TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

KALLE IMMONEN

REAL-TIME NOISE REMOVAL IN FOVEATED PATH TRACING

Master of Science thesis

Examiners:
D.Sc. Pekka Jääskeläinen
M.Sc. Matias Koskela

The examiners and topic of the thesis
were approved on 1st March 2017.

ABSTRACT

KALLE IMMONEN: Real-Time Noise Removal in Foveated Path Tracing
Tampere University of Technology
Master of Science thesis, 51 pages
June 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiners: D.Sc. Pekka Jääskeläinen, M.Sc. Matias Koskela
Keywords: foveated rendering, path tracing, denoising, ray tracing

Path tracing is a method for rendering photorealistic two-dimensional images of three-dimensional scenes based on computing intersection between the scene geometry and light rays traveling through the scene. The rise in parallel computation resources in devices such as graphics processing units (GPUs) have made it more and more viable to do path tracing in real time. To achieve real-time performance, path tracing can be further optimized by using foveated rendering, where the properties of the human visual system are exploited to reduce the number of rays outside the central point of vision (fovea), where the human eye cannot discern fine detail.

The reduction in the number of rays can, however, lead to several issues. Noise appears in the image as a result of an inadequate number of path tracing samples allocated to each pixel. Furthermore, the variation in the noise from one animation frame to the next appears as flicker. Finally, artifacts can appear when the spatially subsampled image is upsampled to a uniform resolution for display.

In this thesis, solutions to the aforementioned issues are explored by implementing three noise removal methods into a foveated path tracing rendering system. The computational performance and the visual quality of the implemented methods is evaluated. Of the implemented methods, cross-bilateral filter provides the best quality, but its runtime doesn't scale well to large filter sizes. Large filter sizes are enabled by the À-Trous approximation of the cross-bilateral filter, at the cost of generating more artifacts in the result. Overall, while the implemented methods are able to provide visually pleasing results in some scenarios, improvements in the algorithms (e.g., local filter parameter selection) are needed to reach the quality seen in offline methods.

TIIVISTELMÄ

KALLE IMMONEN: Reaaliaikainen kohinanpoisto katsekeskeisessä polunjäljityksessä
Tampereen teknillinen yliopisto
Diplomityö, 51 sivua
Kesäkuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: TkT Pekka Jääskeläinen, DI Matias Koskela
Avainsanat: katsekeskeinen piirto, polunjäljitys, kohinanpoisto, säteenjäljitys

Polunjäljitys on menetelmä fotorealististen kaksiulotteisten kuvien piirtämiseksi kolmiulotteisen maailman kuvauksesta. Menetelmä perustuu maailman geometrian ja valonsäteiden välisten leikkauspisteiden laskemiseen. Näytönohjainten ja muiden laitteiden rinnakkaisten laskentaresurssien kasvun myötä myös reaaliaikainen polunjäljitys on muuttunut entistä mahdollisemmaksi. Reaaliaikasuorituskyvyn saavuttamiseksi polunjäljitystä voidaan tehostaa entisestään käyttämällä katsekeskeistä piirtoa, jossa ihmisen näköjärjestelmän ominaisuuksia käytetään hyväksi säteiden määrän vähentämiseksi tarkan näön pisteen ulkopuolella, missä ihmissilmä ei havaitse yksityiskohtia tarkasti.

Säteiden määrän vähentäminen voi kuitenkin aiheuttaa useita ongelmia. Riittämätön määrä näytteitä pikseliä kohden polunjäljityksessä tuottaa kuvaan kohinaa. Lisäksi kohinan ajallinen vaihtelu aiheuttaa välkkymistä animaatioissa. Kuvaan voi myös syntyä häiriöitä, kun tilallisesti alinäytteistetty kuva skaalataan täyteen resoluutioon, jossa kuvaelementit ovat tasaisesti jakautuneita.

Tässä diplomityössä tutkitaan ratkaisuja edellä mainittuihin ongelmiin toteuttamalla kolme kohinanpoistomenetelmää katsekeskeiseen polunjäljitykseen perustuvaan piirtojärjestelmään. Työssä arvioidaan menetelmien laskennallista suorituskykyä ja tuotetun kuvan laatua. Toteutetuista menetelmistä *cross-bilateral-suodin* tuottaa parhaan kuvanlaadun, mutta sen ajoaika skaalautuu huonosti suurilla suodinko'oilla käytettäväksi. *Cross-bilateral-suotimen* niin sanottu *À-Trous-approksimaatio* mahdollistaa suurempien suodinkokojen käyttämisen, mutta tuottaa kuvaan enemmän häiriöitä. Kaiken kaikkiaan, vaikka toteutetut menetelmät kykenevätkin tuottamaan visuaalisesti miellyttäviä tuloksia tietyissä tilanteissa, suodatusmenetelmiä tulisi parantaa (esimerkiksi valitsemalla suotimen parametrit paikallisesti) ei-reaaliaikaisten suodatusmenetelmien kaltaisen laadun saavuttamiseksi.

PREFACE

This thesis was done as part of the research of the Virtual reality and Graphics Architectures (VGA) group at Tampere University of Technology (TUT). Most of the work presented in this thesis was done by the author. The foveated rendering system that was used as the basis of this work was largely the work of Matias Koskela. That includes research on sampling patterns, which are briefly discussed in the thesis.

I would like to thank the supervisors of this thesis, Matias Koskela and Pekka Jääskeläinen, for providing help and guidance during the process, and for suggesting this interesting topic. In addition, thanks to Markku Mäkitalo and Timo Viitanen for providing helpful comments during the late stages of writing.

In Tampere, Finland, on May 24, 2017

Kalle Immonen

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

2D	Two-Dimensional
3D	Three-Dimensional
BLS–GSM	Bayes Least Squares–Gaussian Scale Mixtures
BM3D	Block-Matching and 3D Filtering
BRDF	Bidirectional Reflectance Distribution Function
BSDF	Bidirectional Scattering Distribution Function
BTDF	Bidirectional Transmittance Distribution Function
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
FOV	Field of View
fps	Frames Per Second
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
HDC	Hierarchical Discrete Correlation
HMD	Head-Mounted Display
LTE	Light Transport Equation
MAD	Median Absolute Deviation
MSE	Mean Squared Error
PDF	Probability Distribution Function
RGB	Red Green Blue
SAT	Summed-Area Table
spp	Samples Per Pixel
SURE	Stein's Unbiased Risk Estimator
SVD	Singular Value Decomposition
VAF	Visual Acuity Function
VR	Virtual Reality

1. INTRODUCTION

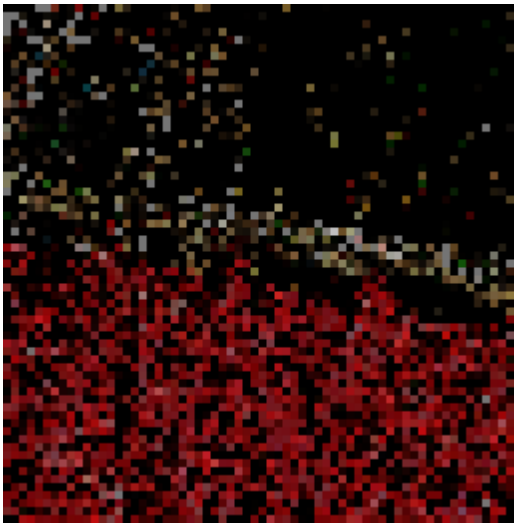
Path tracing is a method for rendering photorealistic two-dimensional (2D) images of three-dimensional (3D) scenes based on computing intersections between the scene geometry and light rays traveling through the scene, in effect mimicking the physical behavior of photons. Because path tracing is based on solving a complex multi-dimensional integral with Monte Carlo methods, it has historically been too computationally expensive to be performed in real time. However, path tracing is a highly parallelizable method, and benefits greatly from the rise of parallel computation resources in devices such as graphics processing units (GPUs) during recent years. Thus, it is becoming more and more viable to do path tracing in real time.

Path tracing can be further optimized by using foveated rendering, where the properties of the human visual system are exploited to reduce the number of rays—and thus the amount of processing—outside the central point of vision, *fovea*, in the field of view. Foveation is especially suited for head-mounted displays (HMDs) used in virtual reality (VR) applications due to their large field of view (FOV) and the conversely smaller foveal area. Foveation is made possible by the HMD having only one user, which makes it possible to integrate eye tracking within the HMD. The user's immersion in VR can also be heightened by the realistic rendering results provided by path tracing.

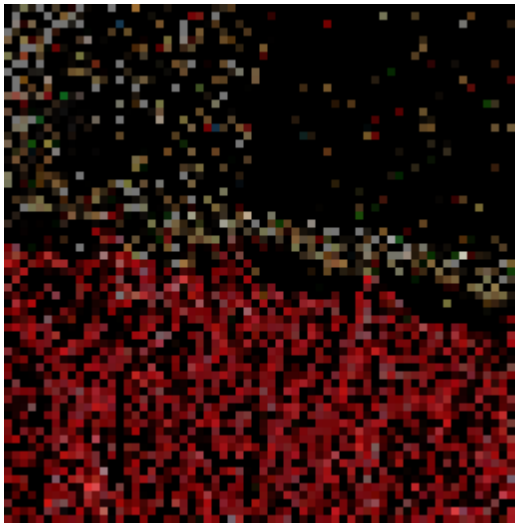
The reduction in the number of rays in foveation can, however, lead to several issues: (1) Visually jarring Monte Carlo noise caused by an inadequate number of path tracing samples allocated to a pixel. This noise is illustrated in Figure 1.1(c) and Figure 1.1(d), where 64×64 pixel close-ups of a path traced image at 16 samples per pixel (spp) are shown. (2) Temporal variation in the noise patterns, causing visible flicker. Subfigures (c) and (d) in Figure 1.1 are two frames of an animation from the same location. Variation of the error in separate frames causes flicker. (3) Artifacts caused by filling in the missing samples resulting from spatial subsampling in foveated rendering. Spatial subsampling is depicted in Figure 1.1(a). Figure 1.1(b) is the corresponding filled image with visible artifacts.

(a) Subsampled

(b) Filled



(c) Monte Carlo noise (16 spp), frame 1



(d) Monte Carlo noise (16 spp), frame 2

Figure 1.1 *Problems of foveated path tracing. Spatial subsampling (a) causes artifacts when upsampled to a uniform resolution (b). A low number of Monte Carlo samples per pixel causes noise in the image (c, d). The variation in noise patterns causes visible flicker in animations, as depicted by the differences seen in subfigures (c) and (d). Note that the subsampled area in (a) may appear black unless properly zoomed in.*

In this thesis, solutions to the aforementioned issues will be explored. The theoretical background involved in removal of Monte Carlo noise (henceforth, "denoising") will be covered, ending in a review of some of the state-of-the-art methods. Note that Monte Carlo denoising in real time has not been studied extensively in the past, especially in the context of foveated rendering: most existing filtering methods assume an offline rendering setting. For this thesis, denoising methods were implemented into an existing path tracing renderer, with the goal of finding out the viability of the methods for denoising in a real-time setting. The computational and visual performance of the implemented methods is evaluated.

This thesis is organized as follows. Chapter 2 discusses real-time rendering methods. First, the concept of foveated rendering is covered, including properties of the human visual system that enable foveated rendering to be used. Next, the most common rendering methods are discussed: rasterization and ray tracing. The principles of an advanced form of ray tracing called path tracing are described. In Chapter 3, different types of general-purpose image-space filters are described. Chapter 4 covers some of the state-of-the-art methods specifically designed for Monte Carlo noise removal. Chapter 5 describes the foveated rendering and denoising system that was implemented. The performance and visual quality of different methods are assessed. Final conclusions are given in Chapter 6.

2. REAL-TIME RENDERING

In *real-time rendering*, images of a mathematically defined 3D scene are drawn by a computer in a rapid succession, providing a sense of immersion to the user. Typical applications of real-time rendering are ones where the user is able to influence the scene in some way, e.g., by moving the camera or a character in the simulated world.

A sufficient frame rate is required to provide immersion to the user. A sense of interactivity starts to grow at around 6 fps (frames per second), and at 72 fps and higher the differences in frame rate start to be undetectable [AMHH08, p. 1]. In VR applications the requirement is even higher—around 95 fps is required because of low persistence needed in VR displays [Abr14]. The render time budget is defined by the inverse of frame rate. For example, to achieve a 95 fps frame rate, the system must be able to render every frame in less than about 10.5 ms (milliseconds). Real-time rendering can be contrasted with *offline rendering*, where the rendering may take hours, or even days.

The display resolution is also an important factor in providing immersion. In current consumer hardware for VR, a typical resolution is 1080×1200 pixels (≈ 1.3 Mpixels) per eye [IGN16]. In this thesis, an approximative figure of 1 Mpixels is used when comparing runtimes of different filtering methods, although note that this only accounts for one eye.

The high frame rate and resolution requirements mean that the rendering system must be highly performant. The rendering performance can be optimized by employing *foveated rendering*. The concept of foveated rendering, its issues, and properties of the human visual system relevant to the subject are presented in the next section.

In Section 2.2 and Section 2.3, two methods for rendering 3D scenes are introduced: rasterization and ray tracing. Rasterization is discussed at a basic level, highlighting some of its properties and differences compared to ray tracing with respect to foveated rendering. More emphasis is given to ray tracing, where the path tracing method of rendering is discussed in detail. Monte Carlo approximation methods used in path tracing are also covered.

Finally, in Section 2.5, the auxiliary feature buffers that are typically available from a renderer with little to no additional cost are briefly discussed. The feature buffers can be used to aid in filtering out noise from the image while preserving edges that are represented well in the feature buffers.

2.1 Foveated Rendering

In foveated rendering, certain properties of the human visual system are exploited in order to lessen the computational load required for rendering. More precisely, the fact that human's visual acuity is much higher in the foveal region can be used to reduce the amount of computations outside the fovea. There are various options as to how foveated rendering can be implemented, and the techniques also depend on the properties of the rendering system in use.

The main issue in foveated rendering is the need for spatial subsampling. In other words, more samples should be taken close to the fovea than outside it. When the sparsely sampled contents are upsampled (i.e., the missing samples are interpolated) for uniform displays, upsampling artifacts can appear. Thus, some kind of a filter is required to reduce these artifacts. Additionally, if the samples are themselves noisy (as is the case in path tracing), the noise is spread to the interpolated samples, which accentuates the artifacts further.

