École polytechnique fédérale de Lausanne

Master's Thesis

Spatially-varying specular microstructures and reflectance filtering in a production renderer

Author:
Sebastien Speierer

Supervisor:
Wenzel Jakob (EPFL)
Andrea Weidlich (Weta Digital)

A thesis submitted in fulfillment of the requirements for the degree of Master's Degree

in the

Realistic Graphics Lab
School of Computer and Communication Science

in collaboration with

Weta Digital

August 6, 2018

Chemin des Colombettes 1, 1740 Neyruz, Switzerland

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Abstract

Communication Systems
School of Computer and Communication Science

Master's Degree

Spatially-varying specular microstructures and reflectance filtering in a production renderer

by Sebastien SPEIERER

Many surfaces of interest to computer-generated visual effects include spatially-varying specular microstructures or glittery effects. Such surfaces include rocks, snow, skin, as well as a wide range of manufactured materials. As a result, finding an appearance model capturing those behaviours is of particular interest to the visual effects industry. To be successful, this model needs to be computationally efficient, consistent across scales, and expressive (i.e., allow enough control to an artist in order to achieve a desired appearance). Existing models currently only satisfy two out of those three goals. Yan et al., 2016 achieves perfect consistency and expressiveness via the use of high-resolution textures, but at a computational cost that is prohibitive to visual effects production. Atanasov and Koylazov, 2016 presents a procedural technique that is computationally efficient while remaining consistent at all scales. However, it is limited to non-spatially-varying inputs, which significantly reduces its expressiveness and thus its usefulness in a production context. As a result, the main tools currently used in visual effects are general-purpose appearance models combined with texture filtering techniques such as Dupuy et al., 2013. These allow efficiency and expressiveness but result in a loss of consistency across scales when the surface normals exhibit anything other than smooth Gaussian statistics. This discrepancy is particularly evident with glittery surfaces, which are neither smooth nor Gaussian. Using a similar scheme, Tan et al., 2005 leverages the expressiveness of Gaussian Mixture Models to reduce the loss of consistency across scales, while introducing higher storage and filtering cost. The goal of this project is to investigate appearance and filtering models that would satisfy all three constraints of efficiency, consistency and expressiveness in a production renderer.

Contents

Chapter 1

Introduction to appearance modeling and texture filtering

1.1 Appearance modeling

As a major topic of computer graphics, rendering is the process of computing photo realistic images from a description of a 3D scene. Rendering finds applications in the movie creation, computer games, simulation, architecture, and many others. We often distinguish between real-time and offline rendering. Real-time rendering focuses on performance to achieve interactivity, trading realism and scene complexity for efficiency. This type of rendering mainly rely on the use of dedicated hardware like Graphics Processing Units (GPUs) and mostly uses a rendering algorithm called rasterization. On the other hand, offline rendering is a slow and computationally intensive process, and is typically used in the movie industry. With offline rendering, there is no limitation to the level of detail, complexity of the 3D models and scale of the scenes we can render. Visual effects companies leverage the computationally power of large clusters, often called *render farms*, to execute the rendering tasks on many cores in parallel. However, even with state of the art hardware and vast resources, render times can vary from a few minutes to hours and sometimes even days. To give a order of magnitude, the movie "Life of Pi", Academy Award winner for Best Visual Effects in 2012, containing many complex computer generated characters with fur and high level of detail, used around 1500 years of processor time. Considering parallelizing this process on a thousand cores, it would still take over a year to render.

In the more recent years, researchers in computer graphics worked on including physical correctness into the rendering process, which yields a new era in the field called physically-based rendering. One solution to this problem is to simulate how light flows in the real world. Emitted by light sources like the sun or a table lamp, light travels through space. After interacting with objects like a concrete wall or the skin of a character, the light rays loose energy and bounce further. A fraction of the light emitted will eventually reach the observer, which could be a camera or a human eyes for instance. An algorithm called *path tracing* stochastically constructs light paths through the scene trying to connect the sensor to the different light sources, hitting different objects in the scene.

Mathematical models can describe every step of this process, from the light emission, the light interaction with surfaces, to the light accumulation on a sensor going through complex lenses. Those models can then be combined into a single equation, the **rendering equation**. This equation exists under many forms but here is its most common formulation:

$$L_o(\boldsymbol{p}, \boldsymbol{w}_o) = L_e(\boldsymbol{p}, \boldsymbol{w}_o) + \int_\Omega f(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i) L_d(\boldsymbol{p}, \boldsymbol{w}_i) |\cos(\boldsymbol{w}_i)| d\boldsymbol{w}_i$$

with

- $L_o(\boldsymbol{p}, \boldsymbol{w}_o)$ the exiting radiance at $\boldsymbol{p}$ in the direction $\boldsymbol{w}_o$
- $L_e(\boldsymbol{p}, \boldsymbol{w}_o)$ the light emitted at the location $\boldsymbol{p}$ in the direction w_o
- $f(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i)$ the fraction of light scattered at $\boldsymbol{p}$ with a incoming direction $\boldsymbol{w}_i$ and outgoing direction $\boldsymbol{w}_o$
- $L_d(\boldsymbol{p}, \boldsymbol{w}_i)$ the light hitting $\boldsymbol{p}$ coming from direction $\boldsymbol{w}_i$

Intuitively, it enforces the exiting radiance at $\boldsymbol{p}$ to be equal to the emitted radiance at $\boldsymbol{p}$ plus the fraction of incident radiance scattered by the surface.

In the context of this work, we will focus on the material response which is represented by the $f(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i)$ term in the rendering equation. This is the **Bidirectional Scattering Distribution Function (BSDF)** and it describes how light scatters from a surface. One of the simplest BSDF model is the perfectly specular, mirror-like BSDF. A mirror surface reflects light only when the surface normal $\boldsymbol{n_p}$ (**macro normal**) is equal to the **half vector** bisecting the angle between the incoming light and the viewer. Given the incoming light direction $\boldsymbol{w}_i$ and the viewer direction $\boldsymbol{w}_o$, we can compute the half vector $\boldsymbol{h}$ in the following way:

$$\boldsymbol{h} = \frac{\boldsymbol{w}_o + \boldsymbol{w}_i}{||\boldsymbol{w}_o + \boldsymbol{w}_i||}$$

Therefore, the mirror-like BSDF model can be written as

$$f(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i) = \delta(\boldsymbol{n_p} - \boldsymbol{h})$$

On the other hand, a diffuse surface (also called Lambertian) scatters light in all directions. The amount light scattered in a specific direction is directly proportional to the cosine of the angle between the direction of incident light and the surface macro normal, as it obeys the *Lambert's cosine law*. Here is the BSDF equation of the Lambertian model:

$$f(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i) = \cos(\boldsymbol{n_p} \cdot \boldsymbol{w}_i)$$

Microfacet theory

In reality, diffuse and mirror-like surface models are far from enough to describe the whole spectrum of materials that compose our the real world. Microfacet theory was introduced by Torrance and Sparrow, 1967 as a physically plausible model of specular reflectance of different materials and will be used in Computer Graphics to represent more sophisticated materials. Walter et al., 2007 introduced a BSDF model for rough surfaces, handling reflection and refraction together, with the following equation:

$$f(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i) = \frac{F(\boldsymbol{w}_o, \boldsymbol{w}_i) G(\boldsymbol{w}_o, \boldsymbol{w}_i) D(\boldsymbol{h})}{4(\boldsymbol{n_p} \cdot \boldsymbol{w}_o)(\boldsymbol{n_p} \cdot \boldsymbol{w}_i)} \tag{1.1}$$

where F denotes the Fresnel reflection coefficient, D the microfacet distribution, and G the shadow-masking term. Microfacet theory assumes that surfaces are composed of a large amount of tiny microfacets, each of which is a perfect specular mirror with its own normal vector. Those normals (**micro normals**) are distributed according a **Normal Distribution Function (NDF)** (also called Microfacet Distribution Function) which corresponds to the $D(\boldsymbol{h})$ term in Equation 1.1. This function determines the overall roughness of the surface, and is most responsible for the size and shape of the specular highlight. As properly stated in Heitz, 2014, in order to construct an energy conservative model, the NDF needs to be normalized such that

$$\int_{\Omega} D(\boldsymbol{h})(\boldsymbol{n_p} \cdot \boldsymbol{h}) dw = 1 \tag{1.2}$$

In practice, the most commonly used Normal Distribution Functions are the Beckmann distribution and the Trowbridge-Reitz distribution:

- The **Beckmann distribution** is a physically-based microfacet distribution introduced by Katzin, 1964.

  $$D_{\text{Beckmann}}(\boldsymbol{h}) = \frac{1}{\pi\alpha^2(\boldsymbol{n_p} \cdot \boldsymbol{h})} \exp\left(\frac{(\boldsymbol{n_p} \cdot \boldsymbol{h})^2 - 1}{\alpha^2(\boldsymbol{n_p} \cdot \boldsymbol{h})^2}\right)$$

  where α is the *roughness* parameter.

- The **Trowbridge-Reitz** distribution has a sharper peak and a larger tail than the Beckmann distribution, which artists often find more suitable for modeling realistic specular surfaces like metal.

  $$D_{\text{Trowbridge}}(\boldsymbol{h}) = \frac{\alpha^2}{\pi((\boldsymbol{n_p} \cdot \boldsymbol{h})^2(\alpha^2 - 1) + 1)^2}$$

Less important in this project, the **shadowing-masking** term $G(\boldsymbol{w_o}, \boldsymbol{w_i})$ in 1.1 describes the fraction of microfacets visible in both directions $\boldsymbol{w_o}$ and $\boldsymbol{w_i}$. This function has a greater effect near grazing angles and for really rough surfaces. It is also needed to keep the BSDF model energy conservative as described in Heitz, 2014.

1.2 Texture filtering

While Microfacet Theory tries to account for microscopic geometric details, larger scale features results in variation in the surface properties on the shape manifold. Surface structures like scratches on brushed metal or pores on human skin are often too fine to be represented on the geometry but can be simulated by spatially varying the orientation of the macro normal. This can be achieved using a technique called **normal mapping**, where local normal information is stored in a 3 channels RGB texture which will be projected on the surface during rendering. As for any texturing techniques, aliasing artifacts might occur since textures are sources of high-frequency variation in the final image. Anti-aliasing methods like filtering adjust the frequency content of the texture based on the rate at which the texture is being sampled. Aliasing is a well-known problem from *Sampling Theory* and *Signal Processing*, and we know that the result of the ideal texture sampling process has to be band-limited such that content frequencies beyond the Nyquist limit are removed. We also know that this can be achieved by convolving the texture signal with a filter kernel (sinc, Gaussian, ...). Figure 1.1 illustrates aliasing artifacts in a rendering of a checkerboard texture on a plane.

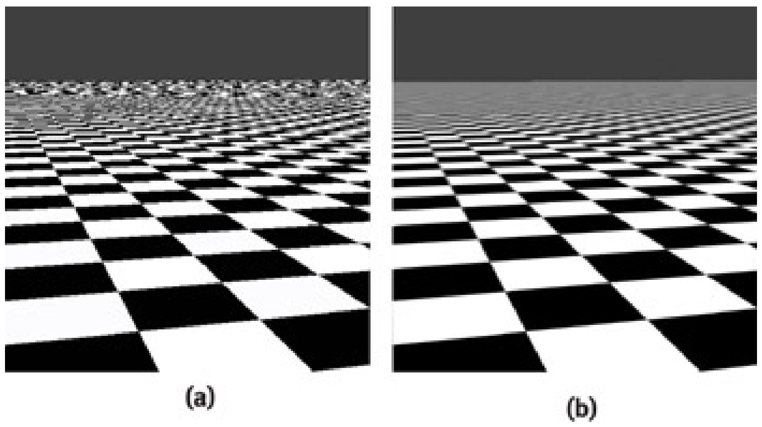


FIGURE 1.1: Checkerboard texture with (a) and without (b) aliasing artifacts (Figure from *GPU Gems*)

In rendering, **texture filtering** is the method that handle aliasing and determine the value of a texture look-up using the neighboring texels around the query location. Texture filtering methods received a lot of attention in computer graphics, being an essential part of the rendering process. Nowadays, most common graphical application leverage dedicated hardware for higher performance, optimizing memory access of the texture. The **filter region** is the projection of the screen space region covered by a pixel in texture space. Most method approximate this projected square with a parallelogram (a center and 2 vectors), or an ellipse. The size and shape of the filter region will depend on the distance between the viewer and the texture surface, as well as the orientation of the textured surface in respect to the camera orientation. Based on the size filter region, we can distinguish two types of texture filtering:

- **Magnification**: If the distance is short enough, the filter region will be smaller than a texel. In this case, texture filtering will magnify the texture and interpolate the texture content, avoiding *blocking artifacts*.
- **Minification**: If the texture is further away, the filter region will then cover many texels, which might cause aliasing because of the higher frequency of the texture content. The appropriate value will be define by texture filtering, preventing aliasing.

Over the years, researchers suggested many different filtering methods, and here are the most common ones:

- The **nearest-neighbor filtering** method isn't a filtering method properly speaking since it doesn't alter the texture content at all. For a given pixel center, it uses the value of the closest texel on the texture. It results in "blocking" artifacts when the screen pixel is smaller than a texel and aliasing when larger.
- **Bilinear filtering** performs a bilinear interpolation of the 4 neighboring texel around the projected screen pixel center. The weights used for the interpolation are defined by the distance of projected screen pixel center to the center of the different texels.
- **Anisotropic filtering** applies a different amount of filtering along the different axes which prevents over filtering at grazing viewing angle.

- **Elliptically weighted average (EWA) filtering** compensates aliasing artifacts due to perspective projection. It calculates an ellipse in texture space and performs a convolution of this ellipse with a filter kernel (often a Gaussian filter). It is a much more flexible filter, having different filter extends in different directions, improving the results quality by properly adapting to different sampling rates along the different axes.

Mipmapping

Texture filtering becomes expensive when the filter region covers many texels. For that reason, we can use a technique called **mipmapping** that pre-filters the texture in order to reduce texture I/O. The pre-filtered data will be stored in a **mipmap texture**, which is a pre-computed sequence of images (called mipmap level) representing a texture at different resolutions and was originally introduced by Williams, 1983. Every image of the sequence has half the resolution of the previous one, starting at the original resolution. Therefore, a mipmap on disk takes is a third bigger than the original texture:

$$\sum_{n=1}^{\infty} \frac{1}{4^n} = \frac{1}{3}$$

A texture look up decides which mipmap level to access based on the width of the filter region. It will only access texels of this pre-filtered image, rather than all the texels covered by the filter region. Then the filter method will only be applied on those pre-filtered texels to compute the final value. In order to get a smooth transition between the mipmap levels, one can use **trilinear filtering** which performs a linear interpolation between the results of bilinear filtering applied on two different mipmap levels. Similarly, the same linear interpolation between the mipmap level results can be applied with any other filtering methods.

Reflectance filtering

Unfortunately for us, shading a surface with linearly filtered normals does not result in the proper reflectance filtering. Consider the V-groove surface on Figure 1.2 with two normals pointing in different directions. Assuming the filter region covers the whole V-groove, the filtering operation will return the average of both normal. This means that the surface will appear totally flat at some distance due to filtering, changing drastically the appearance of the object. For this reason, normal map texture cannot rely on methods like mipmapping and/or standard texture filtering techniques.

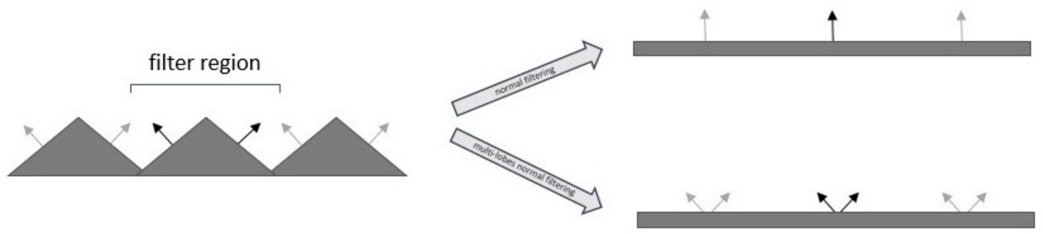


FIGURE 1.2: Reflectance filtering can drastically impact the appearance of the surface since normals cannot be linearly filtered

One solution to this problem is to super sample the texture using a naive Monte Carlo approach and attempt to compute the reflectance over the filter region by taking more samples. However, this performs poorly when dealing with specular material under high frequency lighting (like a sunny day). The specular energy will be contained in a tiny region of the filter region, where the normals satisfy the half-vector constraint, resulting in very high variance in this Monte Carlo integration. The amount of samples needed to resolve renders in those conditions if far beyond the budget production renders have in the normal course. Another solution is to use more complex models that can represent multiple lobes at the same time. As shown on Figure 1.2, a multi-lobe reflectance filtering technique could preserve the statistics of the normal from the original unfiltered texture.

1.3 Previous work

Reflectance filtering

Olano and Baker, 2010 introduced Linear Efficient Antialiased Normal (LEAN) Mapping, a method for real-time filtering of specular highlights in normal maps. Their method stores normal distribution information in a linearly-filterable form compatible with standard filtering methods. They do so using the fact that the anisotropic Beckmann distribution is a 2D Gaussian distribution projected onto a plane one unit above the surface, and that 2D Gaussians can be linearly combined when working with their moments only. They store the Gaussian first and second moments in two different mipmap textures, that will be used during rendering to compute the appropriate NDF. In a similar fashion, Christophe Hery, 2014 introduced a techniques for efficiently calculating the effect of bump-maps on surface roughness. Dupuy et al., 2013 introduced Linear Efficient Antialiased Displacement and Reflectance (LEADR), which extends LEAN Mapping to displacement maps. Their work also introduces a new physically-based microfacet model that includes shadowing and masking effects in their computations.

