, 2005.

[11] Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. The design and evolution of disney's hyperion renderer. *ACM Trans. Graph.*, 37(3), July 2018.

[12] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4), July 2017.

[13] Kenneth Chiu, Changyaw Wang, and Peter Shirley. V.4. - multi-jittered sampling. In Paul S. Heckbert, editor, *Graphics Gems*, pages 370 – 374. Academic Press, 1994.

[14] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali. Ray tracing for the movie 'cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006.

[15] Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Trans. Graph.*, 37(3), August 2018.

[16] Per H. Christensen and Wojciech Jarosz. The path to path-traced movies. *Foundations and Trends in Computer Graphics and Vision*, 10(2):103–175, October 2016.

[17] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, January 1986.

[18] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, August 1987.

[19] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[20] Intel Corporation. Intel® ospray. `https://www.ospray.org/`, 2019. Accessed 2020-01-20.

[21] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch. Edge-avoiding À-trous wavelet transform for fast global illumination filtering. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, page 67–75, Goslar, DEU, 2010. Eurographics Association.

[22] Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François X. Sillion. A frequency analysis of light transport, 2005.

[23] Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. *Computer Graphics Forum*, 32(4):125–132, 2013.

[24] Luca Fascione, Johannes Hanika, Marcos Fajardo, Per Christensen, Brent Burley, and Brian Green. Path tracing in production - part 1: Production renderers. In *ACM SIGGRAPH 2017 Courses*, SIGGRAPH '17, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Academy Software Foundation. OpenEXR, 2020. https://www.openexr.com/.

[26] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. Exploring the use of ray tracing for future games. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, page 41–50, New York, NY, USA, 2006. Association for Computing Machinery.

[27] Alain Galvan. Ray tracing denoising. `https://alain.xyz/blog/raytracing-denoising`, 2020. Accessed 2020-01-10.

[28] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, page 213–222, New York, NY, USA, 1984. Association for Computing Machinery.

[29] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, December 1964.

[30] Intel. Intel open image denoise. `https://openimagedenoise.github.io/`, 2020.

[31] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[32] Daniel Koch, Tobias Hector, Joshua Barczak, and Eric Werness. Ray tracing in vulkan. `https://www.khronos.org/blog/ray-tracing-in-vulkan`, 2020. Accessed 2020-06-10.

[33] Matias Koskela, Kalle Immonen, Markku Mäkitalo, Alessandro Foi, Timo Viitanen, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. Blockwise multi-order feature regression for real-time path tracing reconstruction. *ACM Transactions on Graphics (TOG)*, 38(5), June 2019.

[34] Eric Lafortune and Yves Willems. Bi-directional path tracing. *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics'*, 93, 01 1998.

[35] Christian Lauterbach, Sung eui Yoon, and Dinesh Manocha. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45, 2006.

[36] Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. An efficient denoising algorithm for global illumination. In *Proceedings of High Performance Graphics*, New York, NY, USA, July 2017. ACM.

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

[38] Don P. Mitchell. Generating antialiased images at low sampling densities. *SIGGRAPH Comput. Graph.*, 21(4):65–72, August 1987.

[39] NVIDIA. Nvidia turing gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, 2018. Accessed 2020-06-10.

[40] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[41] Matt Pharr and Pat Hanrahan. Geometry caching for ray-tracing displacement maps. In *In Eurographics Rendering Workshop*, pages 31–40. Springer, 1996.

[42] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 101–108, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[43] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum*, 26(3):415–424, 2007.

[44] Amy Reibman and David Poole. Predicting packet-loss visibility using scene characteristics. pages 308 – 317, 12 2007.

[45] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[46] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery.

[47] I.M Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86 – 112, 1967.

[48] Will Usher. ChameleonRT, 2020. `https://github.com/Twinklebear/ChameleonRT`.

[49] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. *Proceedings of Eurographics Workshop on Rendering*, 01 1995.

[50] D. Vicini, D. Adler, J. Novák, F. Rousselle, and B. Burley. Denoising deep monte carlo renderings. *Computer Graphics Forum*, 38(1):316–327, 2019.

[51] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6–es, January 2007.

[52] Eric Werness, Ashwin Lele, Robert Stepinski, Nuno Subtil, and Cristoph Kubisch. VK_NV_ray_tracing. `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VK_NV_ray_tracing`, 2018. Accessed 2020-06-10.

[53] Turner Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, page 14, New York, NY, USA, 1979. Association for Computing Machinery.

[54] Sascha Willems. Vulkan-glTF-PBR, 2020. https://github.com/SaschaWillems/Vulkan-glTF-PBR.

[55] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. Toward a practical perceptual video quality metric. `https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652`, 2016. Accessed 2020-01-20.

[56] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.

[57] M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, P. Sen, C. Soler, and S.-E. Yoon. Recent advances in adaptive sampling and reconstruction for monte carlo rendering. *Computer Graphics Forum*, 34(2):667–681, 2015.