Furthermore, it should be noted that the goal—in an optimal scenario—is not to maximize the quality of the filtered result. Instead, a compromise should be found between the computational resources spent on rendering and filtering, such that the execution time is minimized, while producing a result that is indistinguishable (with foveation) from a result without foveation. Because human's visual acuity in the parafoveal and perifoveal regions is not as high as within the fovea, excessive quality outside the foveal region would go to waste.

Since the human visual system is at the core of foveated rendering, its most important properties with regard to foveated rendering are discussed next.

Human visual system. A typical human visual field of view (FOV) spans 135° vertically and 160° horizontally. However, fine detail is only sensed in the fovea, which covers merely about 5° of the FOV. [GFD+12] The fovea thus covers less than 1% of area within the field of view.

The difference in acuity is caused by variation in the number of rod, cone and ganglion cells in the retina. In the fovea, the density of the color-sensing cone cells

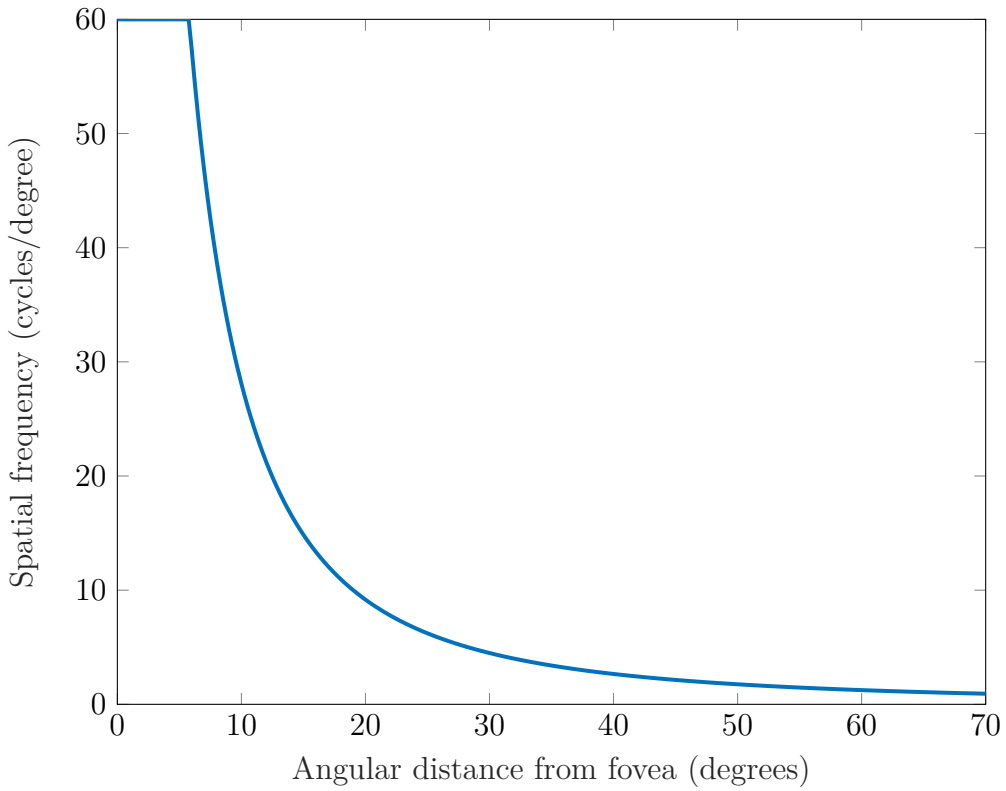
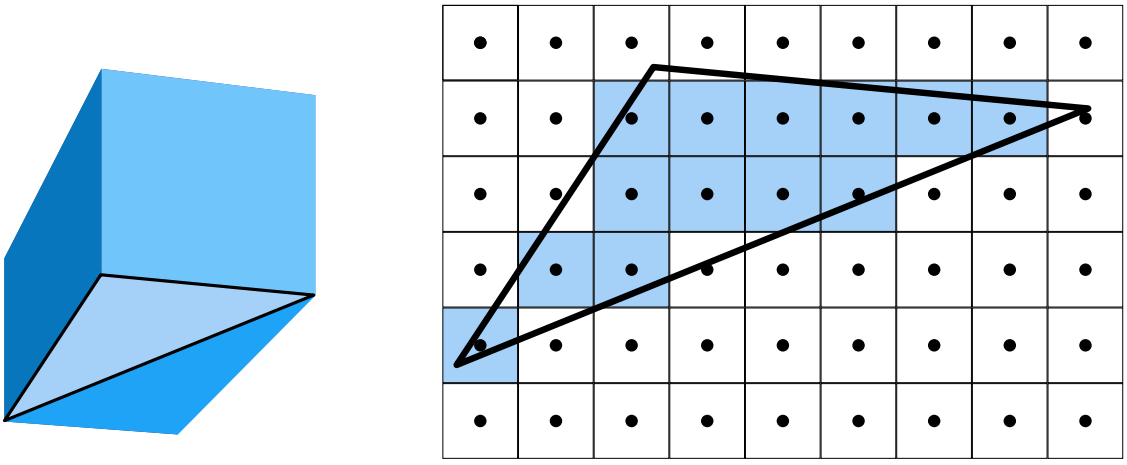


Figure 2.1 *Visual acuity function. Human eye is able to resolve a maximum of 60 cycles/degree of detail when the angular distance from the point of gaze is less than about 6 degrees. As the angular distance grows, the ability to sense detail decreases rapidly. Blind spot is omitted from this graph. Adapted from Reddy [Red01].*

matches the density of ganglion cells, causing little to no loss of spatial information. Outside the fovea multiple cone cells share the same ganglion cells, which makes the ganglion cells act as filters. [PSK+16]

Visual acuity function (VAF) measures how much detail the human eye is able to resolve relative to the angular distance from the central point of gaze [Red01]. VAF, graphed in Figure 2.1, shows that the ability to sense detail drops off rapidly as the angular distance from the point of gaze increases beyond roughly 6 degrees.

Some visual tasks perform uniformly throughout the visual field, while others are affected by the eccentricity. For example, flicker sensitivity and the ability to detect objects' velocities is roughly uniform across the visual field [PSK+16]. Thus, foveated rendering systems should attempt to reduce rendering quality in the periphery, while at the same time avoiding temporal aliasing. The ability to perceive colors also diminishes as the distance to fovea grows [PSK+16], however it's difficult to take advantage of this fact in a rendering system.



(a) (b)

Figure 2.2 *Rasterization. A primitive (e.g., a triangle) of a 3D model is transformed from 3D coordinate space to 2D screen space (a). The pixels whose center points fall under the area covered by the primitive are filled in (b). This process is repeated for all primitives within all models in the 3D scene.*

2.2 Rasterization

Rasterization is the primary method used today for rendering 3D scenes in a real-time setting. Due to its ubiquitous support in modern GPUs, it is utilized in many applications such as 3D modeling software, games, and virtual reality applications. While rasterization is not directly related to the main topic of this thesis, it is briefly discussed here in order to provide a point of comparison against ray tracing and path tracing, which will be covered in more depth later.

Rasterization (also called *scan conversion*) is based on rendering primitives, such as triangles and quadrilaterals, in a straightforward manner by calculating the location of the primitive on screen, and by filling in its pixels. The primitive's on-screen location can be calculated by transforming each 3D vertex of the primitive by a suitable transformation matrix. [AMHH08, p. 20–21] A group of primitives that are connected is called a 3D model. The rasterization process is illustrated in Figure 2.2.

Because primitives are projected onto a 2D plane before they are drawn, rasterization is in essence a 2D operation. The process of filling in the primitives is repeated for all primitives that the 3D scene is composed of, producing a 2D image of the scene. The colors of each pixel of a primitive are calculated in a separate step called *shading* [AMHH08, p. 22]. While rasterization is done in 2D, some parts of shading—e.g., lighting calculations and perspective-correct texture mapping—may need to take 3D information into account.

Current graphics hardware cannot vary the display resolution locally, however,

software-based methods have been proposed for implementing foveated rendering with rasterization. Guenter et al. [GFD+12] use a method where the same scene is rendered 3 times with different resolutions and a varying viewport size. The innermost layer is the smallest, and has the highest amount of detail, while the outermost layer covers the full image, but has a lower resolution. These intermediate layers are composed together to form the final, foveated image. Blend masks are used to blend between the layers to avoid abrupt changes when transitioning from one layer to another. This approach to foveated rendering improves graphics performance by a factor of 5–6, according to the authors.

There are a few downsides to rasterization. First, rasterization makes it difficult to calculate global illumination—where all surfaces of the scene should be able to affect the colors of other surfaces—and other phenomena of light and optics such as reflections, refractions, shadows, and depth of field. In a real-time setting these effects are often approximated or precalculated, which can affect the quality of the rendering result adversely. Secondly, rasterization cannot easily support arbitrary subsampling patterns in screen space, partially because of limitations of current rasterization hardware. Thus, methods such as the one proposed by Guenter at al. [GFD+12] have to be employed to take advantage of foveated rendering. Alternative rendering methods, e.g., ray tracing, have been developed to overcome problems of rasterization.

2.3 Ray Tracing

Ray tracing is a rendering method based on finding intersections between mathematically defined rays of light and the scene geometry [AMHH08, p. 412]. Whereas in rasterization the process is based on looping through primitives and filling in their respective pixels, in ray tracing *primary rays* corresponding to each screen pixel are used to find the primitive that has the closest intersection distance with the ray. The direction and origin of the primary rays are defined by the orientation, position and the type of the camera. This idea is based on light rays traveling from the light source through the scene, and finally reaching the viewer's eye. For practical purposes, however, the process is reversed by tracing rays starting from the eye, as seen in Figure 2.3.

The primary ray—shot from the camera—is only sufficient to find the last point of impact within the light's path before the light reached the camera (eye). To actually shade the pixel, i.e., to calculate its color, various different methods can be used. Methods similar to those used in rasterization-based graphics are possible, however the primary motivation for using ray tracing would be lost, namely more accurate

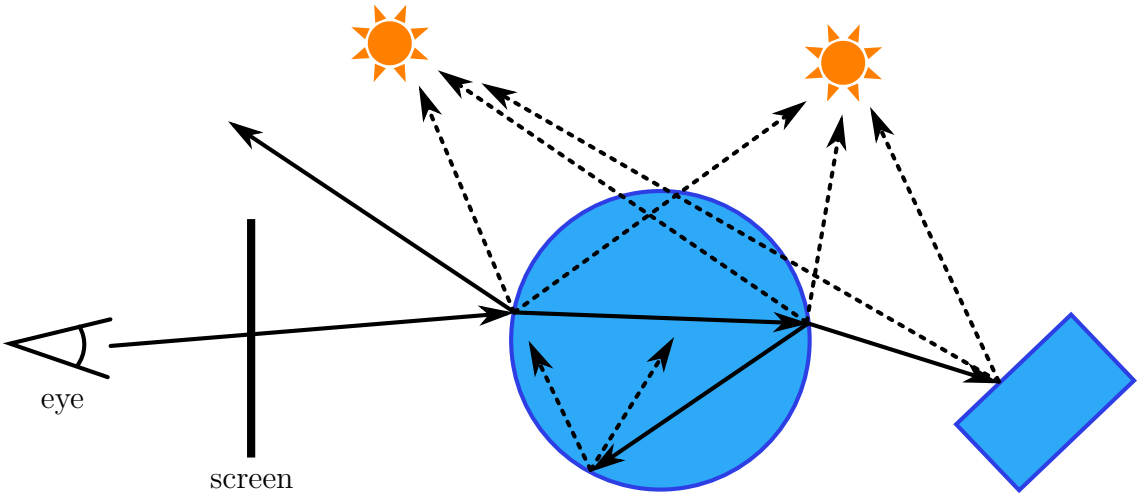


Figure 2.3 *Raytracing. The primary ray is shot from the eye (camera) through the screen plane into the scene. As a ray hits the surface of a primitive (in this case, first, a sphere, and later, a cuboid), new rays are generated recursively to form paths of rays. Dashed lines indicate shadow rays, which are shot towards light sources to find out whether a point is in shadow. Adapted from Akenine-Möller et al. [AMHH08, p. 413].*

simulation of light's behavior. Instead, shading is usually achieved by shooting more rays from the impact point recursively until the final ray reaches a light source. This way, the rays form a path from the eye to the light source. Many different ways are possible for choosing the direction for these new rays.

Often an acceleration structure is used to accelerate the process of finding the primitives before accurate intersections are calculated [PH10, p. 183–184]. One example of an acceleration structure is the bounding volume hierarchy (BVH) [PH10, p. 208]. An acceleration structure can reduce the workload for finding the intersecting primitive from $O(N)$ for a naïve implementation to $O(\log N)$ on average for an accelerated implementation [AMHH08, p. 415], where N is the number of objects (primitives) in the scene. Other factors to take into account are the speed of constructing the acceleration structure, how easy it is to make modifications to the structure once it has been built, its memory requirements, and speed of traversal. The factors are not independent. For example, a structure can be compressed to save memory at the expense of lowering the speed of traversal.

2.4 Path Tracing

Path tracing is a form of ray tracing where full paths of interconnected light rays are traced from light sources to the camera by randomizing the direction of the rays at surface boundaries. In the following sections the *light transport equation*—

which is at the basis of all physically based rendering—will be described. Because light transport equation contains a complex multi-dimensional integral, it cannot be solved analytically. The *Monte Carlo method*, discussed in Section 2.4.2, is a numerical method which can be used to approximate arbitrary integrals, including the one contained in light transport equation.

2.4.1 Light Transport Equation

Light transport equation (LTE) is an integral equation that provides the basis for path tracing, presented separately by Kajiya [Kaj86] (as the *rendering equation*) and Immel et al. [ICG86] in 1986. In Kajiya's original form the equation defines the light energy transmitted between two surface points as a sum of the light emitted from the surface and the light intensity arriving from other surface points in the scene (this is called the *surface form* of the equation). In an alternate form, using a notation adapted from Pharr and Humphreys [PH10, p. 752], the radiance leaving from a surface point in a specific direction can be specified as

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\mathcal{S}^2} f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos(\theta_i)|\, d\omega_i, \tag{2.1}$$

where

p	is the surface point,
L_e	is the light emitted from the surface point,
$\mathcal{S}^2$	is a sphere of directions centered at the surface point,
f	is the bidirectional scattering distribution function (BSDF) of the surface,
ω_i	is the incoming light direction,
ω_o	is the outgoing light direction,
L_i, L_o	are the incident and exitant radiance, respectively, and
θ_i	is the angle between the incoming light direction and the surface normal.