As seen with Figure 1.2, a single Gaussian lobe is often inadequate for modeling the more complex microfacet normal distributions. Tan et al., 2005 introduced a new framework using Gaussian Mixture Models (GMMs) for reflectance filtering. Their method precomputes the parameters of those Gaussian Mixture Models using Expectation-Maximization (EM) algorithm at different resolution and stores them in a mipmap texture. Their method ensures alignment of the GMM in neighboring texels such that the centers of the individual Gaussian elements in the GMM are closely located. This way they can interpolate the individual Gaussian elements using the same mathematical framework as LEAN. They do so by introducing an alignment penalty coefficient in the cost function of the EM algorithm so that their fitting algorithm automatically aligns the GMMs. Han et al., 2007 improves on the previous method by leveraging lighting-BSDF convolution. They introduce a new mathematical framework to perform reflectance filtering in the frequency domain. They fit Mises-Fisher distributions (spherical Gaussian like functions) rather than Gaussian distributions using a spherical Expectation-Maximization algorithm to improve accuracy and facilitate normalization. The use of spherical distribution prevents distortion of the hemisphere projection onto a plane, resulting in better accuracy at grazing angle.

While the previously introduced methods focus on efficiency, other methods praise accuracy and correctness. Yan et al., 2014 computes an accurate solution to the reflectance filtering integration by tessellating the normal map into fine triangular elements and analytically integrates over the filter region. Yan et al., 2016 proposes a more efficient method that fits a large four-dimensional Gaussian Mixture Model on the entire normal map. This parametric model has a closed-form solution for evaluating the normal distribution function for a given filter region. The paper introduces the notion of *position-normal distribution*, a four-dimensional function on the cross-product space of surface positions and normals. Mixing millions of 4D Gaussian, they can approximate this position-normal distribution function accurately. They also use an acceleration hierarchy to efficiently query the position-normal distribution and compute the underlying NDF for a given filter region. Yan et al., 2018 improves on the two previous method, using Gabor kernels rather than Gaussians. This way they are able to compute a wave optics reflection integral over the surface path, introducing color effects in the highlights.

Spatially-varying BSDF models

Other research papers focus on spatially-varying BSDF models that are not driven by textures. Those models often try to simulate a specific type of materials. For instance Raymond, Guennebaud, and Barla, 2016; Werner et al., 2017; Dong et al., 2015 focuses on materials exhibiting many tiny scratches like brushed metal.

Sparkling and glittering surfaces like snow, sand, skin or metallic paints were investigated by Jakob et al., 2014. Leveraging memory-efficient procedural algorithms, they introduced a stochastic hierarchy able to sample and evaluate discrete microfacet distributions. Their method treats a surface as a fixed, finite collection of microscopic facets. They use a stochastic hierarchy that defines a different specific set of flakes at each pixel. The key idea behind their algorithm is that the microscopic flakes are not stored in memory, but their count is computed by a deterministically seeded stochastic process during the traversal of the hierarchy at render time. By varying the density of flakes on the surface, we can change the appearance of glitters in the image.

Atanasov and Koylazov, 2016 improved on the previous method, introducing a model that does not require any precomputation and offers better overall performance. Their method first performs a query on the stochastic hierarchy to explicitly generate the normals of the flakes. The set of flakes can then be used as discrete NDF for the microfacet model. The normals of the flakes are sampled from a real microfacet distribution. Therefore, when increasing the density of the flakes within the filter region, the material appearance tends to the smooth continuous microfacet model. This allows their method to blend between both models, drastically reducing cost when the filter region gets large.

1.4 Solutions for a production renderer

This project was done in the context of an internship at Weta Digital, aiming at assessing existing current models for spatially-varying specular microstructures and investigating new solutions that would fit in their production renderer.

Production renderers are highly complex systems, often decoupled in different tools, libraries and plug-ins. When implementing new models in such frameworks, we need to be well-aware of the magnitude of scenes visual effects studios render nowadays. Please refer to ACM, 2018 for a deeper look at production rendering environment of the largest visual effect studios. Here is a non-exhaustive list of challenges imposed by the production environment at Weta Digital:

- Currently, final render frames have a resolution of 2K (2048x1080) or 4K (4096x2160). However, in order to represent fine details on a large assets, and handle the fact that for close-up shots, a fraction of this asset might take most of the camera viewport, the resolution of production textures vary between 4K (3840x2160) to 32K ($\sim$32000x16000). To give a order of magnitude, an uncompressed 32K texture with a single floating point channel represents 2 gigabytes of memory.
- A surface is often composed of a multitude of BSDF layers in order to simulate more realistic materials. For instance, artists can coat a gold surface with a lacquer layer and add another layer of dust at the top. Each one of those layers drive a different BSDF model, with its own set of spatially-varying parameters, normals, tangents, ... In practise, production shaders can combine up to 32 layers, which drastically impacts the performances of shaders evaluation and sampling.
- On top of this, complex scenes contains thousands of different assets, not counting instantiating methods. Production renders are already limited by the amount of memory of the cluster's nodes (in average 128Gbs). Therefore, we need to be careful about the additional memory cost of our solutions.
- Creativity is key in a production environment. It is crucial to make sure our new models do not reduce artists freedom and expressiveness. Also, it is important to come up with easy-to-use solution, finding the right set of parameters to expose to the users, and take away from the artist the burden of setting dozen of incomprehensible thresholds and other variables.
- Our solution needs to be as transparent as possible from a pipeline point of view, avoiding to complexify this already really sophisticated system.

Over the years, Weta Digital has developed its own physically-based production renderer, **Manuka**. As most of its competitors, Manuka highly relies on unidirectional path tracer and encompasses multiple importance sampling (MIS). However, unlike other physically-based renderer, Manuka's workflow is composed of a **pre-shading phase** before the **light transport phase** where the Monte Carlo integration is performed. This pre-shading phase finely tessellates the scene's geometry into micro polygons composed of **vertices**. More importantly, the shading graphs are evaluated and the material parameters are stored on a per vertex basis. The storage containing the vertices and their parameters is called the **grid**. Then, during the light transport phase, when a ray hits a surface, the parameters of the closest vertices are bilinearly interpolated and used to evaluate and sample the material models. The advantage of pre-shading (as oppose to *shade-on-hit*) is a better coherency in the texture look up and shaders evaluation, leveraging caching and parallel/vectorized computations. It also means that textures are not needed during the *light transport* phase, so texture storage and I/O can be properly optimized. For more details regarding Manuka's architecture, please refer to Fascione et al., 2018.

This special workflow adds another constraint to our solutions: textures can only be accessed during the pre-shading phase, and only the data stored on the grid is available during the light transport phase. We are also really limited with the amount of additional data we can store on the *grid*, memory already been an important bottleneck in this production environment. Currently Manuka supports dozens of BSDF models and uses the LEADR technique for reflectance filtering of displacement maps.

This document

This document is structured as follow:

- Chapter 2 presents a novel technique for reflectance filtering, inspired by the LEAN framework, using autoencoder neural networks.
- Chapter 3 describes our implementation of a discrete stochastic microfacet model in Manuka that can simulate glittering surfaces and random scratches.
- Chapter 4 gives an overview of our modified expectation-maximization algorithm for computing a mipmap of Gaussian Mixture Models on a normal map efficiently. This chapter also discusses our GMM simplification scheme, leveraging optimal transport theory for computing the adequate number of Gaussian elements in the mixture to achieve a target quality. We investigate different approaches for properly and efficiently combining Gaussian Mixture Models of different sizes in a texture filtering context. Finally, we propose an practical implementation of this method in Manuka that scale to production needs and improves on LEADR.
- Chapter 5 gives a conclusion to this thesis.

Chapter 2

Autoencoder for Reflectance Filtering

2.1 Overview

This chapter presents a novel technique for reflectance filtering, leveraging the power of autoencoder neural networks. Inspired by the LEAN mapping method introduced by Dupuy et al., 2013, our new method generates a texture of linearly filterable parameters that can be used to drive a microfacet model implemented as a neural network at render time. The key idea of this method is to train the encoder part of an autoencoder to learn how to project high dimensional NDF histogram onto a latent space with lower dimensionality. In other words, the encoder learns how to compress a slice of a normal map into a compressed representation (e.g. 16 floating point numbers). An important constraint on the latent space and the encoder will be to produce latent parameters compatible with texture filtering methods. Thus, we would be able to store those latent parameters in a mipmap texture and use standard filtering techniques to compute the set of parameters for a given filter region. The decoder part of the autoencoder could then be used as a parametric microfacet distribution model, emulating the normal distribution function for the original slice of the normal map given the filtered set of parameters. Figure 2.1 shows an overview of our new method.

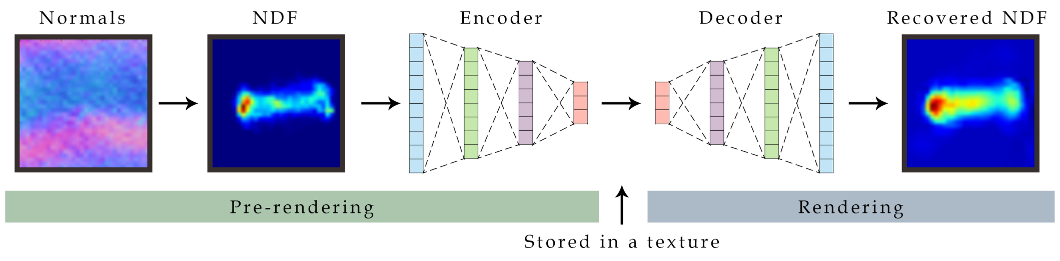


FIGURE 2.1: Autoencoder for reflectance filtering method overview

In this chapter, we first briefly introduce high-level concepts of machine learning to understand the autoencoder's architecture, before diving in the details of our method. We then present some results and discuss the usability of this new reflectance filtering scheme in a production context.

2.2 Training set

The training dataset is the set of examples used to train a model. In supervised learning, each element of the training set is a pair composed of an input data point and the corresponding target output. Using methods like stochastic gradient descent Kiefer and Wolfowitz, 1952, we are able to fit the parameters of the neural network model by feeding the training data points to the neural network and computing the gradient of a loss function comparing the target output and the output of the network. Ideally, we want to build a training set as representative as possible of the input the network will encounter in production. For this we selected a set of 24 normal map textures exhibiting different patterns, scales and surface features like metal scratches or ocean waves as shown on Figure 2.2.

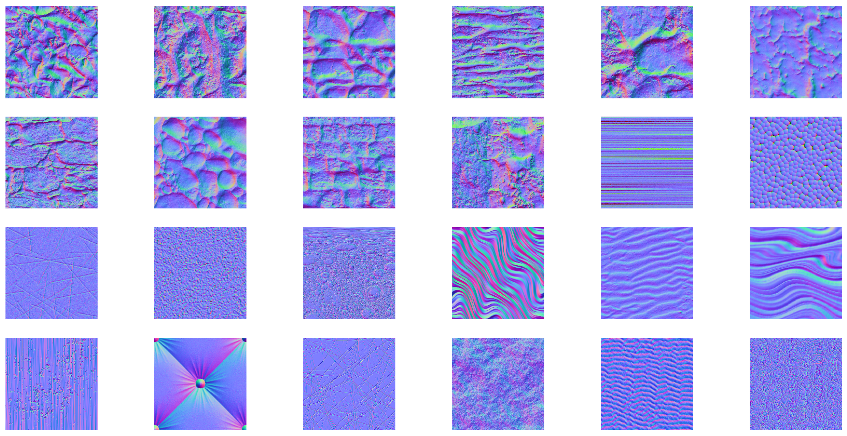


FIGURE 2.2: Training set composed of 24 normal map textures

As discussed in chapter 1, we are interested in computing the normal distribution function of a surface under the filter region for a given normal map. Therefore our training set will consist of pairs of slices of the normal map and their corresponding NDF histogram. This can be done using a binning method where we finely sample the normal map to compute the corresponding NDF histogram. In fact, it is more efficient to directly pass the NDF histogram input data to the autoencoder rather than the slice of normals itself. This way the network doesn't need to learn about the relationship between normals and NDF, and will focus on understanding and compressing NDF histograms.

2.2.1 Normal map classification

To gain in training efficiency and accuracy of the NDF reconstruction, we could restrict the training dataset to a specific type of normal map texture (brushed metal, ocean surface, ...) and train different *specialized* neural networks on the different classes of textures. One could even train and use a classifier to define the classes of textures and target the adequate *specialized* autoencoder for a given slice of normal map. We ran a few experiments to get a better understanding of the feasibility of the normal map classification. Principal Component Analysis (PCA) algorithm Jolliffe, 1986 can be used to project high-dimensional NDF histograms in 2D space. We can generate a cloud points, sampling normal slices on the training set and computing a 2D point by projecting their NDF histogram with PCA. Figure 2.3 is the result of this experiment.

We can observe clusters of points from the same texture on Figure 2.3 which might indicate that some more sophisticated clustering algorithm might be able to define distinct classes of NDF histogram. Note that this is a really naive way of clustering slices of normal map, but it give us some understanding on the feasibility of the problem. Also, the normal maps used in this experiment are really uniform,

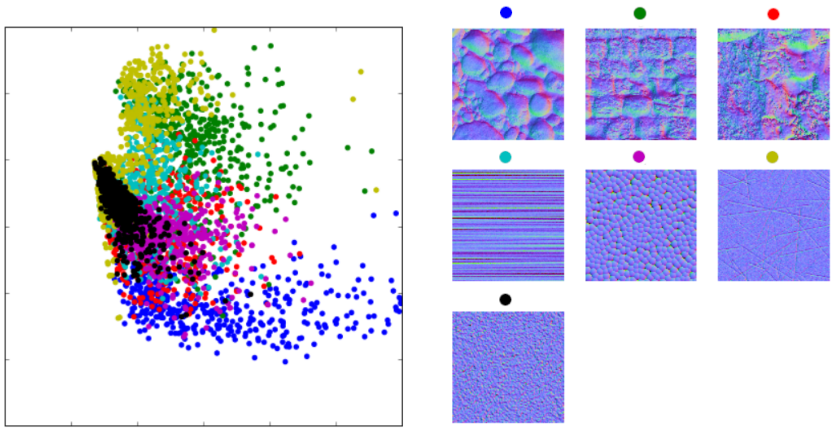


FIGURE 2.3: Normal map slices clustering experiments with PCA

repeating the same pattern across the whole texture. Unfortunately this is rarely the case in practise. Production textures will often contain a mix of different patterns, which will make the classification problem much harder. Additional experiments should be ran to fully understand the complexity of this problem. For this project, we are interested in a more general solution, but this could be part of a future project.

2.2.2 NDF histogram representation

In order to work with neural networks, we need to represent NDFs in terms of a finite set of values. While other methods use parametric models to represent an NDF, we decided to work with discrete NDF histograms.

A normal distribution function is a function $f : \Omega \to \mathbb{R}$ with Ω being the unit hemisphere. However a 2D parameterization of this hemisphere on a plane would be more adequate when working with neural networks. We investigated different types of parameterization for representing the NDF histograms in the context of this project:

- **Orthographic parameterization**:

$$\boldsymbol{h} \to \begin{cases} x = h_x \\ y = h_y \end{cases}$$

- **Spherical parameterization**:

$$\boldsymbol{h} \to \begin{cases} \phi = \frac{\arctan(h_y/h_x)}{2\pi} \\ \theta = \frac{2\arccos(h_h)}{\pi} \end{cases}$$

As we can see on Figure 2.4, the orthographic parameterization provides better locality for centered NDFs. It also introduces some distortion at grazing angle, which might cause some issue when trying to properly normalize the NDF produced by the decoder network. However, this distortion means that a greater part of the $[0,1]^2$ domain is dedicated around the upward normal vector. This will improve the method's accuracy since NDFs are often denser in this area. Note as well that this parameterization doesn't properly utilize the whole $[0,1]^2$, but only the unit disk. On the other side, the spherical parameterization fully utilize the $[0,1]^2$ region and doesn't introduce any distortion a grazing angle. However it distorts the upper

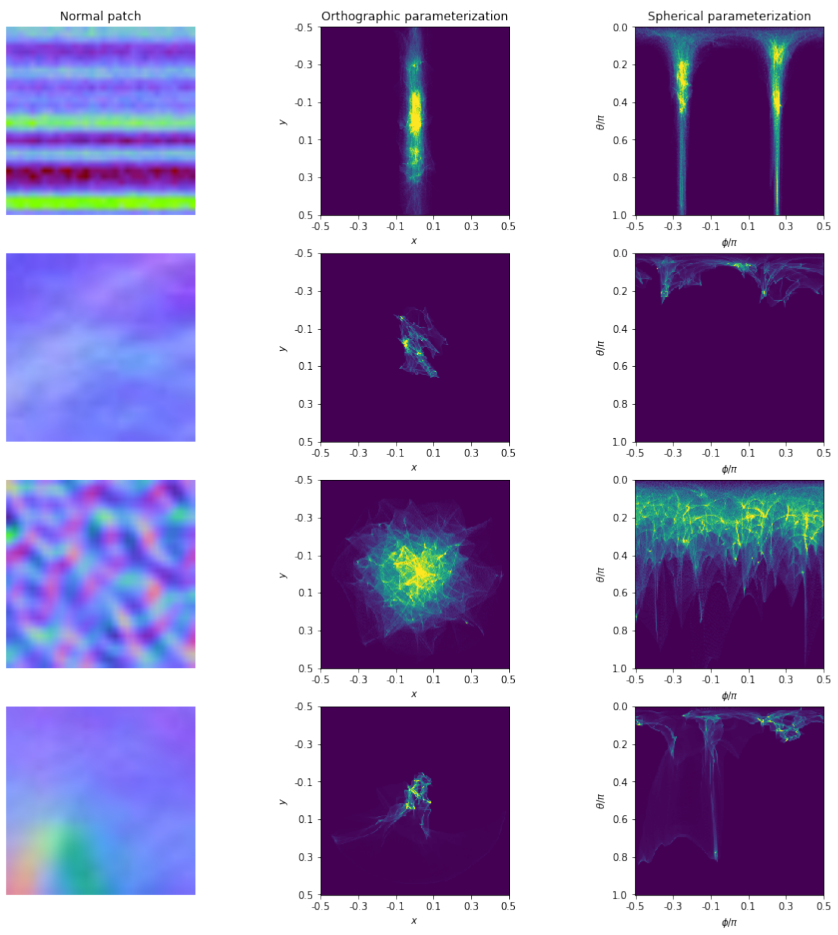


FIGURE 2.4: Comparing the two parameterizations on NDF histograms of different slices of normal map

region of the hemisphere, similar to the North and South poles being distorted on a world map using the Mercator projection. One can also independently vary the resolution of the ϕ and θ axes if needed which can be useful when working with different type of normal maps. However, this parameterization greatly loses in locality meaning that a convolution kernel applied on this parameterization corresponds to an anisotropic kernel shape on the unit hemisphere. Thus, the neural network might lack in understanding of the content's locality in the NDF it tries to reconstruct. In practice, the orthographic parameterization results in more efficient training and better NDF histogram reconstruction so this is the parameterization we will use in our method to represent NDF histograms on the $[0,1]^2$ square.