The right side of Equation 2.1 consists of two parts: the intensity of the *emitted* light (L_e) and the *scattered* light (the integral). The emitted amount depends only on the outgoing light direction. The latter term is an integral over a sphere that is centered at the point of interest. Thus, the integral covers all incident light directions. Note that in some forms of the equation a hemisphere oriented to the surface normal is used instead, in which case the equation doesn't account for transmitted light and the integral term can be referred to as the *reflection equation*. The principles of light transport equation are illustrated in Figure 2.4.

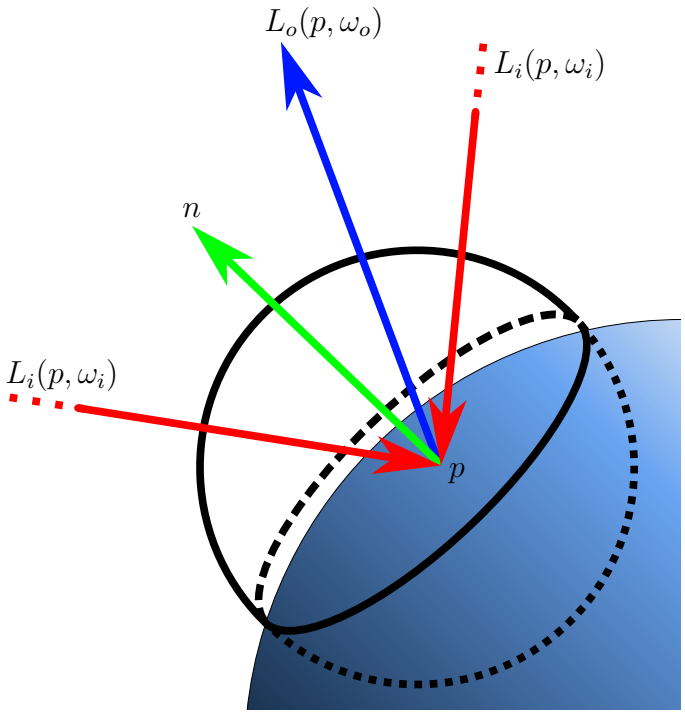


Figure 2.4 *Light transport equation. The exitant radiance L_o (blue) is the integral of incoming radiance L_i (red) from all incoming directions ω_i over a sphere, scaled by the BSDF of the surface material and a cosine term (Equation 2.1). When only reflection is considered, the integral is taken over the upper hemisphere of the sphere, oriented in the direction of the surface normal n (green).*

Within the integral, the bidirectional scattering distribution function (BSDF) term f determines how light is scattered by the surface given a specific incoming and outgoing direction. This way, BSDF allows different surface properties to be modeled. BSDF is a combination of the bidirectional reflectance distribution function (BRDF) and the bidirectional transmittance distribution function (BTDF). The intensity of incident light is scaled by the BSDF term and a cosine term that depends on the angle between the incoming light direction and the surface normal.

In principle, Equation 2.1 allows us to render a 2D image of a 3D scene by calculating the intensity at a surface point seen by the camera. However, a general closed-form solution for the integral in the equation does not exist, necessitating a stochastic (i.e., randomized) approach to solving it. Kajiya [Kaj86] first introduced the idea of path tracing for solving the integral.

2.4.2 Monte Carlo Approximation

Monte Carlo methods can be used to approximate the LTE integral in Equation 2.1. Monte Carlo integration is easy to implement because it only requires the ability to

evaluate the integrand $f(x)$ to estimate the value of the integral $\int f(x)\,dx$ [PH10, p. 638].

To simplify the topic, the basic theory is presented here using one-dimensional integrals. It can be shown that for a one-dimensional integral $\int_a^b f(x)\,dx$, the expected value $E[F_N]$ of the Monte Carlo estimator

$$F_N = \frac{b-a}{N}\sum_{i=1}^{N} f(X_i) \tag{2.2}$$

is equal to the integral, where $X_i \in [a, b]$ are uniformly distributed random variables and N is the number of samples [PH10, p. 641]. In other words, by sampling $f(x)$ randomly, and by summing the results, the value of the integral can be approximated.

For an arbitrary probability distribution function (PDF) $p(x)$ the estimator takes the form

$$F_N = \frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}. \tag{2.3}$$

As before, the expected value of this form is equivalent to the integral specified earlier. Note that a uniform probability distribution $p(x) = 1/(b-a)$ makes Equation 2.3 equal to Equation 2.2.

It is possible to extend the estimator to multiple dimensions [PH10, p. 642–643], required when evaluating the path tracing formulation of the light transport equation. For example, if samples are taken uniformly from a 2D rectangular area from (x_0, y_0) to (x_1, y_1), the estimator becomes

$$F_N = \frac{(x_1-x_0)(y_1-y_0)}{N}\sum_{i=1}^{N} f(X_i). \tag{2.4}$$

Here the PDF is a constant value $1/((x_1-x_0)(y_1-y_0))$. Extensions to higher dimensions can be done similarly. In path tracing the Monte Carlo method is applied by randomizing the direction (ω_i in LTE) of the rays.

Because the Monte Carlo estimator merely approximates the integral, there is inevitably some error in the result. This error, or rather its variation from pixel to pixel, is what shows up as *Monte Carlo noise* in the rendered images.

The error of the Monte Carlo estimator decreases at a rate of $O(\sqrt{N})$, where N is the number of samples [PH10, p. 643]. Thus, to decrease the error by a factor of k, the number of samples taken has to increase by a factor m such that $\sqrt{mN}/\sqrt{N} =$

$k \Leftrightarrow m = k^2$. For example, if the number of samples taken is quadrupled ($m = 4$), error is cut in half ($k = 2$). Notably, Monte Carlo estimator's convergence rate does not depend on the dimensionality of the integrand [PH10, p. 643].

Efficiency of the Monte Carlo method can be improved by applying *importance sampling*. The key idea in importance sampling is to use a PDF $p(x)$ which is similar to the integrand $f(x)$. This way, more samples are taken from the "important" areas where the integrand's value is larger (and thus has more of an effect on the value of the integral). For example, when applied to path tracing, importance sampling might direct more secondary rays toward light sources. Note, however, that a poorly chosen distribution can end up increasing variance. [PH10, p. 688]

To illustrate the behavior of a Monte Carlo estimator and its dependence on sample count, consider that we want to calculate value of the definite integral $\int_0^\pi sin(x)\,dx$, i.e., $f(x) = sin(x)$, $a = 0$, and $b = \pi$, using the terminology that was established before. Unlike the integral in LTE, the true value of this integral can trivially be solved analytically: $\int_0^\pi sin(x)\,dx = -cos(x)\Big|_0^\pi = 2$.

In Figure 2.5 multiple runs of Monte Carlo estimation of the same integral using Equation 2.2 are shown. In other words, different random numbers were drawn for each of the 3 runs. The figure illustrates the high amount of variation in different estimates *of the exact same function* when the sample count is low, and how all estimates converge towards the correct result 2 as the sample count is increased. It also illustrates that when the sample count is high, new incoming samples appear to have less and less of an effect on the result. This happens because each sample is scaled by $1/N$, thus each sample has less of an effect on the overall value when the sample count is high.

To extend this example to images, we could think of the 3 runs as being 3 adjacent pixels in an image. All of the pixels are supposed to have the same value (2). Even though the integral is exactly the same for all pixels (which is generally not the case in path tracing), there still is variation in the estimate from pixel to pixel. This variation in the amount of error shows up as noise in the resulting image. As the sample count increases, the variation decreases, and the noise diminishes. In animations, the variation would also appear temporally as flicker, unless the exact same random numbers are used each frame.

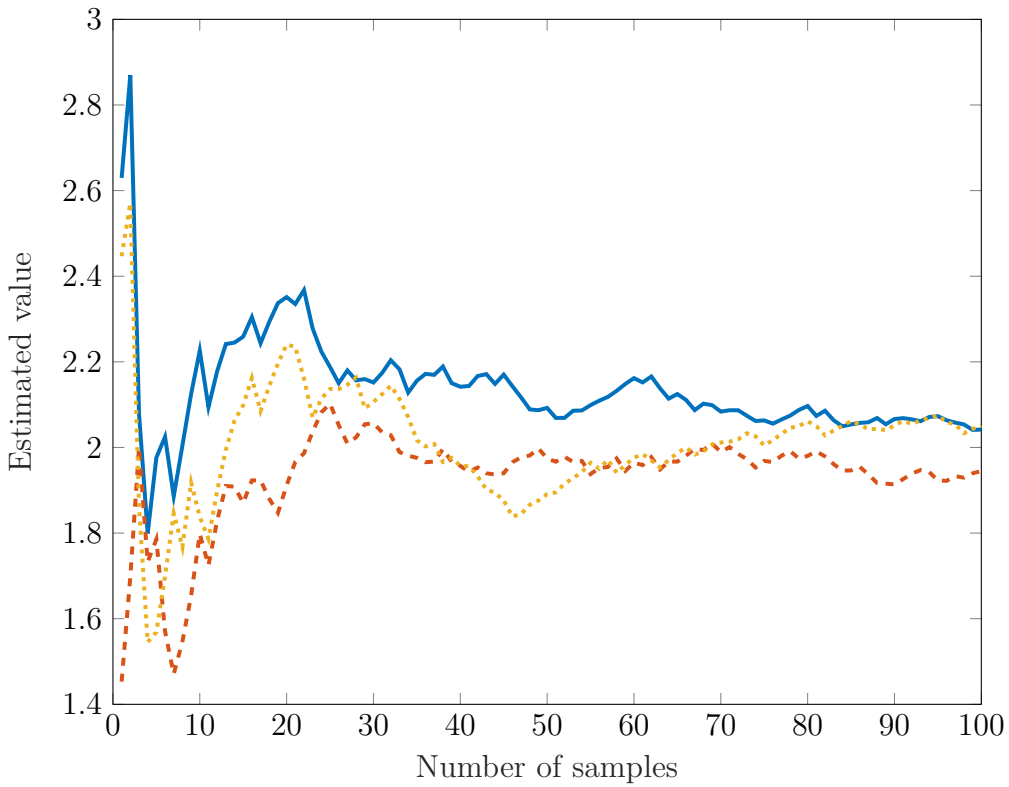


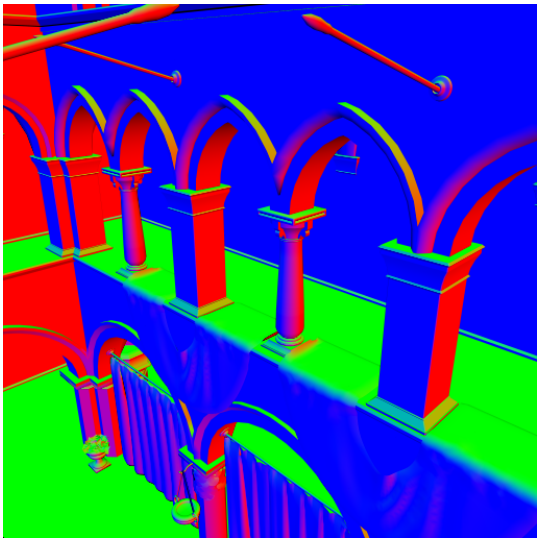
Figure 2.5 *Estimating an integral with the Monte Carlo method. Three runs, where the definite integral $\int_0^\pi sin(x)\,dx$ is estimated with Equation 2.2, are shown. The intermediate values of the estimates differ, but all of them converge towards the correct result 2.*

2.5 Geometric Features

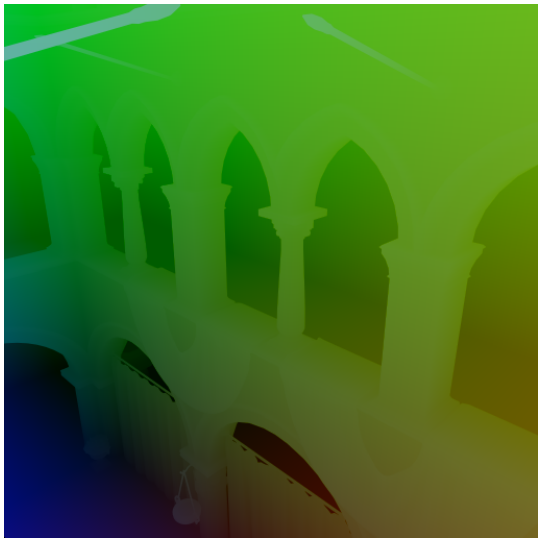
In rasterization and ray tracing, per-pixel geometric features of the scene can be collected during the rendering process to aid in post-processing steps such as denoising. The buffers that hold the features are often referred to as G-buffers (short for "geometric buffers") [ST90]. Some examples of these features are surface normals, depth values, and texture values [AMHH08, p. 279–280]. In renderers (including ray tracers and rasterization-based renderers) these features are usually available as by-products of rendering, because they need to be calculated regardless to determine the color at the corresponding surface point.

Even though the features are readily available, they are not without overhead, because some memory is required to store them for later use, and this in turn also generates additional memory traffic. The denoising algorithm is, of course, also complicated by having to take the feature buffers into account.

The features are collected for each point on the image plane. An illustration of the feature buffers for the features described below is shown in Figure 2.6. Later, in Figure 5.4 in Section 5.5, a practical comparison of the effect of different features



(a) Normal



(b) Position

(c) Texture

(d) Path traced (16k spp)

Figure 2.6 *Feature buffers and the corresponding path traced image with 16k spp. Not one feature is sufficient to capture everything. While the normal buffer (a) captures much of the edges and the form of the geometry, it includes no information about surface textures. The texture information (c) doesn't capture edges between geometry when both sides of the edge have similar texture values. The position information (b) is needed to recognize object boundaries of objects that have similar normals. None of these features directly capture distribution effects such as depth of field and motion blur.*

on denoising is shown.