2.2.3 Data engineering

The size of the slices of normal maps defines the amount of data the NDF relates to, so it's complexity. Therefore it is important that our training dataset contains NDF histograms produced from slices of normal maps of varying size in order the capture the whole spectrum of possible NDF histograms. In practice, we use slices of size 2x2, 4x4 and 8x8 for our training. The filtering ability property of the latent space implies that we should be able to combine multiple of those slices' parameters to reconstruct an NDF histogram that would have originated from a larger slice of normal map. In order to enlarge our training set and prevent overfitting, we also randomly apply transformations to our NDF histograms, so to generate new data points in the training data set. We rotate the NDF histograms to make sure the network learn have to handle structures with different orientations. We also vary the magnitude of the normal map, which is the result of scaling the z component of the normals in order to simulate different depth of the surface features.

2.3 Network architecture

2.3.1 Autoencoders

Autoencoders were first introduced by Ballard, 1987 and their purpose is to learn a representation of the input data in a lower dimensional space. It consists of two neural networks trained simultaneously: the **encoder** network and the **decoder** network. The encoder compresses the input data into a short code, also called the latent parameters. On the other hand, the goal of the decoder is to uncompress that code to recover something that matches closely the input data. Autoencoders have been widely used for different applications: the latent parameters often provide a more intuitive representation of the input data, which can be used in the context of image recognition/classification. Also, by reducing the dimensionality during compression, the autoencoder tends to only preserve the main feature of the input data. This property gave birth to a variety of denoising algorithms using autoencoders.

Neural networks like the encoder and decoder of autoencoders are composed of a sequence of *neural layers*. Over the years, researchers invented many different types and variants of those layers. Let's briefly have a look at the most commonly used type of layers, that will be used to build our autoencoder later on:

- **Fully-connected layer**: Also called *dense layer*, the fully-connected layers can be seen as a large tensor (multidimensional matrix), attributing weights between input elements and output elements. The value of the output elements is computed as the weighted sum over the entire input matrix using the tensor's weight.
- **Convolutional layer**: Unlike fully-connected layers, convolutional layers only sums over a subset of the input elements, called the *convolution kernel*. It reduces the number of overall parameters, allowing the network to be deeper with fewer parameters.
- **Activation functions**: Another type of layer called **Activation layer** employs a non-linear function to transform the output of other layers. Inspired by biological neural network, activation functions allow the neural network to work on nontrivial problems using only a small number of nodes. For our neural networks we played with the following different activation functions:
  - **Sigmoid**:
  $$f(x) = \frac{1}{1+e^{-x}}$$
  - **Rectified linear unit (Relu)**:
  $$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

– **Parametric rectified linear unit (PRelu)**:

$$f(x, \alpha) = \begin{cases} \alpha x & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

2.3.2 Loss function

When working with neural networks, one needs to define a **loss function** that will be used to compute the gradient for the optimization algorithm. The choice of the loss functions plays a big role in producing optimal models and better convergence rate. The loss function is usually a function of the output of the network and the target output and computes their difference.

MSE and Chi-square

We have worked with different loss function for this method:

- **Mean Square Error (MSE)** is one of the most widely used loss function.

  $$MSE(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{N} \sum_i (y_i - x_i)^2$$

  The MSE computes the averaged square different between the target and the output's values.

- **Chi-square loss** is a well-studied tool in statistics used to evaluate how well a statistical model fits a data set.

  $$\chi^2(\boldsymbol{x}, \boldsymbol{y}) = \sum \frac{(y_i - x_i)^2}{x_i}$$

  where $\boldsymbol{x}$ is the reference histogram and $\boldsymbol{y}$ is the result from the autoencoder.

While the loss functions presented above are really easy to implement and can be evaluated efficiently, they only account for local difference in the NDF histograms. Tiny slices of a smooth normal map won't exhibit much variance in their normal distributions, and therefore the resulting NDF histograms will be really sharp. When most of the distribution is concentrated in a fine spike, a slight change in the location of that spike will results in high distance value when using MSE of Chi^2. For that reason, the resulting autoencoders tend to be more conservative and avoid returning sharp NDF histograms. However, from a perception point of view, a slight change in the spike location might be better than losing sharpness in our NDF histogram. Consequently, we have to find better metric to compute the loss when training our autoencoders.

Wasserstein distance

The **Wasserstein distance** is a distance function between probability distributions known in computer science as the **Earth mover's distance**. Assuming those distributions are piles of earth, this metric can be viewed as the amount of dirt needed to be moved times the distance it has to be moved in order to turn one pile into the other. This metric has a strong connection to the *Optimal Transport* problem. A transport plan between two distribution μ, ν is described by the function $\gamma(x, y)$ which gives the amount of dirt to move from x to y. It satisfies the following constraints:

$$\int \gamma(x', x)dx' = \nu(x)$$
$$\int \gamma(x, x')dx' = \mu(x)$$

The Wasserstien distance is defined as the minimal cost out of all possible transport plans:

$$C = \inf_{\gamma \in \Gamma(\mu,\nu)} \int c(x,y)d\gamma(x,y)$$

The **Sinkhorn-Knopp algorithm (SK)** Knight, 2008 is an iterative method for computing an optimal transport plan. Given two NDF histograms $\boldsymbol{p}$ and $\boldsymbol{q}$ (flatten arrays) and a distance matrix (in our case using Euclidean distance) $\boldsymbol{M}$ defined as follow:

$$M_{ij} = \exp(\frac{-(i-j)^2}{\lambda})$$

where λ is the regularization weight, the algorithm iteratively updates the weights of two transport plan vectors $\boldsymbol{u}$ and $\boldsymbol{v}$. $\boldsymbol{u}$ is the transport plan from $\boldsymbol{p}$ to $\boldsymbol{q}$ and $\boldsymbol{v}$ is the transport plan from $\boldsymbol{q}$ to $\boldsymbol{p}$. At each step, the algorithm updates the transport coefficient vectors in the following way:

$$\boldsymbol{u}_{i+1} = \boldsymbol{p}/(\boldsymbol{M}\boldsymbol{v}_i)$$
$$\boldsymbol{v}_{i+1} = \boldsymbol{q}/(\boldsymbol{M}^T\boldsymbol{u}_i)$$

After n iterations, the sub-optimal transport plan matrix $\boldsymbol{\Gamma}$ can be compute as follows

$$\boldsymbol{\Gamma} = (\boldsymbol{I}\boldsymbol{v}_n)\boldsymbol{M}(\boldsymbol{I}\boldsymbol{u}_n)$$

Given the final transport plan, the Wasserstein distance can be computed as

$$W = \sum_{x,y} \boldsymbol{\Gamma}_{x,y}(y-x)^2$$

The key advantage of using an iterative algorithm is that we can trade performance for accuracy by varying the number of iterations taken. In our method, a sub-optimal solution of the transport problem is enough therefore we only need to take a few iterations of the Sinkhorn-Knopp algorithm.

2.3.3 Training procedure and architecture

For both the encoder and the decoder, we ran experiments varying their composition in depth, type and number of layers. The training process was scripted to run different architectures simultaneously on different machines in order to find the optimal sequence of layers. Once the training and the evaluation of the resulting autoencoders finished, we compared the results and refined our hyper-parameters for the next generation of network architectures, based on the best autoencoders of the previous generations.

Encoder architecture

This automatized training process also allowed us to test our method's performance using different number of dimensions for the latent space. The performance of our method at render time is directly linked to the size of the latent parameters. A wider vector implies multiple texture queries and a deeper and more complex decoder network. Also, keeping *Manuka*'s architecture in mind, those latent parameters will have to be stored on the *grid* during the pre-shading phase. In order to minimize the memory impact of this method, we need to keep this latent parameter vector at a reasonable size.

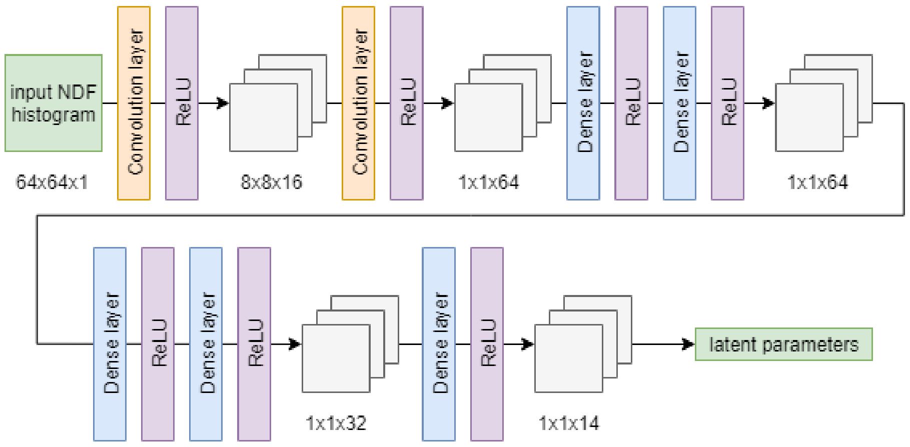


FIGURE 2.5: Encoder architecture with a 14-wide latent parameters

As shown on Figure 2.5, the encoder is exclusively composed of convolution, dense and ReLU layers. After a few convolutional layers reducing the dimensionality of the problem to solve, dense layers are applied to compute the latent parameters.

Decoder architecture

For this project, we experimented with two different types of decoder.

- The first one aimed at reconstructing the whole NDF histogram for a given set of latent parameters. This histogram could then be used to evaluate the PDF of a specific half vector using bilinear interpolation. The overhead of decoding the entire NDF histogram when only evaluating a couple half vector PDFs has a significant impact on the performance of this model. Moreover, the resolution of the reconstructed NDF histogram band limits the frequency of the NDF content. In other words, the resolution of the reconstructed NDF histogram defines the maximum sharpness of our method. In practice, the NDF histogram resolution needed to prevent blurring our materials was higher than acceptable.

- The second type of decoder takes an additional two parameters, representing the half vector we want to evaluate the PDF of. Those are the coordinates of the orthographic projection of the half vector on a plane and will be concatenated at the end of the latent parameter vector. As illustrated by Figure 2.6, this is then passed through the decoder network, which will return a single floating point number representing the PDF value of the passed half vector. Therefore this model only computes the PDF for a single half vector which is much more efficient than reconstructing the entire NDF histogram every time.

While the first type of decoder isn't efficient enough to be used in a production renderer and introduces some intrinsic roughness, in our experiments, it clearly out-performs the second type of decoder in terms of fidelity of the NDF histogram reconstruction. Given a simpler problem to solve, not having to make sense of the 2 additional parameters representing the half vector, this type of decoder doesn't need as much training to produce good quality results on the test set. However, we believe that the second type of decoder could reach similar results with a better understanding of the neural network domain and using modern neural network architectures. In the context of this project, we spend most of our time improving the results of the first type of decoder and elaborating the architecture of the second type of decoder. For this reason, the rest of this section will discuss the architecture and specificity of the second type of decoder while the Section 2.4 will only present results of the first type of decoder, which are more representative of the capability of this novel reflectance filtering technique.

As shown on Figure 2.6, the decoder is only composed of dense and PReLU layers. It first brings the input vector to a higher dimensional space (128 dimensions) and then brings it down a single value using a sequence of dense layers. In theory, by skipping the intermediate discrete representation (NDF histrogram) of the underlying NDF, this model can reconstruct NDF with features at any frequency, therefore doesn't introduce any intrinsic roughness.

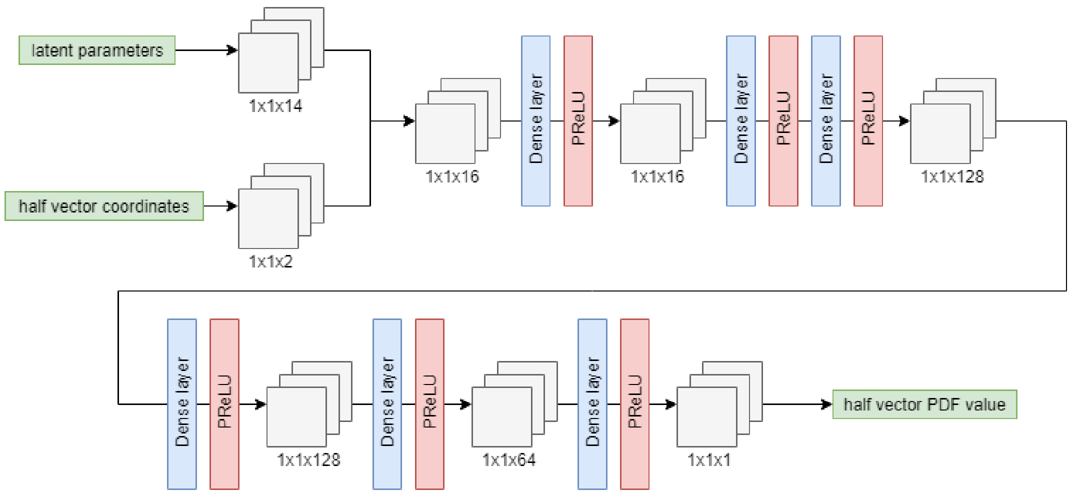


FIGURE 2.6: Decoder architecture with a 14-wide latent parameters

In practice, using latent parameter vectors of length 14 seems to give the best result. It is also ideal since once the two half vector coordinates are concatenated at the end of the vector, the length of the decoder input will be 16 floating point numbers. This way our implementation can fully leverage SSE instructions for optimal performance.

Texture filtering in latent space

In order to enforce the encoder to produce latent parameters that are compliant with standard linear filtering techniques, we sub-sample the normal map slices to produce 4 sub-slices that we will use to generate 4 sets of latent parameters as shown on Figure 2.7. Those parameters will be linearly combined and decoded. The resulting NDF histograms will be compared against the NDF histograms computed from the original slice of the normal map. On top of that, each individual sub latent parameter vectors will be decoded to reconstruct the sub NDF histograms. Those will also be compared to the original sub NDF histograms using the loss function and the gradient of the error will be back-propagated to update the weights of both the decoder and the encoder.

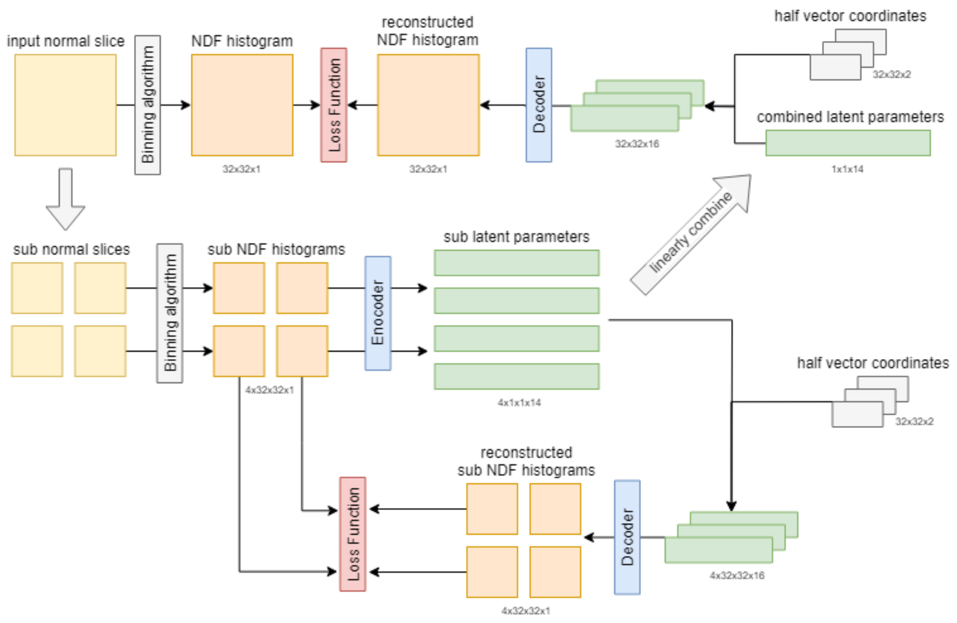


FIGURE 2.7: Training workflow using our autoencoder

In order for the decoder network to be used as a NDF in a renderer, the simulated distribution has to be normalized according to Equation 1.2. Similarly to the linear filtering property of the latent space, this could be enforced as a constraint on the autoencoder during the training of the network. Unfortunately, this normalization strategy wasn't sufficient in practice. Therefore, given that the texture of latent parameters is computed offline, we decided to spend a few extra CPU cycles to compute a normalization map that we will be added as an extra channel in the computed mipmap texture and used to normalize the PDF values at render time. We believe that this way of solving the normalization problem is sub-optimal and finding a better solution could be part of a future project.

2.4 Results

In this section, we present results and limitations of our method using the first type of decoder which reconstruct the whole NDF histogram. First of all, Figure 2.8 shows the reconstruction of NDFs using the latest generation of encoder/decoder after hours of training. As we can see, slightly different architectures produce different results. In overall, the accuracy of the reconstructed NDF is far from enough for being used in a production renderer. Although, our models seem to capture the main feature of the different NDFs in their respective latent parameters. Moreover, autoencoders are known in the deep learning community for blurring reconstruction results due to the reduction of dimensionality in the latent space. Even though there exists plenty of techniques to tackle this issue, we leave that to future work given the restricted time constraint on this project. Another major concern of our method is the fact that the decoder has to be evaluated at render time to simulate the NDF. Our prototype implementation in Pharr, Jakob, and Humphreys, 2016 leverage vectorization capability of modern CPUs to speedup the evaluation routine of the decoder network. However, given its complex architecture presented in Section 2.3, it is far too expensive to be used in a production environment.