Surface normal describes the normal at the hit point of the primary ray. A normal buffer typically works well to describe features within a single object. However, it is not uncommon to run into situations where it fails to capture edges between multiple objects. Consider, for example, a scene with a plane in the background, with its normal pointing directly towards the camera, and an axis-aligned cube in front of the plane. The frontmost face of the cube has the same normal as the plane in the back, and thus the edges between the objects would not show up in a normal buffer.

World space position is the position of the hit point in the world space coordinate system. That is, the position is independent of the view (camera) transform. World space position can be used to disambiguate some situations where the surface normal alone is not sufficient. For example, the edges in the "plane and cube" scene described previously would be captured by a world space position buffer.

Depth value is the distance traveled by the primary ray from the ray's origin (eye). The buffer that stores these values is often called the Z-buffer, because it stores the Z coordinate of a surface point in view space. Therefore, the depth value is closely related to the world space position. However, it has an advantage in storage and memory bandwidth usage, since it uses only one third of the storage space required by position information. Also, note that it's possible, if needed, to map back to world space position by means of an inverse transform of the view space X, Y and Z coordinates, all of which are known.

Texture value is the texture color at the hit point. Because features included in textures are not present in any of the other feature buffers (except the noisy color buffer), utilizing them can preserve a significant amount of detail that would otherwise be lost, especially when large kernel sizes are used.

Features are typically collected from the hit point of the primary ray. Technically, it would be possible to collect features from the following hit points as well, as they, too, can contribute to the final value of a pixel via reflections and other such phenomena. See, e.g., the machine learning approach used by Kalantari et al. [KBS15], where they use the texture value of the secondary hit point.

3. GENERAL IMAGE FILTERS

In this chapter, general image filters will be covered, i.e., image filters that are targeted towards general denoising of images (usually photographs) rather than denoising Monte Carlo renderings. With some modifications, these methods are also applicable to Monte Carlo denoising.

The terminology used in this chapter, and the rest of the thesis, is as follows. *Kernel size* refers to the number of rows and columns in a kernel matrix. Only square matrices will be considered in this thesis, as filter matrices are usually symmetric, and thus necessarily square. Note that the term *kernel radius* is used in some sources (e.g., MATLAB documentation) to refer to kernel size divided by 2. Kernel size is usually directly related to the *bandwidth* of the kernel, parameterized by the standard deviation σ, or "sigma", in a Gaussian kernel. These concepts and their meaning will be covered in more detail in the following subsections.

3.1 Weighted Average

Image filters can be implemented by calculating a weighted average of neighboring samples [GW01, p. 116]. The weights can rely on a variety of parameters, such as spatial distance (Gaussian blur), color distance (bilateral filter, see Section 3.4), or feature distance (cross-bilateral filter). Distances are calculated with respect to the central sample that is being processed. In addition, different filter shapes can be used. Filter shapes are discussed in Section 3.2. An example of a weighted average filter is shown in Figure 3.1.

In practice, weighted average implements discrete convolution [GW01, p. 118]. Convolution (denoted by $*$) is a mathematical operation which can be defined as follows in the 2D case:

$$g(x,y) = f(x,y) * h(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n)\, h(x-m, y-n), \tag{3.1}$$

where f and h are signals of size $M \times N$ that are convolved [GW01, p. 162]. Note that the result of convolution g is a function. In this case h can be considered the

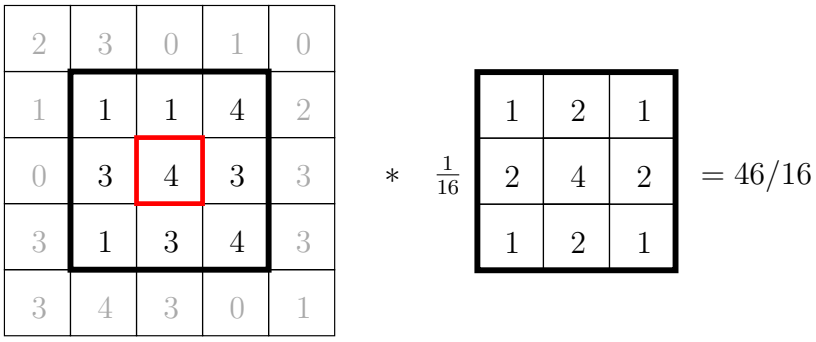


Figure 3.1 *Weighted average filter. Each element in the output image is calculated as a weighted sum of input image elements in the neighborhood of the current pixel (red). In this example, a* 3×3 *Gaussian blur kernel is used to calculate one element in the output image.*

filter kernel (coefficients), and f represents the image that is being filtered. However, convolution is a commutative operation, so the order of operands is not significant in practice.

3.2 Filter Kernels

Various different filter kernels, each with different properties, can be used depending on the needs of the application. A commonly used kernel is the Gaussian kernel, based on the Gaussian function

$$h(x) = \sqrt{2\pi}\sigma A e^{-2\pi^2\sigma^2 x^2} \tag{3.2}$$

where A defines the height of the curve and σ is the standard deviation that defines the shape of the filter [GW01, p. 164].

If a kernel is *separable*, convolution with it can be divided into a horizontal pass over the whole image, followed by a vertical pass [Smi97, p. 404]. This property can lead to significant performance improvements with large kernel sizes over a non-separated implementation. A 2D filter kernel is separable if its kernel matrix can be represented as a product of two vectors (the vectors need not be equal), as shown in Figure 3.2. In a non-separated implementation, a filter kernel with size n has a per-sample complexity of $O(n^2)$, whereas the complexity of a separated implementation is $O(n)$ [Smi97, p. 406]. Unfortunately many of the more complex filters, such as the bilateral filter described later in Section 3.4, are not separable.

Box filter, where all of the kernel coefficients are equal [GW01, p. 120], is also common due to its simplicity and ease of computation. In fact, a box filter is not only separable, but it can be calculated as a moving average with constant time

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Figure 3.2 *Example of a separable kernel. A filter is separable if its kernel matrix can be represented as a product of two vectors.*

complexity (per sample) regardless of the filter size [Smi97, p. 407]. As the window slides over a set of samples, the new sample in front of the window is added into the sum, while the last one at the back is subtracted out. Note that because of dependencies between samples, moving average cannot be calculated in parallel per sample. However, it is possible to parallelize the calculation of each row (or column) in a separated implementation, although in a highly parallel device this might not provide sufficient occupancy.

Alternatively, a 2D box filter output can be calculated by using a summed-area table (SAT) [Cro84]. Each element in SAT stores a sum of all values to the left and up from the current element in the source image. SAT allows to calculate a sum over an arbitrary rectangular area with just 4 lookups into the SAT. However, SAT has to be calculated in a separate preprocessing step. Also, extra precision might need to be allocated for the SAT elements (especially if fixed-point representation is used), because the area sums can grow much larger than the elements in the source array.

3.3 Hierarchical Gaussian Approximation

A Gaussian filter can be approximated by filtering the signal iteratively starting with a so-called *generating kernel*. The method is called hierarchical discrete correlation (HDC), and was presented by Peter J. Burt in 1981 [Bur81]. The key idea in the method is to calculate the correlation of a function with a large kernel as a weighted sum of correlations with smaller kernels.

The algorithm starts by using a small kernel size, and increases the size for each iteration as it progresses. The kernel is expanded by adding zeros in-between the coefficients of the previous level, which means that all iterations have a comparative computational cost, as the zero coefficients don't need to be considered in calculations. Note, however, that the memory access pattern changes slightly in each iteration, which can result in variable performance in realized implementations.

The kernel size grows rapidly with each subsequent iteration, meaning that large filter sizes can be reached with relatively few iterations. For example, with a starting

kernel size of 5, a kernel size of 80 can be reached with 5 iterations (5, 10, 20, 40, 80). The hierarchical approximation is particularly useful for large filter kernels, as in a naïve non-separated implementation the computational cost of an averaging filter scales quadratically with respect to the filter kernel's size.

While Burt presents the method in terms of calculating correlations, the same principles apply to calculation of convolutions, because correlation and convolution are equivalent operations under a reflection of the kernel [Bur81]. In other words, a correlation with a certain set of filter coefficients is equivalent to a convolution with a mirrored set of coefficients. It follows that if the filter is symmetric, correlation and convolution are the same operation.

3.4 Bilateral Filter

Bilateral filter was first introduced by Carlo Tomasi and Roberto Manduchi in 1998 [TM98]. It is an edge-preserving, nonlinear filter. Like traditional filters, it's also noniterative and local. In a non-iterative filter, the iteration count doesn't depend on the input, unlike in other edge-preserving filtering techniques like anisotropic diffusion. Being local, it only requires data from the neighborhood of the pixel that is being processed.

Using the notation of Durand and Dorsey [DD02], the bilateral filter can be defined by the equation

$$J_s = \frac{1}{k(s)} \sum_{p \in \Omega} f(p - s)\, g(I_p - I_s)\, I_p, \tag{3.3}$$

where

s	is the pixel position,
J_s	is the output of the filter,
Ω	is the neighboring region of the pixel,
p	is a point within the neighborhood,
I_p, I_s	are the pixel values,
f	is the filter kernels for the spatial domain,
g	is the filter kernel for the intensity (range) domain, and
$k(s)$	is a normalization term.

The normalization term $k(s)$ is defined as

$$k(s) = \sum_{p \in \Omega} f(p - s)\, g(I_p - I_s). \tag{3.4}$$

Cross-bilateral filter (also known as the joint-bilateral filter) is an extension on the idea of bilateral filter. It was formulated in 2004 in two separate instances [ED04, PSA+04] related to enhancement of photographs based on two source images of the same scene in a dark environment: one taken with a flash, and the other one without. One of the images is used as guiding image, while the other one is being filtered. As an extension of bilateral filter, the cross-bilateral filter can be defined as

$$J_s = \frac{1}{k(s)} \sum_{p \in \Omega} f(p - s)\, g(K_p - K_s)\, I_p, \tag{3.5}$$

where I is the image being filtered, and K is a guiding image. The normalization term $k(s)$ is calculated similarly as in the bilateral case:

$$k(s) = \sum_{p \in \Omega} f(p - s)\, g(K_p - K_s). \tag{3.6}$$

It's easy to see that as a generalization of the bilateral filter, the cross-bilateral filter reduces to the bilateral filter when $K = I$.

3.5 Guided Filter

Guided filter [HST10, HST13] is an edge-preserving filter based on a local linear model. Like cross-bilateral filter, guided filter can use a separate guidance image as its input. The computational complexity of guided filter is $O(N)$, where N is the number of pixels, independent of the filter kernel's size. In contrast, a brute-force implementation of bilateral filter runs in $O(Nr^2)$ time, where r is the kernel size. Also, according to authors, guided filter behaves better (with respect to image quality) than bilateral filter near edges. Note that guided filter doesn't attempt to approximate bilateral filter [HST13], but can give comparative results.

Guided filter is based on an assumption of a local linear model between the guidance image and the filtering output. The model can be expressed in equation form as follows:

$$q_i = a_k I_i + b_k, \forall i \in \omega_k, \tag{3.7}$$

where q is the filter output, I is the guidance image, i and k are pixel indices, and ω_k is a neighborhood around pixel k. Linear coefficients a_k and b_k are unknown, and they are assumed to be constant in the window ω_k. [HST13] Note that the input image (later denoted p) does not appear in Equation 3.7.

The output q can additionally be modeled as $q_i = p_i - n_i$, where n represents the noise (or other unwanted features) in the input image p. The goal is to find out a

solution for the two unknown variables a_k and b_k in the linear model by minimizing the difference between q and p. [HST13] For minimization, the authors present a cost model which has a solution as a function of the mean of I, the mean of p, and the variance of I, all in the neighborhood ω_k. Once a_k and b_k are known, the filter output can be calculated.

One more step is required because the output q_i can vary depending under which window it is calculated. The authors choose to average all possible values of q_i. Due to the symmetry of the box window, the output q_i can finally be expressed as $q_i = \bar{a}_i I_i + \bar{b}_i$, where $\bar{a}$ and $\bar{b}$ are the average coefficients of all of the windows that overlap i. The authors show that this filter in fact implements a weighted average.

Algorithm 1 Guided filter [HST13]

Input: input image p, guidance image I, regularization parameter ϵ
Output: filtered output image q

Part 1: $mean_I \leftarrow f_{mean}(I)$
$mean_p \leftarrow f_{mean}(p)$
$corr_I \leftarrow f_{mean}(I \odot I)$
$corr_{Ip} \leftarrow f_{mean}(I \odot p)$

Part 2: $var_I \leftarrow corr_I - mean_I \odot mean_I$
$cov_{Ip} \leftarrow corr_{Ip} - mean_I \odot mean_p$

Part 3: $a \leftarrow cov_{Ip} \oslash (var_I + \epsilon)$
$b \leftarrow mean_p - a \odot mean_I$

Part 4: $mean_a \leftarrow f_{mean}(a)$
$mean_b \leftarrow f_{mean}(b)$

Part 5: $q \leftarrow mean_a \odot I + mean_b$

The final algorithm is shown in Algorithm 1. Element-wise multiplication and division are denoted $\odot$ and $\oslash$, respectively. The function f_{mean} is a box filter, which can be implemented in $O(N)$ time regardless of the filter size, as discussed in Section 3.2, thus giving the complete algorithm $O(N)$ complexity also. The algorithm itself is divided into 5 parts. Part 1 computes the mean of I and p, the autocorrelation of I, and correlation of I and p. The variance of I and the covariance of I and p are calculated in Part 2, and they can be used to solve a and b in Part 3, as described earlier. In Part 4, the mean of the coefficients a and b are calculated (corresponding to variables $\bar{a}$ and $\bar{b}$ from earlier), and they are used to calculate the final result in Part 5.

The algorithm is shown for the one-dimensional case (both p and I are grayscale images). If the guidance image is multidimensional (e.g., an RGB image), a_k becomes a vector, and the division by the variance of I has to be handled as a multiplication by a matrix inverse of the covariance matrix of I. Multidimensional p is handled

simply by applying the filter to each channel individually. [HST13]

Recently, a faster variant of guided filter [HS15] was proposed. In this variant the input image and the guidance image are subsampled by a configurable subsampling factor, reducing the complexity to $O(N/s^2)$ for a subsampling factor s. Compared to Algorithm 1, I and p are first subsampled, and the filter size is adjusted by the subsampling factor. After $mean_a$ and $mean_b$ have been calculated (Part 4 in Algorithm 1), they are upsampled to the original image resolution and then the final result image is calculated based on the upsampled coefficients. Naturally, because of subsampling the results do not exactly match the results of the original guided filter. In many cases, however, the difference in quality is negligible.

3.6 Non-Local Means

A non-local means filter [BCM05] exploits the high degree of redundancy that exists in natural images [BCM04]. In other words, for any given window of pixels, there typically exist many other windows that are similar. Thus, for a given pixel within a noisy window, the true (non-noisy) value of the pixel can be predicted based on all of the windows that are similar [BCM04]. The non-local means filter calculates the pixel output as an average of all the pixels whose neighborhood looks similar.

For discrete images, the non-local means filter output can be specified in a general form as [BCM04]

$$NL(v)(i) = \sum_{j \in I} w(i,j)\, v(j), \tag{3.8}$$

where v is the noisy image, i is the coordinate of the output pixel, I is the set of all coordinates in the image, and w is a weighting function. The weighting function compares the neighborhood of i to the neighborhood of j and returns a value describing their similarity.

The size and shape of the similarity window can be varied, but for simplicity a square window is often used. The weight calculation can be likewise customized, implementations based on the Gaussian function being a common choice. In practical implementations, the search window as well as the neighborhood window can be limited for increased performance, and a normalizing factor should be introduced to ensure that the weights sum up to 1.

3.7 Advanced Filters: BM3D

Filters more advanced than the aforementioned general filters have been developed for denoising still images. One of these filters, dubbed Block-Matching and 3D Filtering (BM3D) [DFKE07], has also been used for denoising Monte Carlo images [KS13]. Because of this, BM3D will be briefly covered at a high level. Its application to Monte Carlo denoising is covered in more detail in Section 4.5.

BM3D starts by doing a *block-matching* step. This step is similar in purpose to the search for similar windows in non-local means, as discussed in Section 3.6. In other words, block-matching searches for blocks which have a high correlation to the currently processed block. The block-matching is improved by doing a coarse initial denoising in local 2D transform domain. As a result, a set of coordinates of similar blocks (based on some threshold of similarity) is generated.

Denoising is then done in 3D transform domain. The noisy blocks found in the previous step are stacked into a 3D array. A sparse representation of the signal is obtained by doing a 3D transform. Hard-thresholding of the transform coefficients attenuates the noise, and an inverse transform generates a stack of estimates of the true (non-noisy) image blocks. A weight is also generated for each estimate based on the number of non-zero transform coefficients after thresholding. Less noisy estimates are given greater weights.

Finally, the results are aggregated. The final estimate e is calculated as a weighted average of the local estimates, based on the weights calculated in the previous step. The quality depends on how many blocks were matched: the greater the amount of blocks, more overcomplete the representation of the estimated block is, and thus a better final estimate is produced.

The approach can be extended to Wiener filtering using the estimate e calculated previously. First, the accuracy of block-matching is improved by doing it with e. Second, the nonlinear hard-thresholding operator is replaced with the linear Wiener filter.

The initial estimation followed by the similar Wiener filtering part constitutes the full BM3D algorithm. However, some parametric choices have to be made for efficiency in practical implementations: (1) The maximum number of matched blocks is limited. (2) Block-matching is done within a local neighborhood of a fixed size, instead of over the whole image.

Additionally, there is some freedom in the selection of the transform that is used.

Any decorrelating transforms such as discrete cosine transform (DCT), discrete Fourier transform (DFT), and wavelets can be used. In the context of a real-time implementation, the most important properties are separability and the availability of fast algorithms.

3.8 Summary

To summarize, the output of most filters is calculated as a weighted sum of the input samples. Simple blurs can be implemented by basing the weights on the Gaussian function. A naïve implementation of a Gaussian blur has a per-pixel complexity of $O(n^2)$ (where n is the kernel size), whereas the complexity of a separated implementation is $O(n)$. Yet simpler way to implement a blur filter is to use the box filter, which can yield $O(1)$ complexity due to the optimizations that are possible because of equal coefficients. However, box filter produces worse results than a Gaussian filter. Gaussian blur can also be closely approximated by calculating it in a hierarchical manner. This approximation has sublinear complexity.

Bilateral filter is an edge-preserving filter, which extends the weight calculation to take the image color into account. This way, it is able to prevent filtering over dissimilar areas (edges) in the image. Cross-bilateral filter extends the idea further by using a separate image to guide the edge-preservation. Complexity for both filters is $O(n^2)$, and they are not separable. Guided filter is another edge-preserving filter, which has $O(1)$ complexity per-pixel. In other words, the complexity is independent of the filter size. A faster subsampling variant of the guided filter can reduce per-image complexity further.

Non-local means filter exploits the redundancy that exists in natural images. By looking for similar image patches elsewhere in the image, it is able to average out noise in the image while preserving edges. BM3D is a high quality denoising filter based on the same principal idea as non-local means. It looks for similar image blocks, stacks them into a 3D array, transforms the array, denoises in the 3D transform domain, and then transforms the results back. The computational complexity of these methods is highly dependent on implementation details, e.g., the size of the search window.

4. MONTE CARLO DENOISING METHODS

Many of the state-of-the-art methods used for denoising Monte Carlo renderings are based on the building blocks described in earlier chapters. In this chapter the state-of-the-art methods will be discussed, followed by discussion of how said methods combine the building blocks to form complete denoising algorithms.

Only methods that operate solely on the output image after the rendering process has completed will be considered. These methods have no knowledge of the input that went into the rendering process. In a recent survey [ZJL+15], this kind of methods were named "a posteriori", contrast with "a priori" methods, where analysis of the light transport equation is used to form the reconstruction filters.

Most "a posteriori" methods select a filter from a filter bank locally based on an estimation of the error of filter output [ZJL+15]. Naturally, the goal is to minimize the error in the filter output. However, since a "golden reference" result is not available at render time, the real output error is not known, and thus estimates have to be used. Many methods also include adaptive sampling, where new samples are iteratively distributed based on error estimates.

Some of the individual denoising methods will be covered in the following sections. Most of the methods presented here are designed primarily for offline rendering. Offline methods are included in the discussion in the interest of considering potential ideas to be adapted for real-time filtering. This discussion is not exhaustive—for a more complete coverage refer to the survey by Zwicker et al. [ZJL+15], although note that a few new methods have been published since, e.g., the work by Bauszat et al. [BEEM15], Kalantari et al. [KBS15], and Bitterli et al. [BRM+16]. Most recent work is by Chaitanya et al. [CKS+17] and Bako et al. [BVM+17]. Some older methods were omitted from the survey (e.g., adaptive manifolds [GO12]).

4.1 À-Trous

Dammertz et al. [DSHL10] use the hierarchical Gaussian approximation (see Section 3.3) to implement a real-time edge-preserving filter for Monte Carlo renderings.

Their filter's spatial component is based on so-called *algorithme à trous* ("algorithm with holes"), where each iteration the filter length is extended by adding zeros (holes) in between the filter coefficients. The number of non-zero elements in the filter stays the same for all iterations. Alternatively, this can be thought of as keeping the filter length constant, while subsampling the original signal.

They incorporate an edge-stopping function into the weight calculation by using feature buffers of color, normal, and world space position information. In essence, the resulting filter is formally quite similar to a cross-bilateral filter with an *à trous* approximation for the spatial kernel by applying multiple iterations.

The authors report an execution time of 5.6 ms (179 fps) for 5 iterations of the filter on an 0.26 Mpixel image, when run on an NVIDIA GeForce GTX 285 GPU with an implementation written in GLSL (OpenGL Shading Language). This number doesn't include the cost of uploading the ray tracing buffers to the GPU, which is required if a CPU-based ray tracer is used. Normalized to a 1 Mpixel image, the execution time is 22.4 ms (45 fps).

4.2 Guided Filtering

Bauszat et al. [BEM11] utilize the guided filter (see Section 3.5) for filtering path traced images with interactive frame rates.

The authors separate the light transport equation to direct and indirect illumination components. Direct illumination only includes light coming in directly from emissive surfaces. Indirect illumination includes light reflected via non-emissive surfaces. Only the indirect illumination component is filtered.

The guiding image used by the authors is a 4-dimensional image consisting of normal (3 components) and depth (1 component). This method has the same shortcomings as methods based on other filters such as the cross-bilateral filter: if the edges are not represented in the guiding image, they will not be preserved in the result.

With a 4-dimensional guiding image using an NVIDIA GeForce GTX 285 GPU, the authors report a filter execution time of 80 ms (12.5 fps) on a 0.79 Mpixel image. When normalized to a 1 Mpixel image, the execution time is 107 ms (9.4 fps).

4.3 Greedy Error Minimization

Rousselle et al. [RKZ11] describe an iterative adaptive sampling and reconstruction method based on minimization of MSE. Filters are selected from a preset number of Gaussian filters based on estimation of bias and variance of the Monte Carlo samples.

MSE can be calculated as a sum of the bias and variance terms. The filter selection method works on the assumption that filters with a small bandwidth have a small bias and large variance (i.e., amount of noise), whereas filters with a large bandwidth have a large bias and small variance. Additionally, it is assumed that the change in bias and variance is monotonic when going from fine to coarse scale filters. Filter selection is done by iterating over the filter banks in order from fine to coarse scales, and stopping when the change in MSE becomes positive.

Since a ground truth image is not available, estimates have to be used for the bias and variance terms. Variance can be estimated empirically in a straightforward manner, but bias estimation is more difficult. The authors describe a quadratic approximation that can be used to estimate bias without knowing the true pixel value.

4.4 Non-Local Means

Rousselle et al. improved on their previous work [RKZ11] by applying the non-local means filter to Monte Carlo denoising [RKZ12].

The method is based on 3 steps: sampling, filtering, and error estimation. First, the image is sampled, producing noisy images. Then, a non-local means filter is used to denoise the image. Third, the variance and residual error is estimated. The error estimate is used to create a sampling map. The algorithm is then repeated until a user-specified sampling budget is exhausted.

As an interesting idea, the method utilizes dual-buffer filtering. Two image buffers (A and B) are maintained, with the samples split evenly into both buffers. The weights computed based on buffer A are used to filter buffer B, and vice versa. This approach eliminates the correlation between the filter weights and the input signal, and therefore improves the output quality.

The filtering time for a 1 Mpixel image on an NVIDIA GeForce GTX 580 GPU with a CUDA (Compute Unified Device Architecture) implementation with 4 iterations is reported as 8.5 seconds (0.12 fps).

4.5 General Image Denoising Algorithms

Kalantari and Sen introduce a framework [KS13], which enables general image denoising algorithms to be used to remove noise in Monte Carlo renderings. General image denoising algorithms assume that the noise is spatially invariant, meaning that the noise exhibits the same characteristics throughout the image. One example of such noise is additive white Gaussian noise. The assumption about spatial invariance does not hold for Monte Carlo renderings, where the noise originates from the rendering process.

The two advanced image denoising methods discussed in the paper, Bayes Least Squares–Gaussian Scale Mixtures (BLS–GSM) and Block-Matching and 3D Filtering (BM3D), assume a fixed noise level σ. As such, they cannot be applied to Monte Carlo denoising without modifications.

The denoising algorithm starts by estimating the noise level (characterized by the standard deviation σ) in the neighborhood of each pixel by using median absolute deviation (MAD). The standard deviation from MAD is combined with standard deviation of the samples that the pixel is composed of. This is required to account for regions where different Monte Carlo effects overlap.

Next, the algorithm chooses a small number of standard deviations—a representative set—from the full set produced in the earlier step. The algorithm then proceeds to apply the image denoising separately with each standard deviation in the representative set, producing an equivalent number of filtered images.

As a final step, the filtered images are combined into the result image by linear interpolation. The filtered images participating in the interpolation are selected per-pixel by comparing the pixel's standard deviation estimate to the standard deviation used to filter the image. Two images with the closest standard deviation are chosen for interpolation.

This algorithm doesn't utilize feature byproducts of the rendering process, such as the normal, depth and position buffers. However, it does use the Monte Carlo samples that contribute to a pixel in the estimation of the standard deviation for that pixel.

On an Intel Xeon X5570 with 16 GB of memory, authors report that the full adaptive sampling and reconstruction framework takes 70.2 seconds, of which reconstruction takes about half. Most of the reconstruction time is spent executing the BM3D algorithm. The image resolution was not reported.

4.6 Weighted Local Regression

Moon et al. [MCY14] present an adaptive sampling and reconstruction method based on local regression theory. Feature bandwidths are selected locally, followed by applying local weighted regression to smoothen the image while preserving details. The method supports an arbitrary number of features, including noisy ones. A dimensionality reduction method, a truncated SVD (singular value decomposition), is used to reduce noise in the feature buffers.

In addition to typical features (such as normals, texture, and depth), the method utilizes extended features. Two texture buffers—one for moving, and one for non-moving objects—are used to separate the texture information of different object types. This separation is used to better handle images with motion blur effects in them.

The method is extended to reduce temporal noise (flicker) in animations by using temporal features, namely the frame number t. They use a $7 \times 7 \times 5$ filtering window, i.e., 5 frames are used temporally. In other words, the temporal distance (compare with spatial distance) is used to calculate the filter weight, and the bandwidth for the filter is automatically selected, as with all the other features. Similar temporal filtering extensions for the SURE [LWC12] and non-local means [RKZ12] methods are also described in the paper and its supplementary materials.

The authors report that on an NVIDIA GeForce GTX TITAN GPU with an implementation based on CUDA, on a 1 Mpixel image, the intermediate stages take 2 seconds of processing time, and the final stage takes 7 seconds. Thus, the execution time depends on the number of intermediate iterations. The intermediate iterations are used because the method also employs adaptive sampling. Assuming 5 iterations, processing time is approximately 17 seconds (0.06 fps).

4.7 Machine Learning

A recent approach by Kalantari et al. [KBS15] is to use a multilayer perceptron neural network to derive the filter parameters from the noisy scene data. The network is trained offline with noisy images containing a variety of distribution effects (e.g., depth of field, motion blur, and glossy reflections).

The method splits features into *primary* and *secondary* features. Primary features are the ones that are direct outputs from the rendering system, and they are inputs into the filter (e.g., a cross-bilateral filter). In the proposed method, 7 primary

features are used: (1) screen position, (2) color, (3) world position, (4) shading normal, (5, 6) texture values for the first and second intersection, and (7) direct illumination visibility.