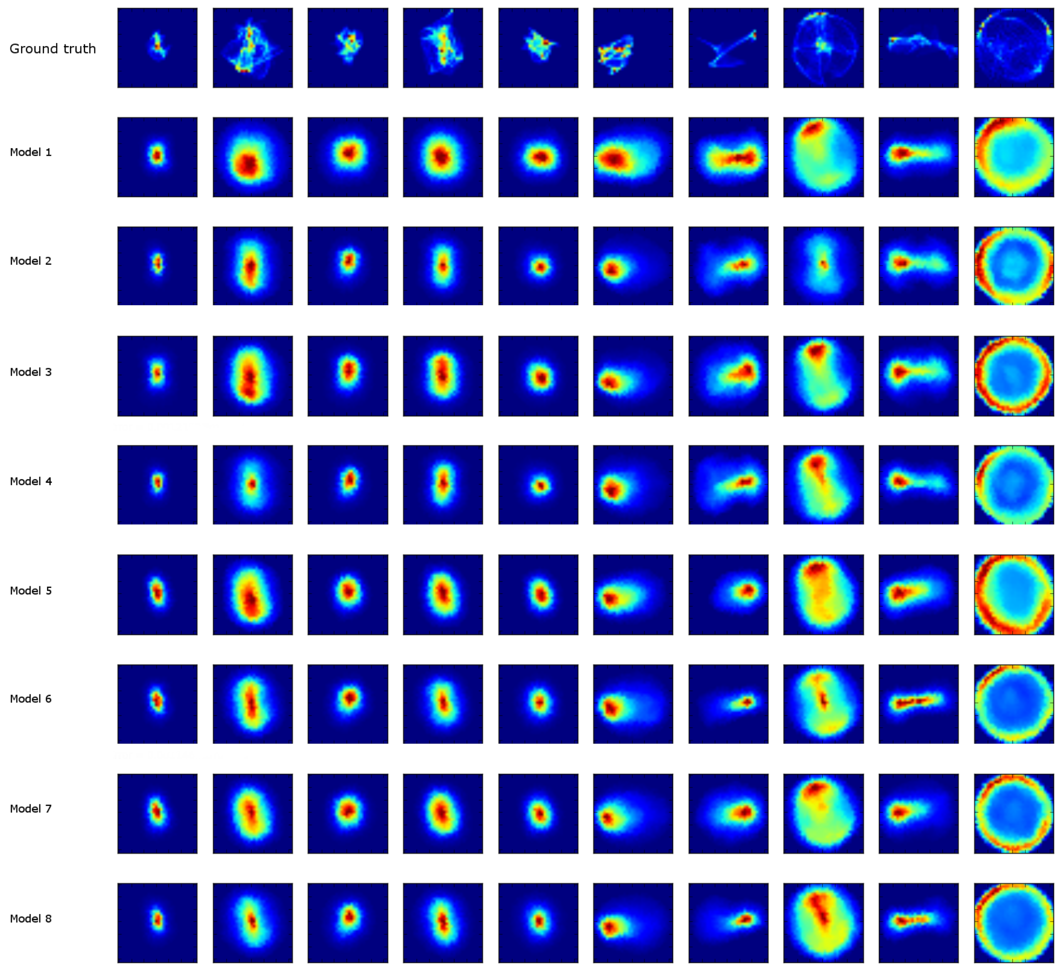


FIGURE 2.8: Results of the reconstruction of NDF histograms for different models of the same generation

More interestingly, Figure 2.9 demonstrates the linear "filterability" of the latent parameters produces by the encoder. The first row shows a slice of a normal map as well as four crops of this slice. We produce NDF histogram using the binning method, projection on the unit square using the orthographic parameterization. Then we compute the corresponding latent parameters using a trained encoder and reconstruct the NDFs passing those to the decoder, which are shown on the third row. Finally, we linearly combine the 4 set of latent parameters and use the decoder to reconstruct the NDF P, which is suppose to represent the normal distribution of the original slice of the normal map (A). By comparing F, K, and P, we see that results are quite promising, and the filtering of the latent parameters behaves properly. Finally, Figure 2.10 shows the stability of the filtering between 4 sets of latent parameters along two axis. The resulting NDFs smoothly interpolate between the 4 original NDF at 4 the corners.

While producing results of insufficient quality and having an expensive evaluation routine, we believe that our method could be improved running further experiments with state-of-the-art deep network architectures. Encouraging progress in neural network research is made every month, which let us hope that our promising

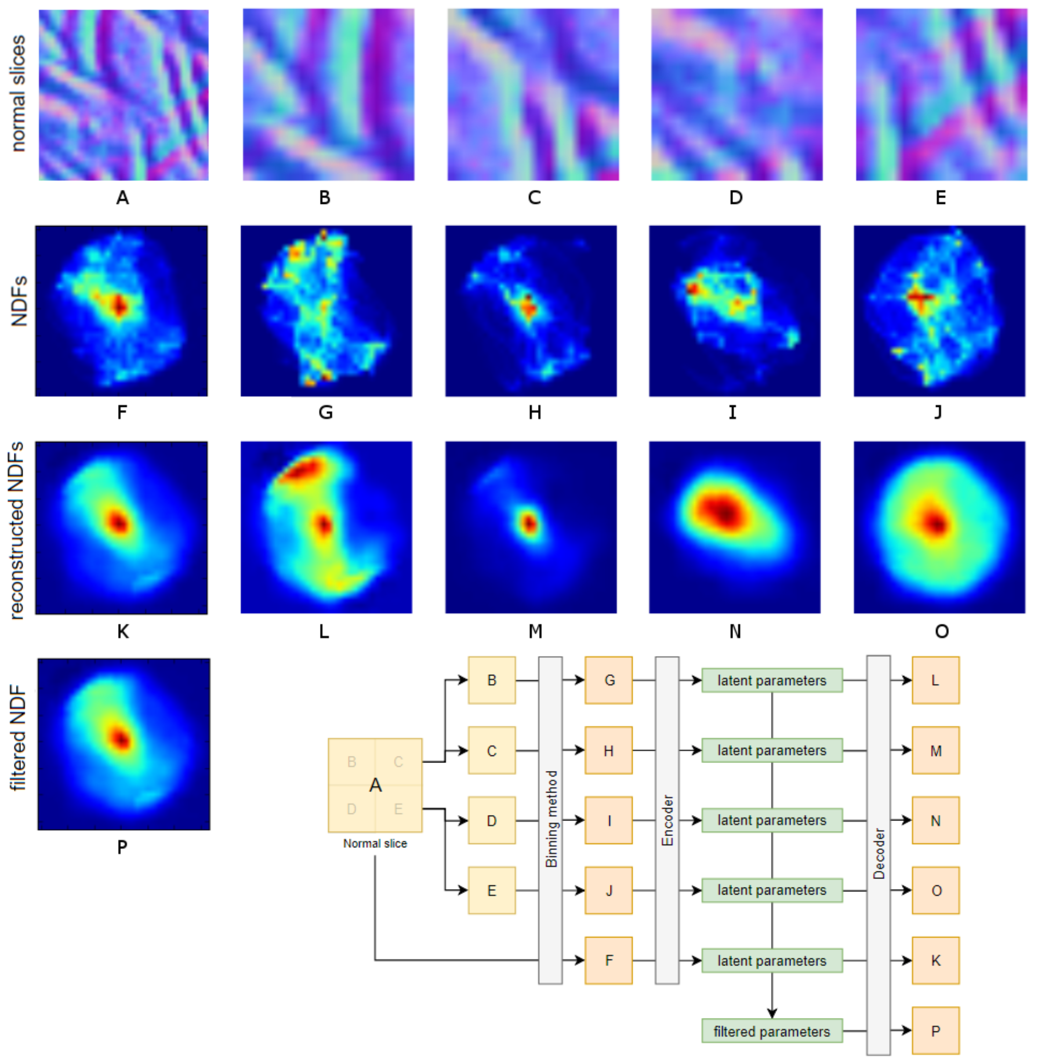


FIGURE 2.9: Latent parameters produced by the trained encoder can be linearly combined before passed to the decoder and still result in the proper NDF histogram

preliminary results will give birth to a usable framework for reflectance filtering in the near future.

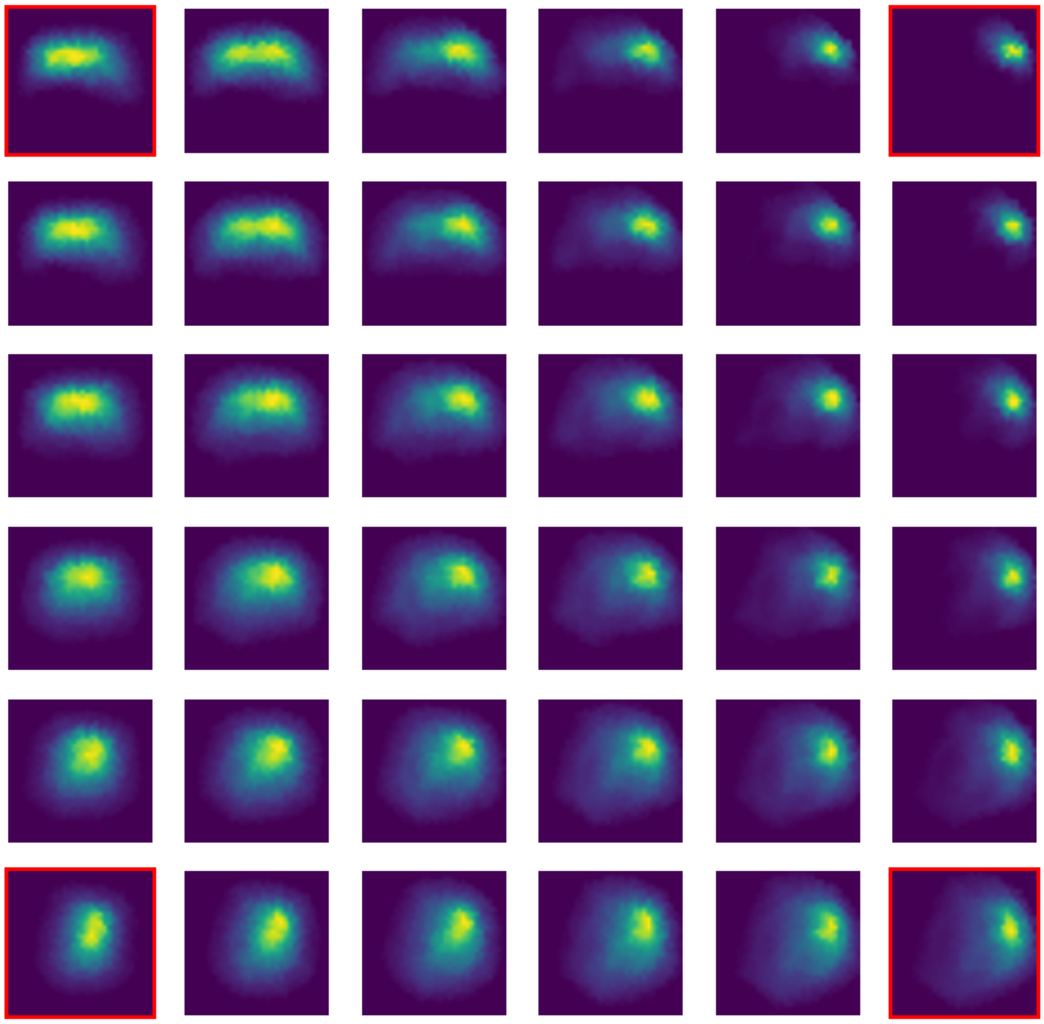


FIGURE 2.10: Interpolation of 4 sets of latent parameters and their reconstructed NDF histograms

Chapter 3

Procedural Microflakes Models

3.1 Introduction to multiscale BRDF

As described in section 1.3, other approaches seek to add the spatially-variant component to microfacet models without relying on external textured input. For instance, procedural generation techniques can be used to randomly disperse scratches or metallic paint flakes over a surface without the need of any storage. However this comes at a higher computational cost and makes it ambiguous onto how to handle filtering in this case. Once the filter region covers many of those micro structures, those need to be accounted for in the underlying microfacet model rather than being stochastically sampled. In this chapter we take a deeper look into the solution introduced by Atanasov and Koylazov, 2016 and Jakob et al., 2014 and present our own implementation in Manuka with its additional features and limitations.

The key idea behind both methods is to simulate the distribution of a finite number of flakes on the unit texture space. In order to evaluate their model, given a filter region, an incoming direction and an outgoing direction, both methods simply compute the fraction of flakes satisfying the half vector constraint, meaning the flakes whose normal is halfway between the incoming and outgoing direction. By varying the density of the flakes and their reflective property, those methods can be used to model a large range of materials.

Jakob et al., 2014 introduces the concept of **multiscale BRDF** which describes the average response of the microfacet BRDF over a finite area $\boldsymbol{A}$ and a finite solid angle $\boldsymbol{\Omega}$ around the incident direction $\boldsymbol{w}_i$. Mathematically, it can be defined as following:

$$\hat{f}(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i) = \frac{1}{a(\boldsymbol{A})\sigma(\boldsymbol{\Omega})} \int_A \int_\Omega f(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i) d\boldsymbol{w}_o d\boldsymbol{p} \tag{3.1}$$

with $a(\boldsymbol{A})$ the surface area of $\boldsymbol{A}$ and $\sigma(\boldsymbol{\Omega})$ the area of $\boldsymbol{\Omega}$.

He then introduces a discrete analogue of the continuous microfacet distribution which represent the fraction of flakes satisfying the location and angle constraints:

$$\hat{D}(\boldsymbol{h}) = \frac{1}{N} \sum_{n=1}^{N} I(\boldsymbol{p}_n \in \boldsymbol{A}) I(\boldsymbol{w}_n \in \boldsymbol{\Omega}) \tag{3.2}$$

Combining both equations 3.1 and 3.2, accounting for the change of variables and assuming that other terms of the microfacet BRDF equation 1.1 (like Fresnel, shadow-masking) are constant between different microfacets, we can derive a simpler formula for the multiscale BRDF:

$$\hat{f}(\boldsymbol{p}, \boldsymbol{w}_o, \boldsymbol{w}_i) = \frac{(\boldsymbol{w}_i \cdot \boldsymbol{h}) F(\boldsymbol{w}_o, \boldsymbol{w}_i) \hat{D}(\boldsymbol{h}) G(\boldsymbol{w}_o, \boldsymbol{w}_i)}{a(\boldsymbol{A}) \sigma(\boldsymbol{\Omega}) (\boldsymbol{n_p} \cdot \boldsymbol{w}_o)(\boldsymbol{n_p} \cdot \boldsymbol{w}_i)}$$

In practice the area A is defined by the filter region and $\boldsymbol{\Omega}$ is computed as the cone of radius γ centered around $\boldsymbol{w}_i$. Intuitively, the parameter γ can we seen as scaling factor on the light source radius. The higher the value of γ is, the smoother and less glinty the surface will appear.

3.2 Implementation

3.2.1 Stochastic hierarchy

The next challenge is to efficiently compute values of $\hat{D}(\boldsymbol{h})$ for a given filter region. For this Atanasov and Koylazov, 2016 and Jakob et al., 2014 introduced algorithms for generating the flakes count on the fly using a stochastic hierarchy. Both methods employ a similar approach, however Jakob et al., 2014's method requires expensive pre-computations which prevents it to scale to a production level of complexity. Therefore, for the rest of this chapter we will focus on Atanasov and Koylazov, 2016's approach.

The stochastic hierarchy in Atanasov and Koylazov, 2016 is implemented as a quad-tree where the root node contains all the flakes. Each node uniformly distributes its flakes between its four children. On top of that, every node consists of a 64-bits state, which will be used to assign a 64-bits state to each one of its 4 children using a four xor-shift-based generators like Code 3.2.1. None of that is stored in memory, except for root's total number of flakes and its 64-bits state. The rest of the nodes will be generated on the fly during the hierarchy traversal, keeping a very low memory profile for this method.

```
uint_64_t NextState(uint64_t x)
{
    x ^= x << 13;
    x ^= x >> 7;
    x ^= x << 17;
    return x;
}
```

LISTING 3.1: xor-shift-based generator code

3.2.2 Evaluation and sampling procedure

In order to define the number of flakes under the filter region and their orientations, we query the stochastic hierarchy by recursively computing intersection of the nodes and the filter region in a depth-first order, starting at the root. During the traversal, we maintain a list of all the flakes that intersect with the filter region, which will be processed later-on. A node is considered as a leaf node if it satisfies one the the two following conditions:

1. the number of flakes in the node is smaller than the threshold K
2. the depth of the node is higher than T

In practice, working with $(K, T) = (16, 15)$ seems to be adequate. When a leaf node is encountered, we use the 64-bits state of this node to generate the normal vectors $\boldsymbol{n}_j$

for all the flakes it contains. Those normals are sampled out of the user-defined underlying flake distribution (GGX or Beckmann). Note that if the node is not entirely contained within the filter region, we also need to generate the location of those flakes and add to the list only the ones that fall in the filter region. For all the flakes for which the reflection direction $\boldsymbol{r}_j$ is contained in the solid angle $\boldsymbol{\Omega}$, we compute a weight

$$w_j = \boldsymbol{n}_j \cdot \boldsymbol{r}_j$$

where $\boldsymbol{r}_j$ is the reflection direction given the flake's normal $\boldsymbol{n}_j$ and the incoming light direction. Note that the set of pairs $(\boldsymbol{r}_j, w_j)$ are cached such that they can be reused for the sampling procedure or importance sampling computations. By summing those weights, we can compute an accurate approximation of the microfacet distribution

$$\hat{D}(\boldsymbol{h}) \approx \frac{1}{N} \sum_{j:\boldsymbol{r}_j \in \boldsymbol{\Omega}} w_j$$

In order to sample a direction $\boldsymbol{i}$ out of this discrete microfacet distribution, we first stochastically sample one of the cached flakes proportionally to their weights. Then we uniformly sample a direction in the cone of radius γ around the sampled flake's reflection direction with probability

$$\frac{1}{\pi(1 - \cos(\gamma))}$$

Finally, since this direction could also have been sampling after picking another flake (with overlapping cones), we have to compute the true probability of sampling $\boldsymbol{i}$ out of the list flakes as

$$p(\boldsymbol{i}) = \frac{\sum_{j:\boldsymbol{i}\cdot\boldsymbol{r}_j \geq \cos(\gamma)} w_j}{\pi(1 - \cos(\gamma)) \sum_{j:\boldsymbol{i}\cdot\boldsymbol{r}_j \geq \cos(\gamma)} w_j}$$

Given the fact that flake's normals are drawn out of a standard microfacet distribution (GGX, Beckmann), it is expected that, at a far distance, when the number of flakes contained within the filter region is large enough (> 1000), the appearance of the material tends to the appearance of the same material using the standard microfacet distribution instead. Therefore, since evaluating the standard microfacet distribution is much cheaper, we can switch to the standard model when the amount of flakes is greater than a user-defined threshold. Moreover, to ensure smooth transition between the two models, we perform a progressive blending starting at a lower threshold (500 flakes) ending at the higher threshold (1000 flakes), where we evaluate both models and linearly interpolate the results.