Secondary features are calculated from the neighboring noisy samples. These features are inputs into the neural network, which attempts to determine the filter parameters that minimize the error between the filter output and the unknown ground truth value. The secondary features used in the method are: (1) the mean and standard deviation, (2) gradient magnitude, (3) mean deviation, (4) median absolute deviation (MAD), and (5) sampling rate. Most of the secondary features are calculated from the primary features. Also, note that secondary features are calculated for each primary feature individually, so, e.g., 5 MAD values are produced. In total, 36 secondary features are calculated for each pixel and input into the neural network.

The network is trained using back-propagation using noisy input images and converged ground truth images. The training attemps to find network weights such that the filter parameters given from the network output minimize the error between the filtered output and the ground truth image.

To reduce flickering in animations, a spatio-temporal filter with 3 neighboring frames on both sides of the current frame is used. This extension of the cross-bilateral filter can be done without retraining the network.

On an NVIDIA GeForce GTX TITAN GPU with an implementation based on CUDA, using a cross-bilateral filter, the approach takes 8.4 seconds (0.12 fps) to filter a 0.96 Mpixel image, of which feature calculation takes 5 seconds, and network evaluation and filtering takes 3.4 seconds. For animations, the filtering time increases sevenfold due to the spatio-temporal filtering. When normalized to a 1 Mpixel image, the total execution time for a single frame is thus 9.2 seconds (0.11 fps), and with spatio-temporal filtering about 64 seconds (0.02 fps).

Due to its genericity, this method can be used with arbitrary filters and an arbitrary number of features. However, the neural network has to be retrained if the filter type is changed, because the filter parameters are usually specific to the filter type.

4.8 Summary

Summary of the runtimes of the discussed Monte Carlo denoising methods is shown in Table 4.1. The image size used in the General Image Denoising method was not

Table 4.1 *Summary of Monte Carlo denoising methods. Runtimes were normalized to a 1 Mpixel image from the values reported by the authors. Note that the frame rate is a best-case number that assumes that all processing time is spent on denoising. Greedy Error Minimization [RKZ11] is omitted from the table, because its runtimes were not reported.*

Method	Runtime (ms)	(fps)	Hardware
À-Trous [DSHL10]	22	45.45	GTX 285
Guided Filtering [BEM11]	107	9.35	GTX 285
Non-Local Means [RKZ12]	8 500	0.12	GTX 580
General Image Denoising [KS13]	70 200	0.01	Xeon X5570
Weighted Local Regression [MCY14]	17 000	0.06	GTX TITAN
Machine Learning [KBS15]	9 200	0.11	GTX TITAN

reported, but was assumed to be in the same order of magnitude as other methods. For other methods, the reported times were normalized (by linear interpolation) to a 1 Mpixel image.

The most relevant method for this thesis is À-Trous, since it explicitly targets real-time applications. Guided Filtering can also reach runtimes that are close to real-time requirements. On more modern hardware, the runtimes should improve further. The other methods are targeted for offline rendering, and thus are able to provide a higher quality reconstruction at the cost of a high runtime. Even though the reported numbers are only approximative (due to varying hardware, possibly non-optimized implementations, and other factors), it is clear that the runtimes of the offline methods would have to be improved by a factor of 1000 to be applicable to real-time rendering.

5. EVALUATION

In this chapter the denoising methods that were implemented for this thesis will be described and compared to each other with respect to computational performance and the quality of the end result. The overall framework used for implementation is also detailed.

5.1 Framework

The general framework for implementations is based on the following successive steps:

(1) Sampling of the image plane based on the chosen sampling pattern.

(2) Filling in the missing pixels. (With an algorithm that might depend on the sampling pattern.)

(3) Denoising to remove Monte Carlo noise and artifacts from the filling process.

To start, the image is subsampled (step (1)). Sampling patterns are covered in more detail in Section 5.2. Because the image plane is sparsely sampled, a method is needed for filling in the non-sampled pixels. This is handled in step (2) with an algorithm that can be generic, or dependent on the sampling pattern. For example, if the sampling pattern samples uniformly once within every 2×2 grid, 3 pixels in the grid must be filled in some way. Note that the fill operation has to be applied to the feature buffers as well if they are subsampled. Also, note that because filling is done before noise removal, this step spreads the noise to the filled areas. Step (3) operates on a filled buffer to remove Monte Carlo noise and fill artifacts.

This framework was chosen because most of the filters used for Monte Carlo denoising assume that the images and features are sampled uniformly, thus by filling in the non-sampled pixels the framework allows the denoising algorithms to be used without modifications. While it is possible in some cases to modify the algorithms to

account for missing samples, in general it can be problematic. For example, to calculate a cross-bilateral filter output for a pixel location, a reference value is needed for comparisons with neighboring pixels. This reference value is not available if the pixel was not sampled. While the filter could account for this, e.g., by choosing the value of the nearest neighbor as the reference value (or a combination of values of multiple neighbors), this is highly dependent on the sampling pattern in use, and would complicate and slow down the filter.

5.2 Sampling Patterns

Sampling pattern refers to the way the image plane is sampled to achieve a variable sampling rate for foveated rendering. Typically, the pattern is constructed such that more samples are placed in the fovea than outside it. Naturally, there are many ways to construct these patterns. Note that the choice of a sampling pattern can also affect the way that the filling in of non-sampled pixels has to be performed.

For this work, a sampling pattern based on an expanded grid was selected. The idea is to start from a uniform grid that doesn't cover the whole image area, and expand the grid points outward based on a mathematical expression so that the expanded grid covers the full image (or close to it—non-sampled areas at the far borders of the image can be tolerated, since they are far from the center of the gaze, and thus not noticeable with a large enough field of view). This approach has a nice property that points can easily be mapped back and forth between the expanded and unexpanded grids by using the mathematical formula. Inside the fovea, uniform sampling can be used by not expanding grid points where the distance from the center of expansion is less than the desired radius of the uniformly sampled area.

Missing pixels can be filled in by mapping the expanded grid coordinate back to the uniform grid, and by sampling the non-expanded grid, since it is known that samples exist in each point of the non-expanded grid. It is also possible to use bilinear filtering to get better results than with nearest neighbor sampling. Alternatively, a general-purpose method of filling can be used, such as pull-push [GGSC96], which can be used with any sampling pattern.

5.3 Implemented Filters

Three filtering methods were implemented. They will be referred to as Simple Blur, Cross-Bilateral and À-Trous.

Simple Blur This method implements a Gaussian blur with a kernel size (blur amount) that varies depending on the distance from the central point of the gaze. Inside the foveal area, less blur is applied, leaving details untouched, while outside the foveal area more blur is applied, with the goal of reducing artifacts from the sparse sampling. This method is very simple, doesn't utilize information from the geometry features, and thus cannot preserve detail outside the fovea.

Cross-Bilateral This is an implementation of a cross-bilateral filter (covered in Section 3.4). Several feature buffers can be used to aid the filtering: color, surface normal, world space position and texture value of the first surface hit point. The parameters of each feature can be configured separately.

À-Trous The À-Trous implementation is based on the paper by Dammertz et al. [DSHL10]. This implementation utilizes the cross-bilateral filter as the basis because of similarities between the filters. The difference between the two methods is that À-Trous goes through several iterations of a cross-bilateral filter with a small kernel size, while varying the step size between iterations. Because the step size is related to the kernel size and grows as powers of two, few iterations of this filter can approximate larger cross-bilateral filters while taking significantly less time.

The rationale for selecting these methods is as follows:

(1) The Gaussian filter used in Simple Blur is easy to implement, and while its applications are limited, it provides a baseline that the other methods can be compared to. Moreover, it helped to provide a clearer idea about the problem space early on. For these reasons, it was the very first filter that was implemented.

(2) Cross-bilateral filter is a building block in many of the offline filtering algorithms that are available. Since it can utilize the feature buffers produced as byproducts of the rendering process, it can provide visually pleasing results even with low sampling rates, as long as the geometry features are well represented in the feature buffers.

(3) À-Trous was chosen because it is one of the few methods available that are explicitly targeted for real-time usage. Also, because it is an approximation of a cross-bilateral filter, it provides a good point of comparison to it.

Guided filter was considered for implementation because of its $O(N)$ complexity (where N is the number of pixels in the image) independent of the filter parameters, but was ultimately left out because it doesn't scale well to large number of feature

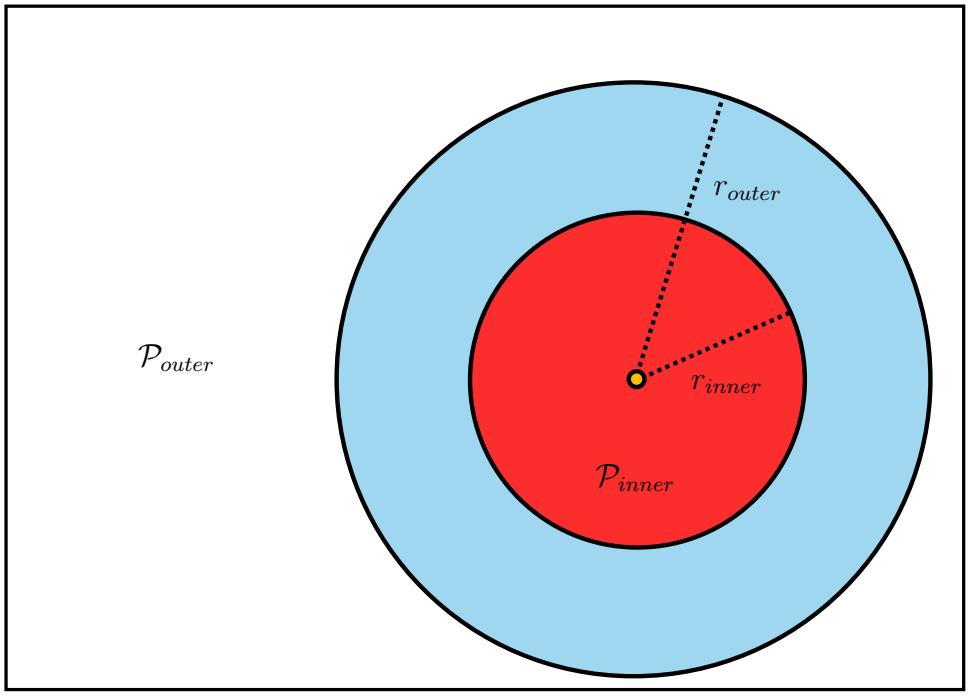


Figure 5.1 *Spatially varying parameters for foveation. In this example, user's gaze (the yellow dot) is pointed at the right side of the field of view. The inner region (distance* $\leq r_{inner}$*, red) uses a set of filter parameters* $\mathcal{P}_{inner}$*, while the outer region (distance* $\geq r_{outer}$*, white) uses a different set* $\mathcal{P}_{outer}$*, where distance is the distance from the point of gaze. In between the regions (*$r_{inner} < distance < r_{outer}$*, blue), the parameters are interpolated.*

buffers. The reason is that the number of dimensions in the guiding image affects the size of the covariance matrix required by the algorithm, as discussed in Section 3.5. The matrix inversion becomes troublesome to implement with a large matrix size, and it is not easy to provide a generic implementation of the filter that can handle an arbitrary number of features because of this. A generic implementation would be especially useful when prototyping, when the number of features required to achieve a sufficient quality is not yet known.

In order to account for foveation, all of the implemented methods allow the filter parameters to be varied based on the distance from the central point of the gaze by defining the parameters separately for an *inner region* and an *outer region*. Both regions are circular and are centered at the point of gaze. In between the regions (defined by radii), the parameters are linearly interpolated. This concept is illustrated in Figure 5.1.

Even though the filter parameters can depend on distance from the fovea and thus are spatially varying, they do not adapt to local characteristics of the image (e.g., by locally choosing filter parameters which minimize the error of the filter output). The reason is that a a sufficient algorithm for local selection of the parameters wasn't

discovered. Such algorithms are plentiful in offline methods, but none were found to be geared towards real-time use. However, this would be an interesting avenue for future work, because offline methods have shown that per-pixel parameter selection can significantly improve the filtering quality, while also being straightforward to integrate into an existing cross-bilateral filter. The challenge is finding a parameter selection algorithm which is sufficiently robust, while being able to provide stable real-time performance.

5.4 Computational Performance

In this section, the complexity of the implemented methods is discussed. Then, to get a better idea about practical performance, measurements from a real-life implementation are presented.

5.4.1 Analysis

Simple Blur method is a Gaussian blur with a varying kernel size. As the kernel size varies as a function of distance to the point of gaze, the expected performance is dependent on how fast the kernel size grows as distance grows.

In a non-separated implementation, assuming constant memory access cost and a constant amount of computation per pixel, the processing time for a single pixel is proportional to the square of the kernel size. Since the kernel size is spatially varying, the total expected processing time for the full image has to be calculated as a sum over the image space:

$$t = \sum_{y=0}^{h-1} \sum_{x=0}^{w-1} k(x, y, g_x, g_y)^2, \tag{5.1}$$

where w is the image width, h is the image height, g is the point of gaze, and k is the kernel size at a given point.

Because of the varying kernel size, this filter is not separable. However, it can be separated it if it's acceptable that the resulting effective filter does not reproduce the exactly same (rectangular) window of neighbors and filter weights as the original filter. Separation can significantly reduce the required processing time.