3.2.3 Flakes control

Our implementation exposes different parameters to the user that change flake's properties in this discrete model. By playing with those parameters, the user will be able to generate a large spectrum of varying materials needed in production. For instance, we extend the describe model such that flakes can take different shapes and orientations. We also worked around the density assumption of Atanasov and Koylazov, 2016's method such that flakes can be greater than a pixel and sparsely distributed. Finally, for better integration in Manuka's layering system, we define different reflection/transmission modes that will be triggered when no flakes are found under the filter region.

3.3 Results

We implemented the discrete microfacet model presented above in *Manuka*, extending the already existing large palette of BRDF models in Weta's pipeline. This new set of BRDFs are particularly useful given the pre-shading architecture of *Manuka* since it is not possible to evaluate procedural functions at render time like other renderers would do when evaluating the shading graph for a hitting point. Figure 3.1 demonstrates the blending between our discrete microfacet model and the underlying standard microfacet model (here GGX). The pig at the back is fully rendered using GGX while the pig at the front uses the discrete model. Figure 3.2 shows different appearances achieved varying the flake density and their size.

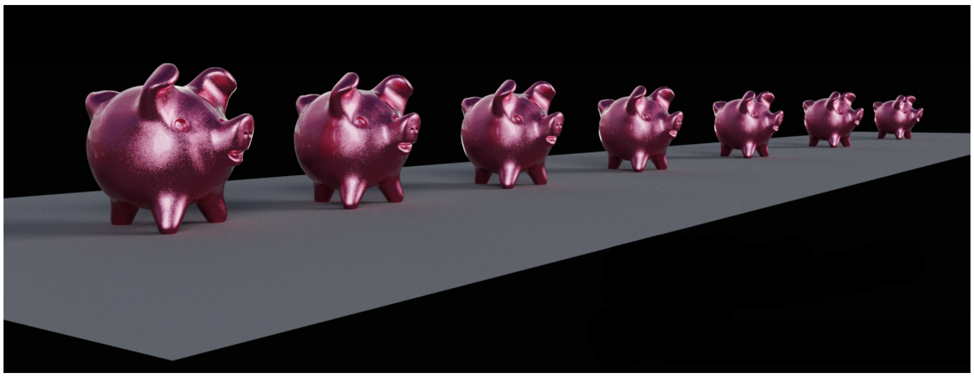


FIGURE 3.1: The procedural flakes model smoothly transition to the underlying microfacet distribution

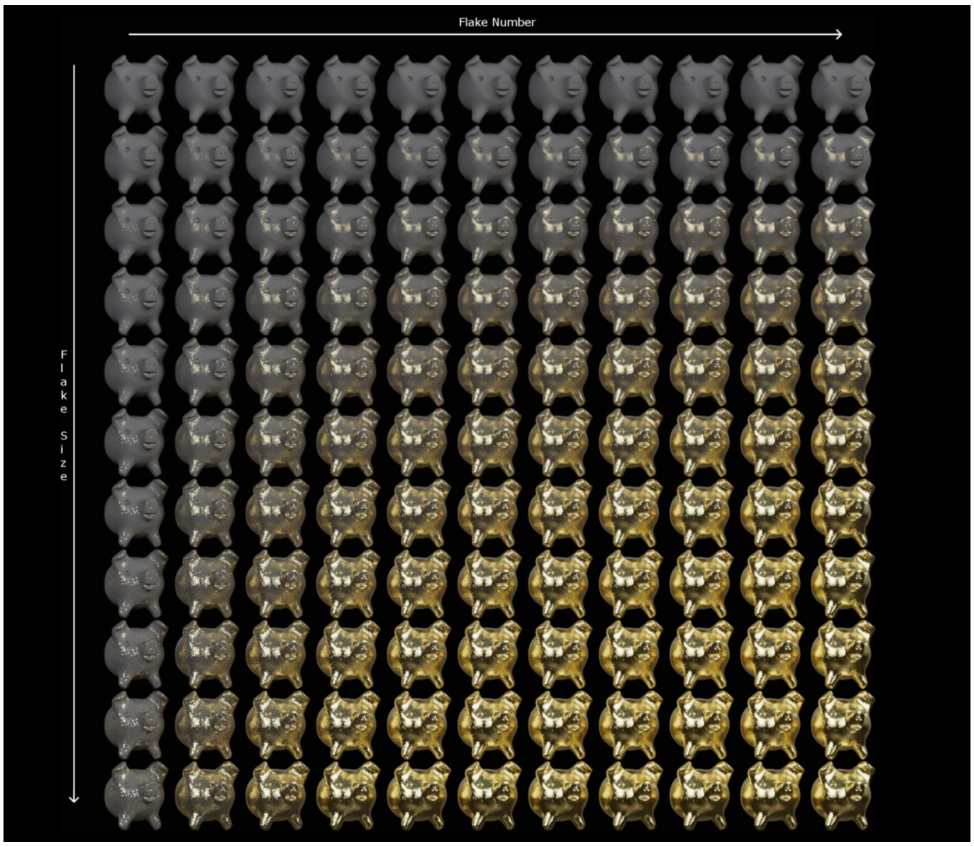


FIGURE 3.2: Different look varying the flakes size and density

Chapter 4

Texture-based reflectance filtering with Gaussian Mixture Models

4.1 Method overview

From an artistic standpoint, the procedural techniques introduced in Chapter 3 lack of control and expressiveness. Despite the parameters controlling the flake's density and shape, the user cannot adjust the location of the procedural flakes. In this chapter, we introduce a method that tackles this problem by giving control over the spatially-varying features of the material using texture mapping as in Chapter 2. As discussed in Section 1.3, already many texture-driven reflectance filtering methods exists with their respective set of limitations and constraints. Our method is greatly inspired by the work of Tan et al., 2005 which combines the power of Gaussian Mixture Models and the texture mipmapping technique. One can see it as a extension of the LEAN method (Olano and Baker, 2010) or LEADR (Dupuy et al., 2013), which is really convenient for us since those two are already implemented in the current rendering pipeline at Weta Digital.

We compute a mipmap of Gaussian Mixture Models such that every texel of each mipmap level best fit the Normal Distribution Function represented by that texel. With production texture's resolution reaching 32K, using a standard Expectation Maximization algorithm would be to slow to fit all the parameters of the Gaussian Mixture Models in the mipmap. For instance, texels at the coarsest levels of the mipmap would have to fit parameters of their Gaussian Mixture Model on input sets of approximately half a billion normals. Therefore we introduce in Section 4.4 a reformulation of the EM algorithm which fits the model's parameters on a larger target Gaussian Mixture Model. When computing the mipmap level sequentially, parameters of a specific texel can be computed using this algorithm on the combination of 4 Gaussian Mixture Models sampled one level below. As mentioned in Section 1.4, memory efficiency is key in order to handle complex production scene. Tan's method uses a constant number of Gaussian elements in the Gaussian Mixture Models throughout the whole mipmap texture. This is very inefficient for textures exhibiting high-frequency details only on a fraction of the texture. Imagine a flat piece of metal where only a few scratches are scattered on the surface. While multiple Gaussian elements are needed to properly represent the underlying NDF close to the scratches, a single Gaussian element can perfectly represent the flat areas. For this reason, we use a simplification scheme that we introduce in Section 4.6 to reduce the size of the Gaussian Mixture Models based on the NDF's complexity. This way, we dynamically allocate more Gaussian elements around the high-frequency contents (like scratches) in the texture and therefore wisely manage the memory impact of our method. Notice that in the case where we only use a single Gaussian elements, our method should fall back to LEAN/LEADR.

4.2 Gaussian Mixture Models

In the field of statistics, statistical models are used to approximate the distribution of a data set. The **Gaussian distribution** is a very commonly used model for the distribution of continuous variables. The D-dimensional multivariate Gaussian distribution takes the form

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^D|\boldsymbol{\Sigma}|}} \exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}(\boldsymbol{x}-\boldsymbol{\mu})\right\}$$

with $\boldsymbol{\mu}$ the mean of the distribution and $\boldsymbol{\Sigma}$ its $D \times D$ covariance matrix.

The Gaussian distribution is a well studied model and has already been proven useful in many other Computer Graphics algorithms. This mainly comes from the simplicity of its evaluation and sampling routine. Moreover, there exists a closed-form expression of its integral over a region, as well as a closed-form formula for the convolution of multiple Gaussian distributions together.

Mixture models provide a framework for building more complex probability distributions, combining multiple simpler distribution functions. The power of mixture models lies their ability to represent complex distribution function with a limited number of parameters. Moreover, one can often find analytic expression for evaluation of integral over those models, where we would rely on expensive numerical methods such as *Monte Carlo integration* when working directly with the large data set. **Gaussian Mixtures Models** (*GMM*) are widely used in pattern recognition, statistical analysis, data mining, etc. They can be seen as a simple linear combination of of Gaussian elements, aiming at providing a richer model than the single Gaussian distribution. The probability density function of $GMM(\boldsymbol{\Theta})$ is defined by

$$p(\boldsymbol{x}|\boldsymbol{\Theta}) = \sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

with the set of parameters

$$\boldsymbol{\Theta} = \{w_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k : 0 < k < K\}$$

Given the constraint that distribution functions need to integrate to 1, the parameters must respects

$$\sum_{k=1}^{K} w_k = 1$$

The more Gaussian elements in the *GMM*, the more expressive our model is. This is really important in our method since it will allow users to adjust the complexity and cost of our method depending on their needs, the complexity of the texture and the level of accuracy they are aiming for.

4.3 Expectation-Maximization algorithm

Suppose we have a data set $\boldsymbol{X} = \{\boldsymbol{x}_1, ...\boldsymbol{x}_N\}$ that we want to model using $GMM(\boldsymbol{\Theta})$. We define the **log-likelihood** metric

$$\ln(p(\boldsymbol{X},\boldsymbol{\Theta})) = \sum_{i=1}^{N} \ln\left\{\sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right\}$$

which represents the probability of the data set $\boldsymbol{X}$ to be drawn from $GMM(\boldsymbol{\Theta})$.

Expectation-Maximization (EM) algorithm introduced by Dempster, Laird, and Rubin, 1977 is an iterative method for compute optimal parameters of a statistical model by maximizing the log-likelihood function. The EM algorithm repetitively performs three steps:

1. **Expectation step**: use the current GMM to compute the responsibilities of every Gaussian element regarding to each data point of the input data set. This quantity is defined by the following formula:

$$\gamma_{i,k} = \frac{w_k \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma})}{\sum_j^K w_k \mathcal{N}(\boldsymbol{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

2. **Maximization step**: re-estimate the weight, mean and covariance matrix of each Gaussian elements of the GMM using the responsibilities computed in the expectation step.

$$\begin{aligned} w_k &= \frac{\sum_i^N \gamma_{i,k}}{N} \\ \boldsymbol{\mu}_k &= \frac{\sum_i^N \gamma_{i,k} \boldsymbol{x}_i}{N w_k} \\ \boldsymbol{\Sigma}_k &= \frac{\sum_i^N \gamma_{i,k} \boldsymbol{x}_i x_i^T}{N w_k} - \boldsymbol{\mu}_k \boldsymbol{\mu}_k^T \end{aligned}$$

3. **Evaluation step**: evaluate the log-likelihood function with the current model. It can be shown that each update of the parameters $\boldsymbol{\Theta}$ increases the log-likelihood, ensuring the convergence of the algorithm to the maximum log-likelihood.

The EM algorithm iteratively repeats those three steps until it reaches the user-defined convergence criterion:

$$\ln(p(\boldsymbol{X}, \boldsymbol{\Theta})) > \lambda$$

Initialization With the log-likelihood function being non-convex and possessing many stationary points, the convergence of the EM algorithm often depends on the initialization of the model. A lot of research has been done on how to best initialize the parameters $\boldsymbol{\Theta}$, and clustering algorithms are popular for approaching this problem. In our implementation, we use an initialization method called **K-mean++** introduced by Arthur and Vassilvitskii, 2007. This method spreads out the Gaussian element's center in order to better cover the region of interest. The first center is chosen at random from the data points and the subsequent centers are chosen based on the remaining data points with a probability proportional to the square of the distance of the closest existing center. Note that the original method is stochastic and produces different initialization parameters every time we run it. To ensure temporal coherency across frames, we need our initialization procedure to be deterministic such that two subsequent varying textures use similar initialization parameters. This can be achieved by removing the random component of the algorithm described above, by always choosing the data point that is the furthest from all existing centers as our next center. Blömer and Bujna, 2013 introduced a method that modify and extend the resulting set of K centers to a complete GMM by computing their corresponding weights and covariance matrices.

4.4 EM algorithm for Gaussian Mixture Models (*GMM*-EM)

In this section we present a reformulation of the EM algorithm called *GMM*-EM that computes the optimal parameters of a *GMM* with K Gaussian elements to fit a larger *GMM* with K' Gaussian elements (so $K < K'$) (let's call the target Gaussian Mixture Model GMM') This reformulation is greatly inspired by the work of Verbeek, R. J. Nunnink, and Vlassis, 2006 which introduced the *Accelerated EM algorithm*. Their method uses cached statistics of clusters of the input data points to define a lower bound on the data log-likelihood. It offers a speedup at least linear in the number of data points which is great when dealing with large input dataset. By considering the Gaussian elements in the target *GMM* as clusters of data points, $\mathbf{\Theta}$ becomes the cached statistics in the *Accelerated EM algorithm*.

As descibe in the paper, a lower-bound on the log-likelihood between *GMM* and GMM' can be defined as follow:

$$\Lambda = \sum_{k'}^{K'} w_{k'} \sum_{k}^{K} \gamma_{k',k} \left(\log \frac{w_k}{\gamma_{k',k}} + \langle \log(p(\mathbf{\Theta}'_{k'}, \mathbf{\Theta}_k)) \rangle \right)$$

where

$$\langle \log(p(\mathbf{\Theta}', \mathbf{\Theta})) \rangle = -\frac{1}{2} \left(\log |\mathbf{\Sigma}| + \boldsymbol{\mu}^T \mathbf{\Sigma}^{-1} \boldsymbol{\mu} + tr(\mathbf{\Sigma}^{-1}(\mathbf{\Sigma}' + \boldsymbol{\mu}'^T \boldsymbol{\mu}')) - 2\boldsymbol{\mu}^T \mathbf{\Sigma}^{-1} \boldsymbol{\mu}' + \log(2\pi) \right)$$

and $\gamma_{k',k}$ are the responsibility values between the Gaussian elements of *GMM* and GMM' computed during the expectation step. Similarly to the original EM algorithm, the *GMM*-EM algorithm is repetitively performs 3 steps:

1. **Expectation step**: compute the responsibilities of every Gaussian elements $\mathbf{\Theta}_k$ in *GMM* regarding to every Gaussian elements $\mathbf{\Theta}'_{k'}$ in GMM'.

$$\gamma_{k',k} = \frac{w_k \exp(\langle \log(p(\mathbf{\Theta}'_{k'}, \mathbf{\Theta}_k)) \rangle}{\sum_{k''}^{K} w_{k''} \exp(\langle \log(p(\mathbf{\Theta}'_{k'}, \mathbf{\Theta}_{k''})) \rangle}$$

2. **Maximization step**: re-estimate the weight, mean and covariance matrix of each Gaussian elements of the *GMM* using the responsibilities computed in the expectation step.

$$w_k = \frac{\sum_{k'} \gamma_{k',k}}{N}$$
$$\boldsymbol{\mu}_k = \frac{\sum_{k'} \gamma_{k',k} \boldsymbol{\mu}_{k'}}{N w_k}$$
$$\mathbf{\Sigma}_k = \frac{\sum_{k'} \gamma_{k',k} (\mathbf{\Sigma}_{k'} + \boldsymbol{\mu}_{k'}^T \boldsymbol{\mu}_{k'})}{N w_k} - \boldsymbol{\mu}_k \boldsymbol{\mu}_k^T$$

3. **Evaluation step**: evaluate the log-likelihood function of the updated model.

Again, similar to the standard EM algorithm, *GMM*-EM iteratively repeats those three steps until it reaches the defined convergence criterion:

$$\Lambda > \lambda$$

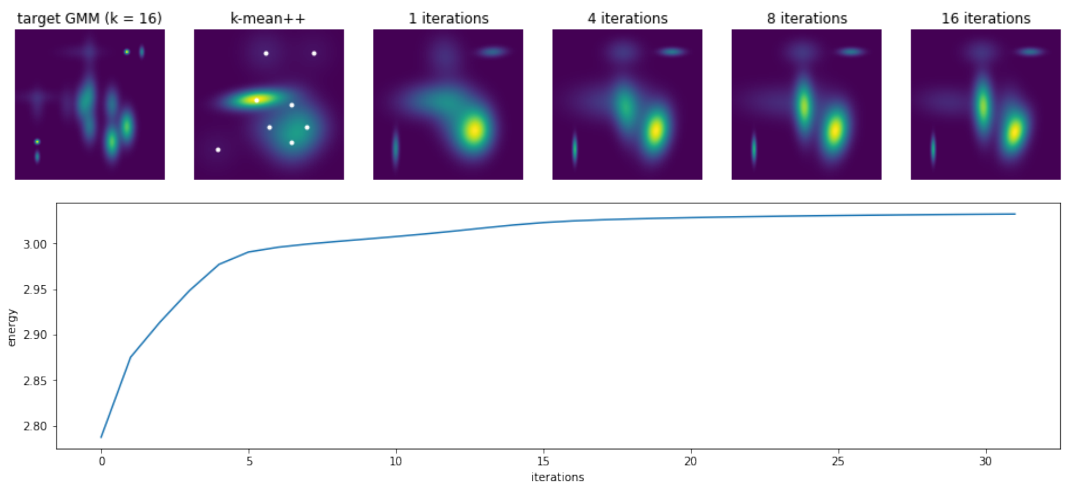


FIGURE 4.1: Fitting a *GMM* with 8 Gaussian elements on a target *GMM* with 16 Gaussian elements using the *GMM*-EM algorithm

4.5 Computing a mipmap of Gaussian Mixture Models

In this project, we extend Tan et al., 2005's work using the *GMM*-EM algorithm for better scalability. As briefly mentioned in Section 1.4, texture resolutions in a production context can reach 32K, which breaks the usability of Tan's method since the mipmap pre-computation would be too expensive. Using the *GMM*-EM algorithm, we are able to fit the parameters of the *GMM* of a specific texel on the combination of the corresponding four *GMM*s from the mipmap level below. This allows the mipmap generation to scale to high resolution textures since the fitting algorithm will never have to work with more than $4 \times K$ data points at a time, where K is the maximum size of the *GMM*s. On the other hand, with Tan's method, the fitting algorithm has to deal with large regions of the input texture for computing the GMM parameters of the texels. However, our solution implies a loss in accuracy at every level of the mipmap since the fitting isn't perfect when limiting the size of the *GMM*s. We found more important to have a method that scale to production's needs and note that it is also always possible increase the accuracy of our method by increase the maximum size of the *GMM*s (K) if needed. As a future experiment, we would like to try to fit the *GMM*'s parameters a specific texel on a wider combination of the *GMM*s coming from deeper levels of the mipmap. This could even be a control parameter in the mipmap generation tool that users could set based on the texture content and resolution. In practise, we found that using the *GMM*-EM algorithm produces good results, even at a coarser level of the mipmap. As shown on Figure 4.11, the second to last level of the mipmap (4 texels) generated with our method from a carbon fiber pattern normal map successfully represents the 4 fibers of the carbon pattern. And as expected, the coarsest texel of that mipmap represents the cross-shaped NDF, typical from the carbon fiber materials using 8 Gaussian elements, as shown on Figure A.6. Those results wouldn't be achievable using simpler methods like LEAN or LEADR.

Slope domain parameterization

Based on the current implementation of LEAN and LEADR in Manuka, we opted for the slope domain parameterization (see Heitz, 2014) for representing the NDF with *GMM*s. With this parameterization, half vectors are projected onto a plane perpendicular to the surface normal, one unit above the surface. By using the same parameterization as LEAN and LEADR in our implementation, it greatly simplifies the integration of this new scheme with the current BRDF models and the current shader library. On top of that, our mipmap generation tool takes a LEAN map as input, stores it as the finest level of the new mipmap and generates the subsequent levels based on the LEAN data. Therefore it makes sense to work with the same parameterization.

4.6 GMM simplification

As mentioned in the introduction of this chapter, our method exploits a simplification technique to adapt the size of the Gaussian Mixture Models to the complexity of the Normal Distribution Function they approximate. By doing so, we are able to drastically reduce the memory cost of our method in comparison to Han's approach as we will see later in this section.

Garcia, Nielsen, and Nock, 2010 presents a *GMM* simplification method based on a hierarchical clustering algorithm, where they explore a hierarchical representation of the initial *GMM* to find the optimal number of elements and their parameters. While their method aims at reducing the size of very large *GMM*s (thousands of elements), our goal is to find a more compact version of the original *GMM* (less than 32 elements) focusing on accuracy. Note that the simplification process will happened during the mipmap generation and not at render time, therefore we can afford a computationally more expensive approach to get more accurate results. Given the efficiency of our *GMM*-EM algorithm, we could simply fit *GMM*s of different sizes and keep the smaller one that reaches the desired level of accuracy (complexity of $O(k)$). With the same idea in mind, we could perform the optimal size search in a bisection fashion, starting fitting a *GMM* with half the number of elements, and progressively fit *GMM*s of different sizes (complexity of $O(\log(k))$). While those two ideas are simple to implement, they both rely on the fact that we have a good metric in hand to measure the accuracy of our *GMM* against the initial *GMM*. Unfortunately, this is a much harder problem than expected, and we will focus the rest of this section of solving it.

4.6.1 Gaussian Mixture Model distance metrics

Kullback-Liebler divergence

The Kullback-Liebler (KL) divergence is the measure of how one distribution diverges from a second. This metric is also called the information gain since it represent the amount of information gained about the second distribution given the first one. In the case of 2D Gaussian distributions, the Kullback-Libler divergence has the following analytic formula:

$$D_{KL}(\boldsymbol{\Theta}, \boldsymbol{\Theta}') = \frac{1}{2}\left(tr(\boldsymbol{\Sigma}'^{-1}\boldsymbol{\Sigma}) + (\boldsymbol{\mu}' - \boldsymbol{\mu})^T \boldsymbol{\Sigma}'^{-1}(\boldsymbol{\mu}' - \boldsymbol{\mu}) - 2 + \ln\left(\frac{|\boldsymbol{\Sigma}'|}{|\boldsymbol{\Sigma}|}\right)\right)$$

The Kullback-Leibler divergence is not a distance metric since it is not symmetric and does not satisfy the triangle inequality. One way of computing a symmetric distance using the KL divergence is by taking the average of the the inverted asymmetric divergence:

$$D_{sym-KL}(\boldsymbol{\Theta},\boldsymbol{\Theta}') = \frac{1}{2}(D_{KL}(\boldsymbol{\Theta},\boldsymbol{\Theta}') + D_{KL}(\boldsymbol{\Theta}',\boldsymbol{\Theta}))$$

The symmetric KL divergence is a metric between two Gaussian distribution, but it is unclear how to extend it to Gaussian Mixture Model. By using the responsibility values computed in the EM algorithm, we can compute the divergence between two *GMM*s as the weighted sum of the KL divergence between their Gaussian elements

$$D_{GMM-KL}(\boldsymbol{\Theta},\boldsymbol{\Theta}') = \sum_{k=1}^{K}\sum_{k'=1}^{K'} \gamma_{k',k} D_{sym-KL}(\boldsymbol{\Theta}_k,\boldsymbol{\Theta}'_{k'})$$

Unfortunately, as described in Section 2.3.2 for the Mean Square Error or Chi-square loss performed on NDF histogram, the symmetric KL divergence metric isn't a perceptive metric in the reflectance appearance sens. This means that while the result of the distance computation between two distributions might be large, the resulting reflectance appearances using one or the other are really similar. Two sharp Gaussian distributions with a slightly different center will give a really close appearance but their KL divergence value will be very high.

Optimal transport for Gaussian Mixture Models

In Section 2.3.2, we introduced the Wasserstein distance as a solution to this problem. Similarly, based on the work of Yongxin Chen, 2018, we can derive an efficient method for computing an upper bound of the Earth-Moving distance for Gaussian Mixture Models. An analytic formula for computing Earth-Moving distance between two Gaussian distributions is derived in their paper:

$$W_2(\boldsymbol{\Theta},\boldsymbol{\Theta}')^2 = ||\boldsymbol{\mu}-\boldsymbol{\mu}'||^2 + tr(\boldsymbol{\Sigma}+\boldsymbol{\Sigma}' - 2(\boldsymbol{\Sigma}^{\frac{1}{2}}\boldsymbol{\Sigma}'\boldsymbol{\Sigma}^{\frac{1}{2}})^{\frac{1}{2}})$$

The Wasserstein distance between two Gaussian Mixture Models is then given by

$$D_{EMD}(\boldsymbol{\Theta},\boldsymbol{\Theta}') = \min_{\pi}\sqrt{\sum_{k=1}^{K}\sum_{k'=1}^{K'} \pi(k,k')W_2(\boldsymbol{\Theta}_k,\boldsymbol{\Theta}'_{k'})^2)}$$

with $\pi(k,k')$ the optimal mass transport (OMT) map. One can compute the solution of the discrete OMT problem using standard linear programming. However, we found that using the following approximation resulted in very similar result for a reduced cost:

$$D_{EMD}(\boldsymbol{\Theta},\boldsymbol{\Theta}') = \sqrt{\sum_{k=1}^{K}\sum_{k'=1}^{K'} \pi^*(k,k')W_2(\boldsymbol{\Theta}_k,\boldsymbol{\Theta}'_{k'})^2}$$

with

$$\pi^*(k,k') \propto \frac{1}{W_2(\boldsymbol{\Theta}_k,\boldsymbol{\Theta}'_{k'})^2} \quad \text{and} \quad \sum_{k}^{K}\pi^*(k,k') = 1$$

4.6.2 Simplification scheme

Our simplification scheme uses the distance metric defined above to compare a reduced-size *GMM* against the fitted *GMM* with the maximum number of elements.

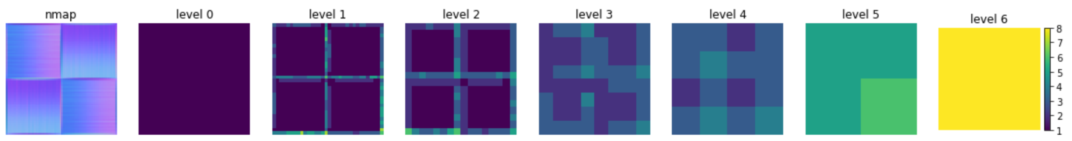
If the distance is smaller than a user-defined **simplification threshold**, then the bisection search continues with a smaller size *GMM*, otherwise larger. The search is performed until the optimal size *GMM* for the given simplification threshold is found. Pseudo code 4.6.2 implements the bisection search using the approximation of Earth-Moving distance function and the *GMM*-EM fitting algorithm.

```
// target Gaussian Mixture Model
GMM targetGMM = ...;

// maximum GMM size
int k = 8;
// simplification threshold
float st = 0.002;

GMM largeGMM = gmm_em_fitting(targetGMM, k);

k /= 2;
int dk = k;

while (dk >= 1)
{
    GMM reducedGMM = gmm_em_fitting(targetGMM, k);

    // bisection based on distance metric
    dk /= 2;
    if(distanceEMD(reducedGMM, largeGMM) > st)
        k += dk;
    else
        k -= dk;
}
```

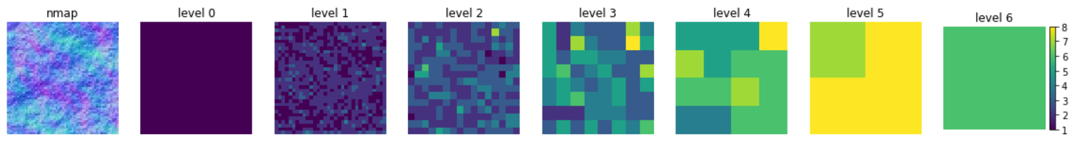
LISTING 4.1: pseudo code for simplification scheme

Figure 4.2 shows the results of our simplification scheme on different normal maps. We render a density map of the *GMM*s for every level of the generated mipmap texture. Texels at level 0 always contain a single Gaussian element since the NDFs of those texels are Dirac delta functions representing a single normal. As we can observe on Figure 4.2c, the size of the *GMM*s around the scratches are higher while only a single Gaussian elements are used to represent the flatter areas. At a higher mipmap level, the size of the *GMM*s depends on their ability to represent really complex NDFs. For instance, at a larger scale, the *waves* normal map exhibits lower frequency features that can be handled by a reasonable sized *GMM*s. However, very large *GMM*s would be needed to capture the details of the scratches at a large scale on the *scratches* texture. Thus smaller *GMM*s are used to represent a rougher approximation of the surface.

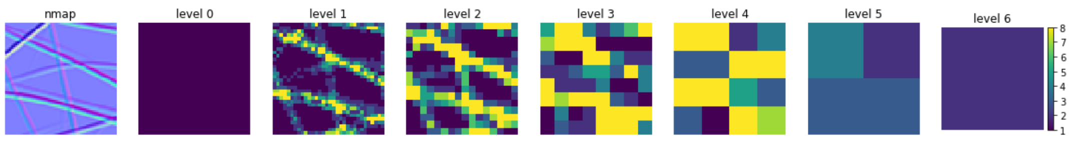
Table 4.1 demonstrates the cost reductions of using the simplification scheme during the mipmap generation. We compare our results to the LEAN/LEADR method which only uses a single Gaussian element per texel, and Tan's method which uses a constant number of Gaussian elements throughout the whole mipmap. Note that the results of the simplification method depends on the simplification threshold defined by the user. This threshold defines the acceptable distance to the larger *GMM* at which we allow the algorithm to reduce the *GMM*. By varying the value of this threshold, we can navigate between using a single Gaussian elements (LEAN) to using the maximum number of Gaussian elements at every texel. Figure 4.2c and 4.2d show the result of the simplification scheme ran on the same normal map with



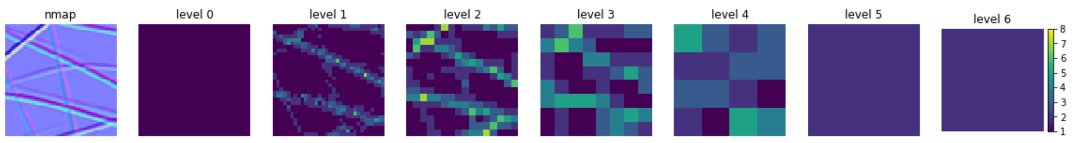
(A) *carbon* normal map - simplification threshold = 0.002



(B) *waves* normal map - simplification threshold = 0.002



(C) *scratches* normal map - simplification threshold = 0.002



(D) *scratches* normal map - simplification threshold = 0.008

FIGURE 4.2: Gaussian density at the different mipmap levels

different simplification threshold. As observed in the last two row of Table 4.1, the overall size of the mipmap decreases as we increase the simplification threshold.

	LEAN	Han's method	ours	reduction (%)
carbon (0.002)	5461	27304	7287	73%
waves (0.002)	5461	27304	12853	52%
scratches (0.002)	5461	27304	10234	62%
scratches (0.008)	5461	27304	6971	74%

TABLE 4.1: Simplification and Gaussian elements density statistics

4.7 Reflectance filtering with Gaussian Mixture Models

Mipmap generation is a form of texture pre-filtering such that at render time we only have to combine a few pre-computed values. In our case, while the fitting algorithm is employed to compute the parameters of the *GMM*s stored in the mipmap, we need a efficient solution to combine *GMM*s sampled on the texture at render time. In this section we take a closer look at different solutions for combining multiple set of *GMM* parameters to produce the final *GMM* that will be used by the renderer as NDF for the BRDF models. In the context of our production renderer, we need to minimize the size of the resulting *GMM*s since those will be stored on the *grid* during the pre-shading phase.

In Chapter 1 we introduced the concept of magnification and minification in the context of texture filtering. As a reminder, minification is the case where the representation of the data has a lower resolution than the input data. Data points have to be appropriately combined to produce a value representative of the input data and prevent aliasing. On the other hand, magnification occurs when the representation of the data has a higher resolution than the data itself. Data need to be interpolated to generate the in between values.

4.7.1 Filtering and interpolation of Gaussian elements

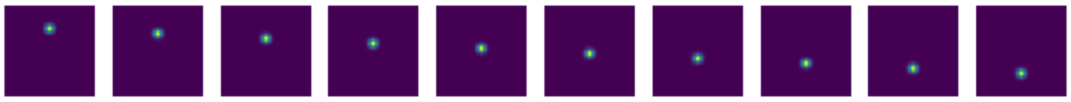
Given two Gaussian elements and their parameters $\mathbf{\Theta}_a$ and $\mathbf{\Theta}_b$, we define here the filtering and interpolation operator that can be used to generate a range of Gaussian distributions along the axis t. Both operators result in $\mathbf{\Theta}_a$ at $t = 0$ and $\mathbf{\Theta}_b$ at $t = 1$.

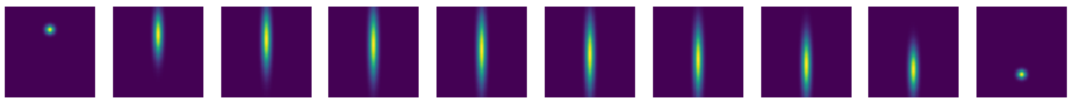
- **Interpolation** linearly interpolate the centers and the covariance matrices of $\mathbf{\Theta}_a$ and $\mathbf{\Theta}_b$. If those Gaussian distributions represent NDFs, intuitively this operator can be seen as the simulation of a smooth curved surface, starting with the orientation and the roughness of $\mathbf{\Theta}_a$ and ending with the orientation and roughness of $\mathbf{\Theta}_b$. Therefore, `Interpolation` is the right operator in the case of magnification.

$$\mathbf{\Theta}_c = \texttt{Interpolation}(\mathbf{\Theta}_a, \mathbf{\Theta}_b, t) = \begin{cases} \boldsymbol{\mu}_c = (1-t)\cdot\boldsymbol{\mu}_a + t\cdot\boldsymbol{\mu}_b, \\ \mathbf{\Sigma}_c = (1-t)\cdot\mathbf{\Sigma}_a + t\cdot\mathbf{\Sigma}_b \end{cases}$$

- **Filtering** results in a distribution that tries to best fit both $\mathbf{\Theta}_a$ and $\mathbf{\Theta}_b$. It does so by increasing the variance of the Gaussian distribution which gives rougher results. Filtering is the result of minimizing the Mean Square Error as described in Olano and Baker, 2010.

$$\mathbf{\Theta}_c = \texttt{Filtering}(\mathbf{\Theta}_a, \mathbf{\Theta}_b, t) = \begin{cases} \boldsymbol{\mu}_c = (1-t)\cdot\boldsymbol{\mu}_a + t\cdot\boldsymbol{\mu}_b, \\ \mathbf{\Sigma}_c = (1-t)\cdot(\mathbf{\Sigma}_a - \boldsymbol{\mu}_a^T\boldsymbol{\mu}_a) + t\cdot(\mathbf{\Sigma}_b - \boldsymbol{\mu}_b^T\boldsymbol{\mu}_b) + \boldsymbol{\mu}_c^T\boldsymbol{\mu}_c \end{cases}$$


(A) interpolation


(B) filtering

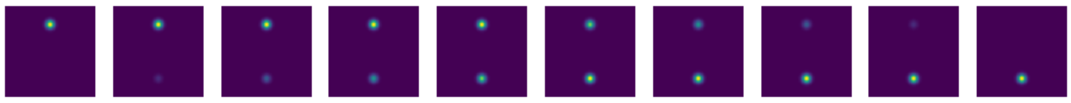

(C) blending

FIGURE 4.3: Interpolation, filtering and blending operation on Gaussian distributions

Figure 4.3a shows a range of Gaussian distributions generated with the `Interpolation` operator. As expected, the distributions produced around $t = 0.5$ do not represent any existing data of the two original Gaussian distributions. On the other hand, `Filtering` increases the variance of the model to counter this effect as we see on Figure 4.3b. Unfortunately, this results in rougher results in the context of reflectance filtering during minification. Ideally, results of the reflectance texture minification would only contain data existing in the two original Gaussian distributions while still providing a smooth transition from one to the other. This ideal solution is shown on Figure 4.3c and is the result of a third operator called `Blending`.