As far as performance goes, the Cross-Bilateral method is quite similar to Simple Blur. The main difference comes from a higher constant factor due to the feature weight computations. However, while a spatially variable Gaussian filter (and other

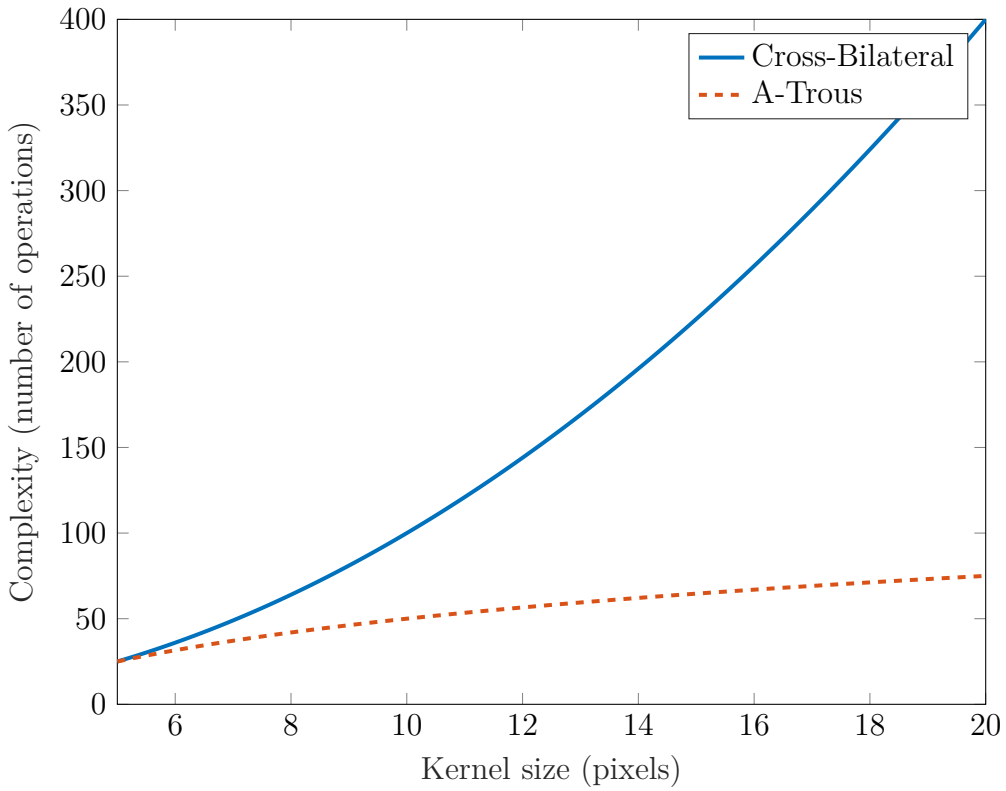


Figure 5.2 *Per-pixel complexity of Cross-Bilateral vs À-Trous with respect to kernel size with generating kernel size $k_g = 5$. Cross-Bilateral has quadratic complexity with increasing kernel size. In À-Trous, the complexity grows sublinearly because of the exponential term 2^{i-1} in the kernel size expression.*

symmetric filters, such as the box filter) can be "forcefully" separated, it does not make sense to do this for the cross-bilateral filter, because the horizontal and vertical feature weights have absolutely no relationship. Thus, the results would be completely wrong. The per-pixel complexity of Cross-Bilateral can be written as $O(fk^2)$, where f is the number of features and k is the kernel size.

A single iteration of the À-Trous method is closely related to the Cross-Bilateral method in performance. However, the filter kernel size (i.e., the generating kernel size) can typically be much smaller because larger filter sizes are enabled by the subsequent iterations. The final effective kernel size is then $k = k_g 2^{i-1}$, and per-pixel complexity is $O(ifk_g^2)$, where k_g is the size of the generating kernel and i is the number of iterations. Thus, the number of iterations required to achieve a specific kernel size is $i = log_2(k/k_g) + 1$. Assuming constant k_g, i can be solved from k, allowing us to compare the complexity to that of Cross-Bilateral. Comparison of complexity of À-Trous vs the Cross-Bilateral method is shown in Figure 5.2.

Another distinguishing factor in À-Trous is the memory access pattern, which varies in each iteration as the step size changes. Whereas in the first iteration the accesses

occur close to each other as step size is 1, in subsequent iterations the accesses become more and more spread out. The accesses do still exhibit a consistent pattern, however, and adjacent pixels would be accessing adjacent memory locations. For example, with step size 4, output pixel 0 might access memory addresses $0, 4, 8, \ldots$, while pixel 1 accesses addresses $1, 5, 9, \ldots$, respectively. Thus, it's expected that the performance of different iterations should be fairly similar.

Overall, there is an inverse relationship between the amount of samples per pixel and the workload of the filter. In other words, less samples that are allocated to each pixel, more work the filter has to do to reconstruct a visually pleasing image. In practice this usually means that a larger filter kernel must be used when the sampling rate is lower. However, because the maximum kernel size is limited by the time constraint set by real-time rendering (e.g., 20 ms per frame for a frame rate of 50 frames per second), the sampling rate has to be set such that no more noise is present than the filter is capable of removing.

5.4.2 Measurements

Measurements were collected with AMD CodeXL 2.2 on a PC running 64-bit Windows 10. CodeXL is an application that can be used for GPU and CPU (central processing unit) profiling and static kernel analysis [AMD17]. The system included an AMD Radeon R9 Fury X GPU with a 1050 MHz core clock and 4 GB of memory, and a 4 GHz Intel Core i7-6700K CPU with 16 GB of system memory. The host application was built with Microsoft Visual Studio Community 2015.

Note that in the following discussion the term "kernel" is—depending on context—used to mean both the image filter kernels (see Section 3.2), and OpenCL kernels (functions run on the OpenCL device).

The denoising methods were implemented as OpenCL kernels and run on the GPU. Because of focus on prototyping, the implementations are quite straightforward, and therefore not optimized; this should be kept in mind when interpreting the results. Also, note that the Cross-Bilateral implementation is somewhat slowed down by its genericity. Features can be dynamically enabled, which adds overhead to the kernel even when the features are disabled. In addition, the configurable step size (required for implementing À-Trous on top of Cross-Bilateral) adds some overhead.

Runtimes for the Simple Blur and Cross-Bilateral methods with different kernel sizes are shown in Table 5.1. The measurements were averaged from 50 executions of the kernel. The kernel size is constant in these measurements, i.e., it is not varied based

Table 5.1 *Runtimes (in milliseconds) for the Simple Blur and the Cross-Bilateral implementations with kernel sizes that are constant over the image space. Image size was* 1024×1024*. The methods denoted Cross-Bilateral (1–3) use 1–3 features, respectively.*

	Kernel size (pixels)							
	1	3	5	9	17	33	65	129
Simple Blur	0.1	0.2	0.5	1.4	5.1	18.1	62.0	228.6
Cross-Bilateral (1)	0.3	0.6	1.1	3.1	10.6	37.7	128.2	505.9
Cross-Bilateral (2)	0.3	1.0	2.4	6.4	17.4	59.5	210.3	767.2
Cross-Bilateral (3)	0.3	1.4	3.4	9.6	29.6	86.8	281.9	1003.0

on distance from the gaze point. The results mostly exhibit the expected quadratic growth as the filter kernel size is doubled. As more features are added, the runtimes show close to linear growth.

The deviations from the expected in the larger kernel sizes can be explained by the border pixels that are rejected. With growing kernel size, more pixels are rejected, causing a lower runtime. For example, for a 1024×1024 pixel input image, a 65 pixel kernel produces $(1024 - 2 \cdot 65)^2$ output pixels, while a 129 pixel kernel produces only $(1024 - 2 \cdot 129)^2$ pixels—approximately 27% less. Note that this is an artifact of the current implementation, and different methods for handling the border pixels would change the results. When this fact is taken into account, the results more closely line up with the complexity analysis.

The device occupancy (as reported by CodeXL) was 100% for all invocations of the Simple Blur kernel, but only 40% for the Cross-Bilateral kernel, due to the additional complexity of the Cross-Bilateral kernel.

Runtimes for different iterations of the À-Trous method are shown in Table 5.2. Again, a constant kernel size was used. There is no significant difference in runtimes between different iterations, despite the variable step size utilized in the algorithm. In fact, the runtimes for later iterations are lower, which is probably caused by the skipping of border pixels. With $k_g = 3$ and $k_g = 5$, the runtime goes up in the 6th iteration. (With $k_g = 9$ this already happens in the 5th iteration.) A probable cause is the caching behavior, or other such "hidden" factors in the target hardware.

5.5 Visual Quality

In this section, the visual quality of the implemented methods is evaluated. Note that because the used rendering system doesn't support alpha channel in textures, pixels with alpha in them appear black in the following comparison images.

Table 5.2 *Runtimes for the À-Trous implementation with generating kernel sizes that are constant over the image space. Two features were used. Image size was* 1024×1024*. The kernel runtime (in milliseconds) is shown for each iteration and generating kernel size* k_g*. A cumulative sum of execution times is shown in the column marked* Σ*, and* k *is the effective kernel size* $k_g 2^{i-1}$*.*

Iteration	$k_g = 3$	Σ	k	$k_g = 5$	Σ	k	$k_g = 9$	Σ	k
1	1.02	1.02	3	2.33	2.33	5	6.10	6.10	9
2	0.98	2.00	6	2.15	4.49	10	5.02	11.12	18
3	0.92	2.92	12	1.77	6.26	20	4.37	15.50	36
4	0.80	3.72	24	1.47	7.73	40	3.73	19.23	72
5	0.67	4.39	48	1.34	9.07	80	3.92	23.15	144
6	0.72	5.11	96	1.50	10.57	160	2.86	26.00	288

Without foveation. To analyze the filtering independent of foveation, the Cross-Bilateral and À-Trous methods are compared to each other in Figure 5.3 without spatial variation in the filter parameters. Spatial kernel size 40 ($\sigma \approx 13.3$) was used for Cross-Bilateral. Generating kernel size $k_g = 5$ ($\sigma \approx 1.67$) with 4 iterations was used for À-Trous, providing comparative results. Normal, position and texture feature weights were used. The filter parameters were equal for both methods.

At 16 spp, Cross-Bilateral produces noticeably better results than À-Trous. The most obvious difference is that À-Trous shows artifacts at the edges of geometry. The edges correspond to areas where less filtering is done (due to the presence of an edge in a feature buffer), and thus it's more likely that the noise in the input shows through. It is also evident that À-Trous has a tendency to blur fine texture detail; see, e.g., the decoration at the edges of the cloth.

At 64 spp, the differences are less visible. The edge artifacts in À-Trous are less prominent; however, texture detail is still lost. Still at this number of samples, the sculpture depicting the head of a lion is significantly more deformed in À-Trous, whereas in Cross-Bilateral it looks slightly better. However, in both methods the material of the lion looks much more metallic than it should according to the ground truth image. A major factor seems to be the loss of shadows in the detailing of the head. In both input images (16 spp and 64 spp), it's also noticeable that there's an especially high amount of noise in the part of the image where the head is, leading to a lot of averaged colors in the filtered results, and causing a specular, metallic look.

Both algorithms are able to improve on the noisy input image and to maintain the general feel of the reference image. Some limitations are apparent, however. The shadows under the cloths and the top of arches are overblurred, since they do not

appear in the feature buffers. Some textures are corrupted (e.g., the red banner at the bottom left corner). The cloths also appear significantly darker than they should.

The inferior quality in the À-Trous method can be explained by its violations of the assumptions made in the formulation of HDC (Section 3.3). In HDC, latter iterations are able to subsample the image because earlier iterations have removed high frequency content from the image. With the edge-preserving modification in À-Trous, the high frequency content is still present in later iterations of the filter, making the sampling rate inadequate, as per the sampling theorem. Despite the artifacts, À-Trous still serves as a useful approximation because of its improved performance.

The blurring of the leaves in the middle in all images could have been avoided by using a smaller bandwidth for the texture feature. However, decreasing the bandwidth too much could cause overtracking of the features in other areas. Again, this goes to show that it's difficult to find global filtering parameters that are suitable for the whole image.

Note that while difficult to show in still images, when using a low number of samples, there is a high amount of temporal variation from frame to frame because of the inherent randomness in Monte Carlo sampling. In animated scenes, this shows up as flickering. While a filter can reduce temporal variation to some extent by spreading the variations over a large area, temporal variation is still present at the edges where less averaging is performed due to the edge-preserving functionality of the filters. Additional temporal filtering—which uses multiple frames as input—would be needed to mitigate the temporal variations further.

Effect of features. The effect of different feature buffers in the filtering result is shown in Figure 5.4. The corresponding normal, position and texture buffers for this scene can be seen in Figure 2.6. When just the spatial component (b) is used, the filter effectively acts as a Gaussian blur. Noise is removed, but all edges and detail are lost also. Note that the kernel size has been set to account for the worst case of noise within the image (recall that the amount of noise is spatially varying). In addition, note that the spatial component is, of course, also active in (c, d, e, f).

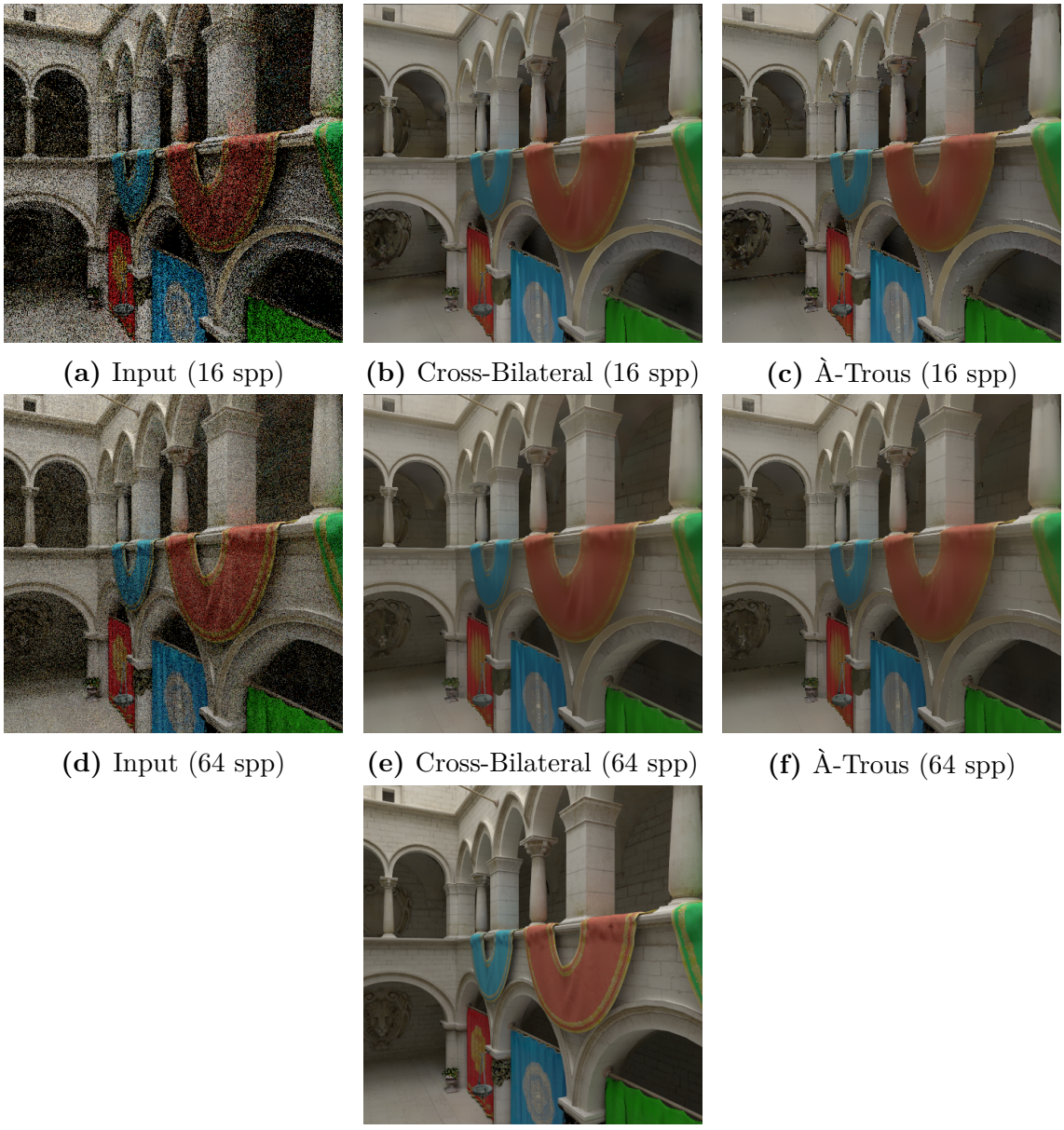
Problems are apparent in the pillars and the arches when just the normal weight is used (c). The position weight (d) is able to distinguish objects from each other, but doesn't preserve detail within objects. This happens partially because it is not possible to find a global bandwidth that would work with the significant scale differences in the position buffer. The normal buffer doesn't have this problem,

because all values are known to be within range $[-1, 1]$. The texture weight (e) preserves texture detail, but shows problems where two distinct objects have similar texture values. When all weights are used (f), most significant detail is preserved while the noise is removed.