- **Blending**, unlike the two other operators, never modifies the parameters of the Gaussian distribution, and therefore only produces distributions that exclusively represent existing data of the two original Gaussian distributions. This can only been achieved using higher order models, such as Gaussian Mixture Models, where only the weights of the Gaussian elements will be affected by the `Blending` operator.

$$\boldsymbol{\Theta} = \texttt{Blending}(\boldsymbol{\Theta}_a, \boldsymbol{\Theta}_b, t) = \{1 - t, \boldsymbol{\mu}_a, \boldsymbol{\Sigma}_a\} \cup \{t, \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b\}$$

Fortunately for us, we are already dealing with Gaussian Mixture Models in our method, so it is not an issue if the minification procedure relies on them to produce more accurate results.

4.7.2 Reflectance filtering operators on *GMM*s

It is necessary for us to extend those operators to work with Gaussian Mixture Models as input parameters rather than single Gaussian distributions in order to use them in our method. Tan et al., 2005; Han et al., 2007 use the filtering operator defined above on individual elements of the *GMM*s they want to combine. They first *align* the neighboring *GMM*s during the mipmap generation by adding a regularization term in the EM log-likelihood function. This term ensures that elements with the same index in neighboring *GMM*s are close to each other. With that, as for Olano and Baker, 2010; Dupuy et al., 2013, they can rely on hardware interpolation to perform the filtering efficiently at render time. While their method is really efficient and compliant with anisotropic filtering, the results aren't really accurate, drastically roughening the result due to the use of the `Filtering` operator. Moreover, alignment is never perfect, which can also worsen the results as shown on Figure A.3.

Linear Gaussian Mixture Models Blending

We therefore propose to use our new `Blending` operator to ensure the preservation of the input *GMM*s. As we have seen earlier, the `Blending` operator produces a higher order *GMM* by linearly combining all the elements of the input *GMM*s. While this would give us a perfectly accurate representation of the input *GMM*s, this solution is too expensive to be ran in a production renderer. Even using a simple trilinear filtering function, we would have to combine 8 set of *GMM*s parameters, which would drastically impact the performance of our method. For this reason, we desire to limit the number of Gaussian elements produced by the `Blending` operator. One solution is to use our GMM-EM algorithm to fit a lower order *GMM* to the *GMM* resulting from the `Blending` operator. We call it the `Fitting` operator. While being accurate, this strategy is expensive, given the fact that it will have to be used at render time. However, by diminishing the number of iteration the algorithm takes, it is possible to drastically reduce its cost, while losing in accuracy. Note also that slight changes to the initialization method will have to be made to ensure its stability and avoid any flickering artifact. To achieve better performances, we explored other strategies that might produce similar results at a lower cost.

To find a better solution, we will go back to a simple scenario where we are trying to blend two *GMMs* with the same number of elements. Figure 4.4 shows the weights resulting from the `Blending` operator on two arbitrary *GMMs* of the same size. We notice that weights of the elements of the first *GMM* (left) become insignificant as t gets closer to 1. We could decide to safely drop those elements as soon as their weight is smaller than a defined threshold. However, we cannot constrain the number of elements in the resulting *GMMs* this way.

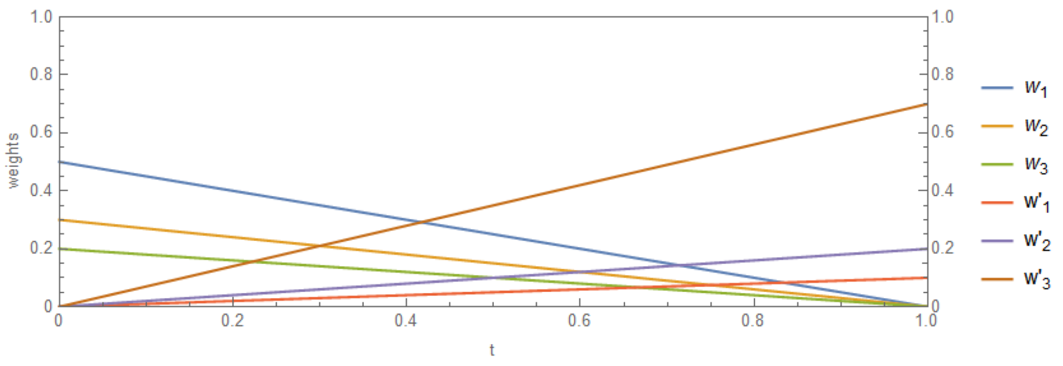


FIGURE 4.4: Gaussian element weights resulting from the `Blending` operation on two *GMMs* of the same size

Max-Blending If we want to constraint our resulting *GMMs* to a specific number of elements K, a solution would be to only keep the K elements with the largest weight at any t. This solution is implemented by the `Max-Blending` operator and Figure 4.5 illustrates its application on the same two *GMMs*.

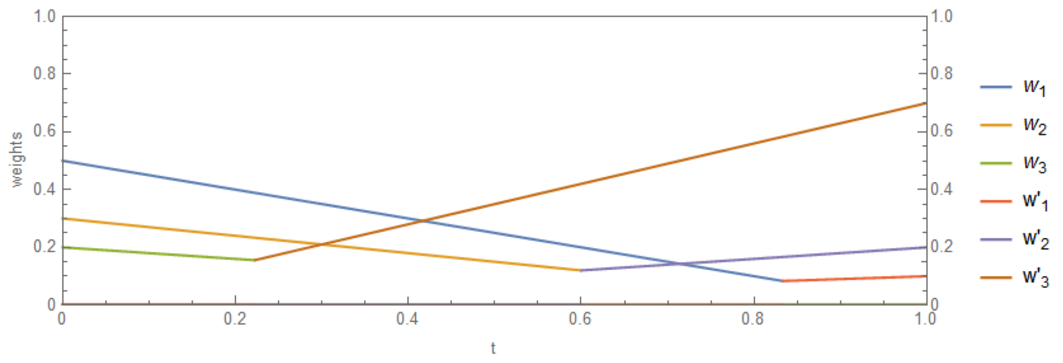


FIGURE 4.5: Gaussian element weights resulting from the `Max-Blending` operation on two *GMMs* with $K = 3$

When looking at a single weight curve on this plot, it is clear that this operator introduces discontinuities that will result in highlight flickering in the renders. In order to prevent the operator to produce any flickering artifacts, we need the weights of the Gaussian elements to reach zero before being replaced by another element. A key observation can be made on Figure 4.5: the order in which the Gaussian elements of the first *GMMs* are being dropped follows the increasing weight order. Moreover, the order in which the Gaussian elements of the second *GMMs* are added to the resulting *GMM* corresponds to the inverse of the increasing weight order. Using this logic, we can define a *replacement mapping* of Gaussian elements between the two *GMMs*, as well their respective *replacement point* on the t-axis defined by the pair of weight $\{w_a, w_b\}$ as:

$$\beta = \frac{w_a}{w_a + w_b}$$

Assuming both *GMMs* have *K* Gaussian elements sorted based on their weights, we can give a definition of the `Max-Blending` operator for the sake of completeness:

$$\texttt{Max-Blending}(\mathbf{\Theta}, \mathbf{\Theta}', t) = \left(\bigcup_{\substack{i=1 \\ j=k-i+1}}^{k} \{W(w_i, w'_j, t), \boldsymbol{\mu}_i, \mathbf{\Sigma}_i\} \right) \cup \left(\bigcup_{\substack{i=1 \\ j=k-i+1}}^{k'} \{\{W(w'_i, w_j, 1-t), \boldsymbol{\mu}'_i, \mathbf{\Sigma}'_i\} \right)$$

with

$$W(w_a, w_b, t) = \begin{cases} (1-t) \cdot w_a & \text{if } t < \frac{w_a}{w_a + w_b} \\ 0 & \text{otherwise} \end{cases}$$

Smooth-Blending To solve the discontinuity issue of the `Max-Blending` operator, we need to ensure that the weights of Gaussian elements reach zero before being replaced. In other words, at the *replacement point*, the weights of the corresponding two Gaussian elements should be equal to zero. This can easily be achieved by changing the slopes of the weights in the `Max-Blending` operator as done in the `Smooth-Blending` operator.

$$\texttt{Smooth-Blending}(\mathbf{\Theta}, \mathbf{\Theta}', t) = \left(\bigcup_{\substack{i=1 \\ j=k-i+1}}^{k} \{W(w_i, w'_j, t), \boldsymbol{\mu}_i, \mathbf{\Sigma}_i\} \right) \cup \left(\bigcup_{\substack{i=1 \\ j=k-i+1}}^{k'} \{\{W(w'_i, w_j, 1-t), \boldsymbol{\mu}'_i, \mathbf{\Sigma}'_i\} \right)$$

with

$$W(w_a, w_b, t) = \begin{cases} \left(1 - \frac{t(w_a + w_b)}{w_a}\right) \cdot w_a & \text{if } t < \frac{w_a}{w_a + w_b} \\ 0 & \text{otherwise} \end{cases}$$

Figure 4.6 shows the weight curves resulting from the `Smooth-Blending` operator on the two sorted *GMMs*.

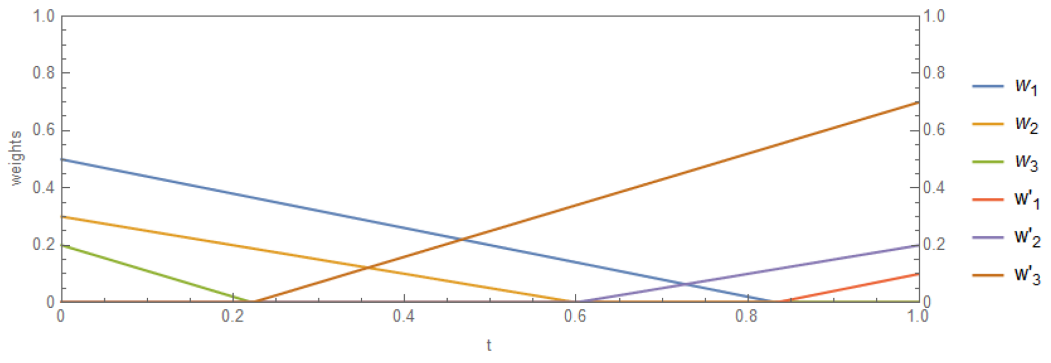


FIGURE 4.6: Gaussian element weights resulting from the `Smooth-Blending` operation on two *GMMs*

As our Gaussian Mixture Models represent distribution function, it is implied that the weights of the Gaussian elements in our *GMMs* sum to one. Unfortunately, this normalization constraint has a big impact on the curve of the weights for our Gaussian elements. As illustrated on Figure 4.7, the curve of the weights do not behave as expected once the normalization applied. Although, the discontinuities in the derivatives of the weights introduces by the normalization do not cause any flickering artifacts in practice.

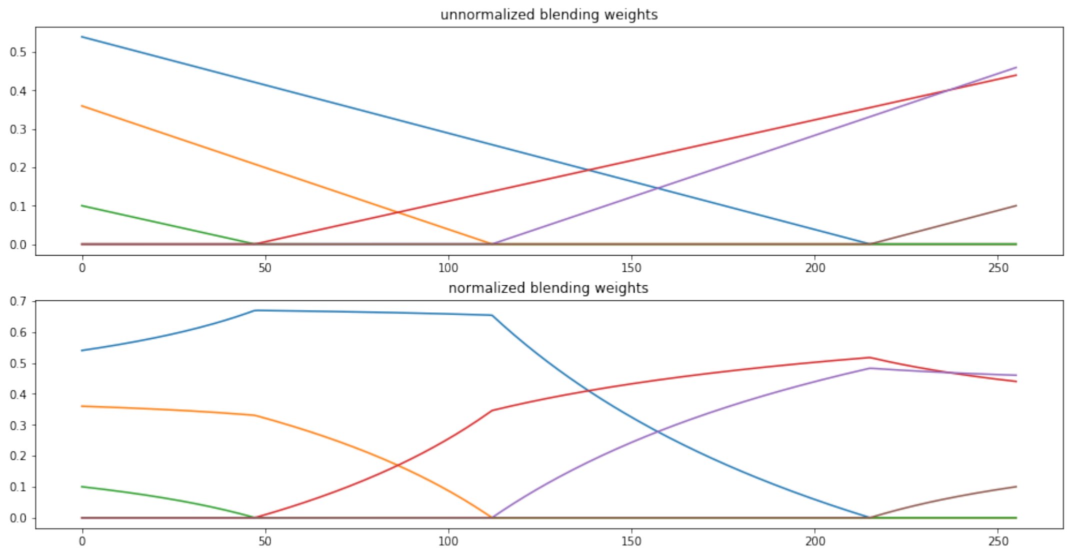


FIGURE 4.7: Impact of the normalization of the weight for the `Smooth-Blending` operator

Bilinear Gaussian Mixture Model Blending

We now have an operator for smoothly blending between two *GMM*s. However, in order to be able to use this operator as a filtering method, we need to be able to blend more than two *GMM*s. To begin with, we are indeed interested in a bilinear extension of the former `Blending` operator. This operator would blend between four *GMM*s $(GMM_A, GMM_B, GMM_C, GMM_D)$ along two axes s and t. This operator returns the original *GMM*s at the unit square corners such that:

$$\begin{aligned} (s,t) &= (0,0) \Rightarrow GMM_A \\ (s,t) &= (0,1) \Rightarrow GMM_B \\ (s,t) &= (1,0) \Rightarrow GMM_C \\ (s,t) &= (1,1) \Rightarrow GMM_D \end{aligned} \tag{4.1}$$

One way to achieve this is to first blend the pairs of *GMM*s (GMM_A, GMM_B) and (GMM_C, GMM_D) with the `Smooth-Blending` operator and then blend their results with the same operator to produce the final blended *GMM*. Unfortunately, this produces unsmooth result due to discontinuities in the pairing of the Gaussian elements during the third `Smooth-Blending` operation.

For this reason, we will define a new `Bilinear-Blending` (smooth) operator which will consider the four *GMM*s all together to produce smooth results. Similarly to the previous blending operators, it will match individual Gaussian elements of the four *GMM*s and perform the blending on those quadruplets of Gaussian elements independently. As GMM_A transitions to GMM_B along the s axis and to GMM_C along the t axis, following the same logic as in the previous operators, Gaussian elements of GMM_A with high weights should be matched with Gaussian elements of GMM_B and GMM_C that have small weights. The same applies to Gaussian elements of GMM_D with larger weights. Therefore, one can say that the blending is perform in ascending weight order on the elements of GMM_A and GMM_D and in decreasing weights order for GMM_B and GMM_C.

As for the `Smooth-Blending` operator, we want the weight of the Gaussian elements to reach zero before being replaced by another Gaussian elements. While we were working with curves to describe the weights in the 1D operator, extending the blending operator to the 2D world implies the use of planes instead. In the same way every pair of Gaussian elements defined a *replacement point* along the t axis in the `Smooth-Blending`, `Bilinear-Blending` defines 5 *replacement points* per quadruplet of Gaussian elements, handling all the possible replacement cases between the four GMMs. The weights of the Gaussian elements are constrained to be zero at those replacement points, which combined with Eq 4.1 results in a set of constraints on the weights of the four GMMs on the unit square shown in Table 4.2. The location of the replacement points, as illustrated on Figure 4.8, are defined as follow:

$$\begin{aligned} a &= \frac{w_A}{w_A+w_B} \\ b &= \frac{w_B}{w_B+w_C} \\ c &= \frac{w_C}{w_C+w_D} \\ d &= \frac{w_A}{w_A+w_C} \\ f &= 1-\frac{w_C+w_D}{w_A+w_B+w_C+w_D} \\ j &= 1-\frac{w_B+w_D}{w_A+w_B+w_C+w_D} \end{aligned} \tag{4.2}$$

s	t	w_A	w_B	w_C	w_D
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1
0	a	0	0	0	0
d	0	0	0	0	0
f	j	0	0	0	0
b	1	0	0	0	0
1	c	0	0	0	0

TABLE 4.2: Constraints table for the weights of the quadruplet of Gaussian elements for the `Bilinear-Blending` operator

Given those 9 constraints, we can derive the equations of 8 planes that can be used to compute the weights of the quadruplet of Gaussian elements. Figure 4.8 shows how those 8 planes are arranged on the unit square and Eq. 4.3 defines the planes equations. Figure 4.9 illustrates those same planes in a 3D plot where the third dimension represents the amplitude of the weights and the colors indicate the different Gaussian element active at a specific location along the two axis.