With foveation. Comparison with foveation is shown in Figure 5.5. A sparsely sampled image (a) is filled in (b). For each pixel, 64 samples were used. A reference filled image at a high spp is shown in (c). The purpose of this image is to show which artifacts are a result of the Monte Carlo noise, and which ones are a result of the fill operation. In this case, the filling was done with pull-push interpolation [GGSC96]. Point of gaze was in the middle of the image. Filtered versions of (b) are shown in subfigures (d), (e), and (f).

In the filtered images, the spatial weight was increased towards the edge of the image, while the feature weights were decreased. In other words, the amount of blur was increased towards the image edge. The feature weights, on the other hand, had to be decreased towards the image edge, because the feature buffers also suffer from fill artifacts. Therefore, if feature weights were used, the result would track the artifacts of the feature buffer. Overall, the goal of this approach was to decrease the amount of artifacts resulting from filling at the edges.

Simple Blur is able to remove most of the peripheral artifacts that resulted from the filling. However, it cannot be enabled in the fovea because it would blur out details noticeably even with a small bandwidth, especially when seen close by on an HMD. Thus, noise is left in the foveal area. Cross-Bilateral is able to reduce fill artifacts, while also removing noise in the foveal area while maintaining some details. À-Trous produces similar results as Cross-Bilateral, however, more texture detail is lost as in Figure 5.3.



(a) Input (16 spp) **(b)** Cross-Bilateral (16 spp) **(c)** À-Trous (16 spp)

(d) Input (64 spp) **(e)** Cross-Bilateral (64 spp) **(f)** À-Trous (64 spp)

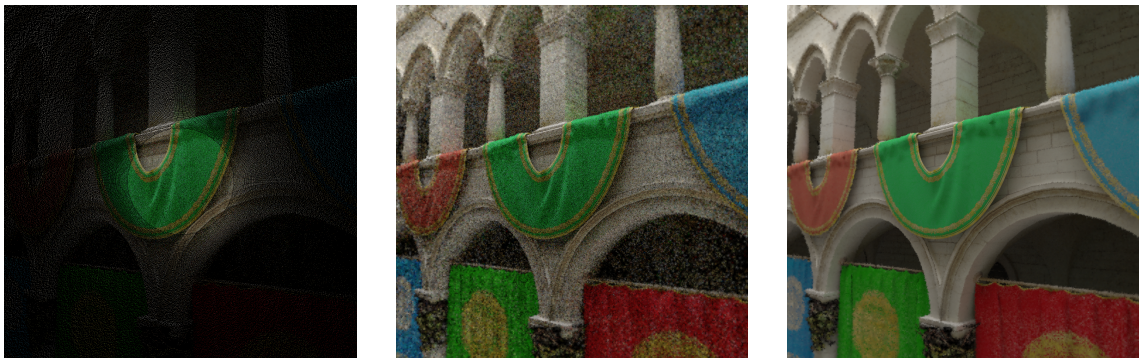
(g) Reference (16k spp)

Figure 5.3 *Visual quality of Cross-Bilateral vs À-Trous without foveation.*

(a) Input (64 spp) **(b)** Spatial **(c)** Normal

(d) Position **(e)** Texture **(f)** All

Figure 5.4 *Effect of different features. A cross-bilateral filter with kernel size 19 ($\sigma = 6$) was used.*



(a) No fill (64 spp) **(b)** Filled (64 spp) **(c)** Filled (16k spp)

(d) Simple Blur (64 spp) **(e)** Cross-Bilateral (64 spp) **(f)** À-Trous (64 spp)

Figure 5.5 *Visual quality with foveation.*

6. CONCLUSIONS

In this thesis, the theoretical background behind foveated rendering was presented. In foveated rendering, properties of the human visual system are used to optimize rendering performance by allocating more samples around the central point of vision where visual acuity is high, and by sampling sparsely in the areas of lower visual acuity. The path tracing method of rendering was given special consideration. Path tracing is a method for rendering photorealistic 2D images from 3D scene descriptions based on calculating intersections between the scene geometry and simulated rays of light.

Filters and denoising methods—which are used to combat the Monte Carlo noise that arises because of the lowered sampling rate in foveated path tracing—were discussed. Additionally, these filters can be used to remove artifacts that occur as a result of upsampling the sparsely sampled areas (i.e., from filling in the non-sampled pixels).

Finally, a few denoising methods suitable for real-time usage were implemented and integrated into an existing foveated rendering system based on path tracing. The runtime performance and the visual quality of the methods was evaluated.

The Gaussian blur filter cannot preserve detail in the image, and thus is only applicable for removal of fill artifacts. For a cross-bilateral filter, filter sizes larger than around 17×17 were not usable in practice because of the exponential slowdown as the filter size grows. The actual limit depends on the target frame rate and hardware, and how much time of the overall rendering budget is allocated to filtering. The À-Trous approximation of cross-bilateral filter can reach much higher filter sizes, with the disadvantage of a somewhat degraded image quality. However, there is also a limit to how large of a filter size is useful in practice—overly large filters tend to produce cartoon-like results.

Temporal flickering was especially noticeable when low sampling rates were used. Large filter kernels can be used to average out temporal differences in the image, but temporal variation still shows near edges and details, where the effective filter

size is smaller due to the feature weights. Spatio-temporal filters could be used to diminish the flicker, although none were tested within the scope of this thesis.

In conclusion, while the discussed denoising methods can provide visually pleasing results in some scenarios, they are not advanced enough to handle arbitrary input. The main limitation of the tested methods is that they can only preserve detail which is present in the feature buffers. Of the compared methods, cross-bilateral filter provides the best quality, but its runtime doesn't scale well to large filter sizes. The À-Trous method can be used as a fast approximation, but generates more artifacts in the result. Improvements in the algorithms are needed to reach the quality seen in offline methods. Furthermore, a user study would be needed to assess the effectiveness of noise removal in foveated rendering.

As future work, it would be interesting to explore opportunities for doing a local selection of the filter parameters in real time. Especially it would be interesting to find how much the visual quality of the À-Trous method could be improved by local parameter selection. Additionally, adaptive sampling could be used to improve the rendering quality by iteratively choosing the areas to be sampled based on error estimates. However, for real-time applications an additional challenge is ensuring that the rendering quality stays consistent over time with different camera placements and the presence of dynamic geometry in the scene.

It would also be interesting to attempt to extend BM3D for Monte Carlo denoising by supporting spatially varying filter bandwidths in a single pass, as multiple passes are not feasible for real-time applications. Feature weights could also be integrated into the similarity metric in the block-matching step.

BIBLIOGRAPHY

[Abr14] Michael Abrash. What VR could, should, and almost certainly will be within two years, 2014. Steam Dev Days, Seattle.

[AMD17] AMD. CodeXL, 2017. Available (accessed on 2017-04-18): http://gpuopen.com/compute-product/codexl/.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 3rd edition, 2008.

[BCM04] Antoni Buades, Bartomeu Coll, and Jean Michel Morel. On image denoising methods. Technical report, CMLA (Centre de Mathematiques et de Leurs Applications), 2004.

[BCM05] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. A non-local algorithm for image denoising. In *Proceedings of Conference on Computer Vision and Pattern Recognition*, pages 60–65. IEEE Computer Society, 2005.

[BEEM15] Pablo Bauszat, Martin Eisemann, Elmar Eisemann, and Marcus Magnor. General and robust error estimation and reconstruction for Monte Carlo rendering. *Computer Graphics Forum*, 34(2), 2015.

[BEM11] Pablo Bauszat, Martin Eisemann, and Marcus Magnor. Guided image filtering for interactive high-quality global illumination. *Computer Graphics Forum*, 30(4):1361–1368, 2011.

[BRM+16] Benedikt Bitterli, Fabrice Rousselle, Bochang Moon, José Iglesias-Guitián, David Adler, Kenny Mitchell, Wojciech Jarosz, and Jan Novák. Nonlinearly weighted first-order regression for denoising Monte Carlo renderings. *Computer Graphics Forum*, 35(4):107–117, 2016.

[Bur81] Peter Burt. Fast filter transforms for image processing. *Computer Graphics and Image Processing*, 16(1):20–51, 1981.

[BVM+17] Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, DeRose Tony, and Fabrice Rousselle. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. *ACM Transactions on Graphics*, 36(4), 2017.

[CKS+17] Chakravarty Alla Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics*, 36(4), 2017.

[Cro84] Franklin Crow. Summed-area tables for texture mapping. *ACM SIGGRAPH Computer Graphics*, 18(3):207–212, 1984.

[DD02] Frédo Durand and Julie Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Transactions on Graphics*, 21(3):257–266, 2002.

[DFKE07] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. Image denoising by sparse 3-D transform-domain collaborative filtering. *IEEE Transactions on Image Processing*, 16(8):2080–2095, 2007.

[DSHL10] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik Lensch. Edge-avoiding À-Trous wavelet transform for fast global illumination filtering. In *Proceedings of the Conference on High Performance Graphics*, pages 67–75. Eurographics Association, 2010.

[ED04] Elmar Eisemann and Frédo Durand. Flash photography enhancement via intrinsic relighting. *ACM Transactions on Graphics*, 23(3):673–678, 2004.

[GFD+12] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. Foveated 3D graphics. *ACM Transactions on Graphics*, 31(6), 2012.

[GGSC96] Steven Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael Cohen. The lumigraph. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 43–54. ACM, 1996.

[GO12] Eduardo Gastal and Manuel Oliveira. Adaptive manifolds for real-time high-dimensional filtering. *ACM Transactions on Graphics*, 31(4), 2012.

[GW01] Rafael Gonzalez and Richard Woods. *Digital image processing*. Prentice Hall, 2nd edition, 2001.

[HS15] Kaiming He and Jian Sun. Fast guided filter. Technical report, Microsoft, 2015. Available (accessed on 2017-05-23): https://arxiv.org/abs/1505.00996.

[HST10] Kaiming He, Jian Sun, and Xiaoou Tang. Guided image filtering. In *Proceedings of the 11th European Conference on Computer Vision: Part I*, pages 1–14. Springer-Verlag, 2010.

[HST13] Kaiming He, Jian Sun, and Xiaoou Tang. Guided image filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, 2013.

[ICG86] David Immel, Michael Cohen, and Donald Greenberg. A radiosity method for non-diffuse environments. *ACM SIGGRAPH Computer Graphics*, 20(4):133–142, 1986.

[IGN16] IGN. PlayStation VR vs PC Oculus Rift vs Vive comparison chart, 2016. Available (accessed on 2017-05-20): http://www.ign.com/wikis/playstation-4/PlayStation_VR_vs_PC_Oculus_Rift_vs_Vive_Comparison_Chart.

[Kaj86] James Kajiya. The rendering equation. *ACM SIGGRAPH Computer Graphics*, 20(4):143–150, 1986.

[KBS15] Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. A machine learning approach for filtering Monte Carlo noise. *ACM Transactions on Graphics*, 34(4), 2015.

[KS13] Nima Khademi Kalantari and Pradeep Sen. Removing the noise in Monte Carlo rendering with general image denoising algorithms. *Computer Graphics Forum*, 32(2), 2013.

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

[MCY14] Bochang Moon, Nathan Carr, and Sung-Eui Yoon. Adaptive rendering based on weighted local regression. *ACM Transactions on Graphics*, 33(5), 2014.

[PH10] Matt Pharr and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2nd edition, 2010.

[PSA+04] Georg Petschnigg, Richard Szeliski, Maneesh Agrawala, Michael Cohen, Hugues Hoppe, and Kentaro Toyama. Digital photography with flash and no-flash image pairs. *ACM Transactions on Graphics*, 23(3):664–672, 2004.

[PSK+16] Anjul Patney, Marco Salvi, Joohwan Kim, Anton Kaplanyan, Chris Wyman, Nir Benty, David Luebke, and Aaron Lefohn. Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics*, 35(6), 2016.

[Red01] Martin Reddy. Perceptually optimized 3D graphics. *IEEE Computer Graphics and Applications*, 21(5):68–75, 2001.

[RKZ11] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive sampling and reconstruction using greedy error minimization. *ACM Transactions on Graphics*, 30(6), 2011.

[RKZ12] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive rendering with non-local means filtering. *ACM Transactions on Graphics*, 31(6), 2012.

[Smi97] Steven Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, 2nd edition, 1997.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. *ACM SIGGRAPH Computer Graphics*, 24(4), 1990.

[TM98] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, 1998.

[ZJL+15] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and Sung-Eui Yoon. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum*, 34(2), 2015.