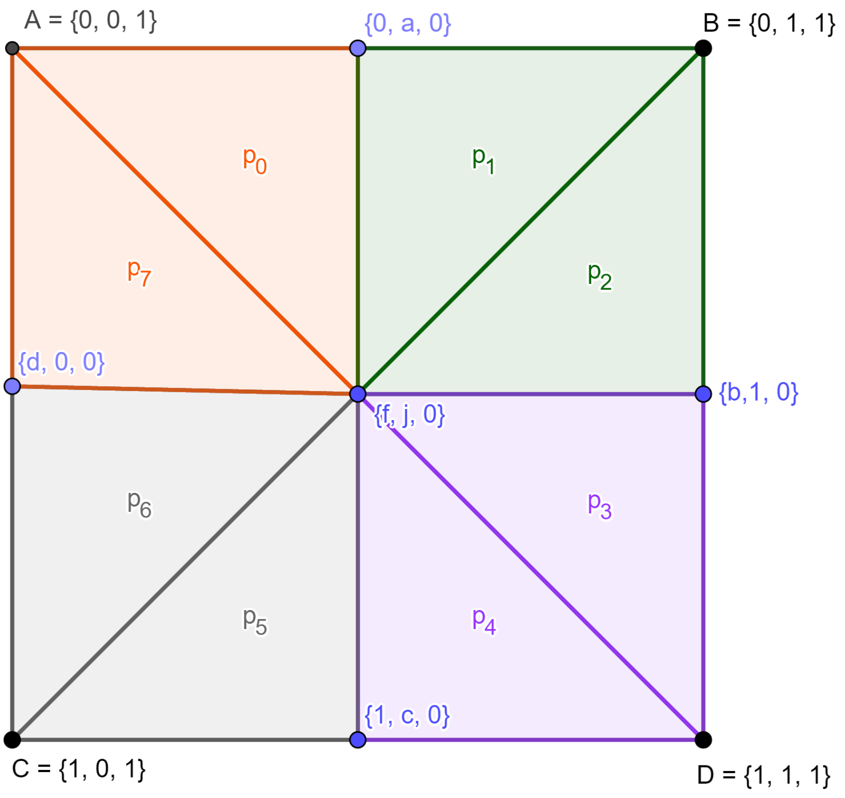


FIGURE 4.8: Configuration of 8 planes in the `Bilinear-Blending` operator

$$
\begin{aligned}
p_0 &= \frac{x(j-a)}{af} - \frac{y}{a} + 1\\
p_1 &= \frac{x(a-j)}{f-af} + \frac{y}{1-a} + \frac{a}{a-1}\\
p_2 &= \frac{y(b-f)}{b-bj} + \frac{f-bj}{b-bj} - \frac{x}{b}\\
p_3 &= \frac{y(f-b)}{(b-1)(j-1)} + \frac{bj-f}{(b-1)(j-1)} + \frac{x}{1-b}\\
p_4 &= \frac{x(j-c)}{(c-1)(f-1)} + \frac{cf-j}{(c-1)(f-1)} + \frac{y}{1-c}\\
p_5 &= \frac{x(c-j)}{c-cf} + \frac{cf-j}{c(f-1)} - \frac{y}{c}\\
p_6 &= \frac{y(d-f)}{j-dj} + \frac{x}{1-d} + \frac{d}{d-1}\\
p_7 &= \frac{y(f-d)}{df} - \frac{x}{d} + 1
\end{aligned}
\tag{4.3}
$$

By varying the weight of the original Gaussian elements, the location of the replacement points move and therefore the planes equations changes as well. Figure 4.10 illustrates the 8 planes with a different configuration for the original weights. Since each plane's projection on the unit square forms a triangle, we can use triangle intersection to define the plane corresponding to a given location along the two axis. The `Bilinear-Blending` operator then evaluates the value of this plane and attributes it to the corresponding Gaussian elements, while setting a zero weight to all the other Gaussian elements of the quadruplet.

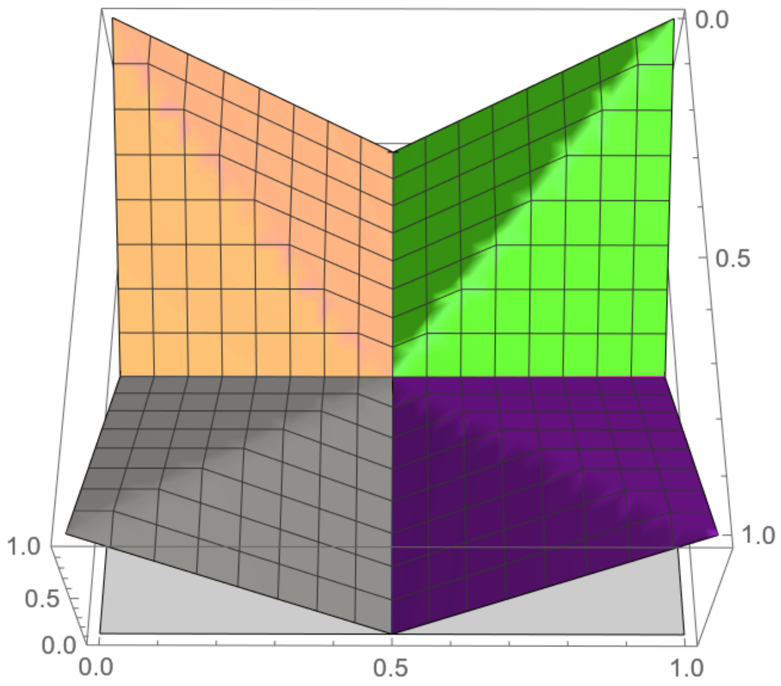


FIGURE 4.9: 3D representation of the 8 planes

Comparison of the different blending strategies

We included figures comparing the different blending strategies introduced in this section in Appendix 5. On those figures, the 4 original *GMMs* (located at the 4 corners of each grid) represent the NDFs of different fiber orientations of the carbon fiber texture shown on Figure 4.11. Figure A.1 is the result of the `Blending` operator, which is our reference solution to the filtering problem, quadrupling the size of the output *GMM* and conserving every single Gaussian element. As expected, the NDF at the center of this grid, which represent the sum of the 4 original distributions (normalized), is a cross shaped NDF. This will result in the well known cross-like shaped highlight of carbon fiber materials.

As illustrated in Figure A.2 and Figure A.3, the lack of possible alignment of the different Gaussian elements in the 4 original *GMMs* results in poor quality of the filtering. Not only the NDF at the center of the grid do not reproduce the cross shape, but every single NDFs on this grid is completely loosing the structures contained in reference results. In practice this would result is render far from the ground truth results and really rough surfaces. Those figures illustrate a failure case for Tan's and Han's method. On the other hand, as shown on Figure A.4 and Figure A.5, the `Max-Blending` operator and the `Smooth-Blending` operator totally preserve the structure of the original *GMM* and produce a cross shaped NDF close to the ground truth solution. As discussed in this section above, `Max-Blending` introduces flickering artifacts due to discontinuities in the curve of the Gaussian element's weights, therefore the `Smooth-Blending` operator is the better solution. Finally, the `Fitting` operator gives the best results compared to ground truth as shown in Figure A.6. Note that those result are using a single iteration of the *GMM*-EM algorithm after a K-mean++ initialization step to minimize the cost of the operator. Better results could be achieved at a higher cost by running multiple iterations of the *GMM*-EM algorithm.

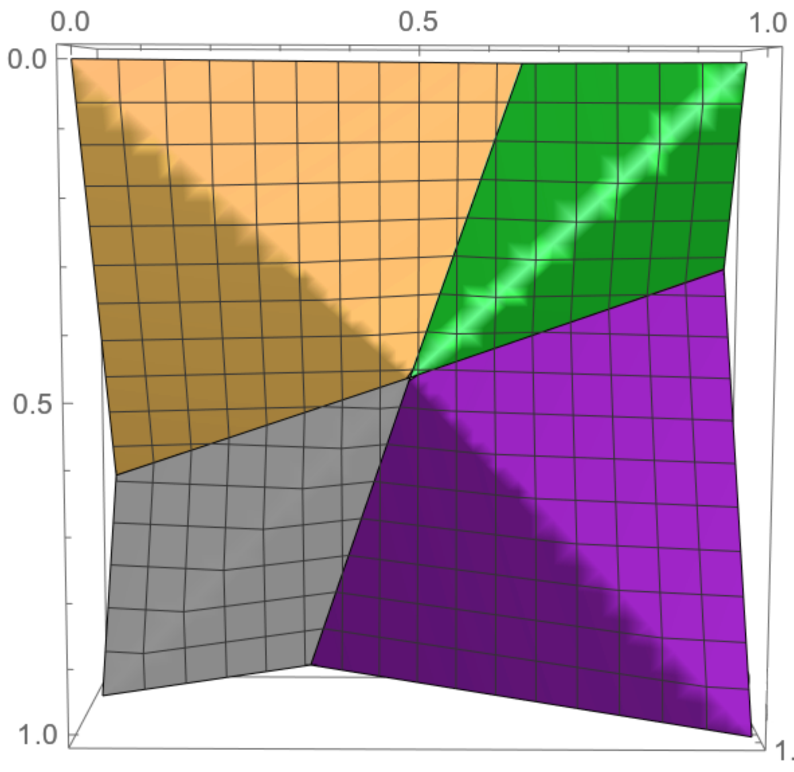


FIGURE 4.10: 8 planes with a different configuration for the original weights

Blending with other filtering methods

As discussed in Chapter 1, there exist many other filtering methods, often more complex than the simple bilinear filtering. Those methods tries to better place samples (and often use more that 4 samples) in the texture space in a way that better fits a given filter region. For instance, for really anisotropic filter region, which often occurs at grazing angle, bilinear filtering will only account for the main axis of the filter region, resulting in blurring of the texture content along the other axis. For our method to be used in a production renderer, it is important to discuss the way blending strategies can work with more sophisticated filtering method.

Trilinear filtering extends bilinear filtering to another dimension, ensuring a smooth transition between different levels of the mipmap. Even for this simple case, it is tricky to extend the `Bilinear-Blending` operator to a higher dimensionality since now constraints in Table 4.2 would become functions of the third axis. However, we realized that in practice, applying the `Bilinear-Blending` operator on both mipmap level individually and then using the `Smooth-Blending` operator to blend the result of the bilinear blending produces good and stable results. Unfortunately, it seems to be impossible to extend the `Bilinear-Blending` operator to anisotropic filtering methods. This comes from the fact that this operator relies on the matching of the Gaussian elements in the 4 *GMM*s, which doesn't exists with an arbitrary number of *GMM*s. While we could take a sequence of bilinear samples at a finer level of the mipmap, and then merge those results again using our blending operators, such solution will probably eventually break due to a lack of robustness and stability.

On the other hand, the `Fitting` operator can be applied on *GMM*s of any sizes, meaning that we can combine as many texture samples (*GMM*s) as we like before proceeding to the blending of the *GMM*s. Also, the `Fitting` operator is stable and

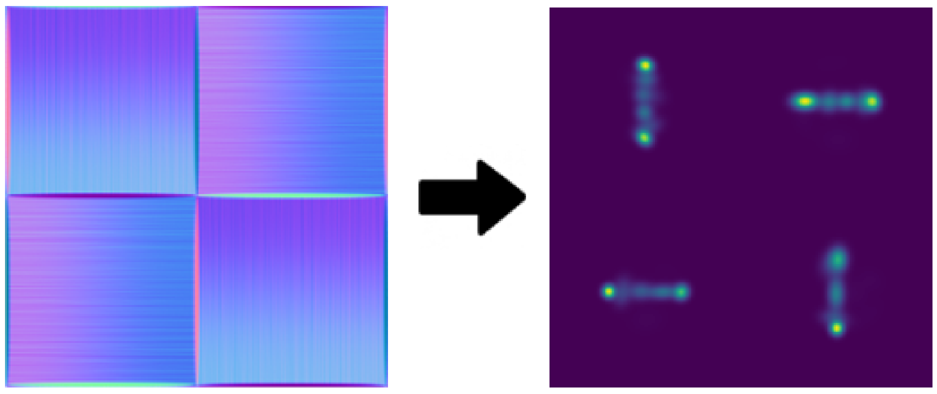


FIGURE 4.11: Carbon fiber normal map texture and the 4 NDFs corresponding to each one of the four fiber orientation

robust, unlike the `Max-Blending` operator. While this operator comes at a higher cost, well written optimized code leveraging vectorization capabilities of modern CPUs and the use of approximation in the *GMM*-EM could bring down to the cost of this operator to acceptable. Also, on top of the benefits of being compatible with any modern filtering method, this operator produces higher quality results than any other operator (besides `Blending`). For this reason, we considered both the `Fitting` operator and the `Smooth-Blending` operator in our implementation in Manuka and ran series of tests to get a better understand of both operator's capability.

4.8 Implementation in a production renderer

In order to implement this method in Manuka, a lot of changes had to be made on various components of the renderer and Weta's pipeline:

- **Mipmap generation tool**: We implemented a C++ *GMM* mipmap generation tool that uses the fitting and simplification schemes introduced above to compute the parameters of the *GMM* mipmap for a given LEAN map. The fitting procedure can be highly parallelized across the texels of a single mipmap level, greatly speeding up the mipmap generation. Each Gaussian element is composed of 6 parameters (weight, x and y coordinates of the center and 3 elements of the diagonal covariance matrix). With K the user-defined maximum number of Gaussian elements in the *GMM*s, each texel can contain up to $6 \times K$ floating points numbers. Those values are stored in different individual channels of in an EXR file, relying on the built-in compression algorithm of the OpenEXR framework to reduced unused regions of certain channels.

- **Texture engine custom sampler**: Manuka's texture engine is highly modular, implementing different sampling and filtering strategies to best fit production needs. We extended the texture engine with a custom sampler that implements both the `Bilinear-Blending` operator and the `Fitting` operator and can properly read the multi-channels mipmap of *GMM* parameters. Code efficiency of this sampler is a crucial part of our method since this will be running at render time during the pre-shading phase. We also expose various parameters on the texture engine's interface such that the shader code has the ability to vary the number of Gaussian elements in the resulting *GMM*s, the number of iteration for the `Fitting` operator, or other values like the simplification threshold or the convergence criterion of the *GMM*-EM algorithm.

- **Shader code**: Custom shader code had to be written as well to ensure the proper use of the new texture engine's interface and make sure to correctly handle the shading data (*GMM* parameters) to the renderer.
- **Parameter storage on the *grid***: In the pre-shading phase, the material parameters computed during the shader evaluation have to be stored on the *grid*. Some code modifications had to be done here to handle the many parameters of the *GMM*s and properly compress those for optimal performance.
- **Stochastic sampling of the *GMM*s**: During the light transport phase, the *GMM* parameters of 4 *vertices* around the hitting point are recovered from the *grid*. To avoid the burden of re-implementing every single BSDF model to support mixture of Gaussians as NDF, we decided to stochastically sample one of the Gaussian elements of those 4 *GMM*s and use it as a single LEAN lobe. Since many samples per pixels are taken during the light transport phase, this strategies will eventually sample all the different Gaussian elements and therefore converge to the right solution. Notice that this will introduce extra Monte-Carlo noise slowing down the convergence of the render. This noise can be reduce by using important sampling (Veach, 1998) based on the weights of the different Gaussian elements in the sampling procedure.

Unfortunately, after implementing both the `Bilinear-Blending` operator and the `Fitting` operator, we realized that the `Bilinear-Blending` operator is not compatible with the pre-shading architecture of Manuka. This operator relies on the subsequent sampling of many points on the unit square (Figure 4.8) to properly account for all the components of the underlying NDF, while Manuka only does a single texture look-up per vertices during the pre-shading phase. Therefore, it will always under-sample the NDF content which results in aliasing and flickering artifacts, where different mixture of Gaussian elements are present on the grid when the tessellation changes (for instance if we slide the tessellation grid along one axis). This would not be the case in a *shade-on-hit* renderer. For this reason, our only option is to use the `Fitting` operation in the texture engine to compute the final *GMM* parameters that will be stored on the grid.

4.9 Results and comparisons

In this section we will compare the results of our reflectance filtering method using Gaussian Mixture Models with the `Fitting` operator against the current LEAN implementation in Manuka. Unfortunately, due to the lack of flexibility and aliasing artifacts, we couldn't properly implement the `Bilinear-Blending` operator in our production environment. Moreover, the fact that it could only work in a "bilinear-fashion" drastically complicated its implementation in Manuka's texture engine. For those reasons, we won't show any renders using that operator in this section. However, given the accuracy of the resulting NDFs as shown in Figure A.5, we believe that it would be a considerable improvement on LEAN in another renderer.

In order to compare our method with LEAN, we produced a range of renders of the same scene at different resolutions. The filter region is driven by the pixel footprint in world space therefore rendering at different resolutions will trigger the texture engine to access different levels of the mipmap. For instance, a high resolution render implies a fine filter region so the texture engine will gather samples from a lower (finer) mipmap level. For the renders using our method, we used a single generated mipmap with a maximum of 8 Gaussian elements per texel. However, we vary the maximum number of Gaussian elements in the mixtures produced by the `Fitting` operator in the texture engine. We also vary the number of iterations that operator performs to get a sens of the possible tradeoff between accuracy and performance. Every figure in Appendix A contains a grid of renders at a specific resolution. Each grid is composed of six renders using different parameters for our method. Also, for every render, we used a compositing tool (Nuke) to rescale the render to a fix resolution in order to facilitate the comparisons between the different figures. Below the rescaled renders are the renders at their respective resolution, which give an idea of the appearance of the reflectance filtering result in practice (zooming in and out). Also, we composed the reference result on the right half of every render in the grid to assess the quality of the reflectance filtering. This reference result is computed at a very high resolution and down-scaled to match the resolution of the renders in the grid.

At the time of writing this thesis, the implementation of our method in Manuka is still far from optimal and hacky, therefore, given the complexity of the production renderer, the recorded timings of the `Fitting` operation and the extra memory allocated on the grid by our method are pretty unreliable and irrelevant. For this reason, we are not including any performance benchmark in this write-up. However, note that it is expected that LEAN will always be cheaper than our solution, and the more Gaussian elements in our *GMM*s or the more iterations the `Fitting` operator performs, the more expensive our method will be.

Most of the tests we ran with our method were using high quality production textures and we spent a lot of time ensuring the correctness and convergence of our method on those textures. Unfortunately, we were not allowed to use those renders in this thesis so we had to run another round of tests on another texture (lower quality). The texture used for those comparisons is the same carbon fiber normal map used to demonstrate the capability of the different blending operators (Figure 4.11). In the test scene, the camera faces a plane on which the texture is projected. The scene is illuminated by a point light source at the camera's position.

On Figure B.1 and Figure B.2, the texture engine accesses the first levels of the mipmap, where the reflectance filtering still approximates fairly well the true underlying NDF. Indeed, the results are really similar across the grids and the reference parts are almost indistinguishable. On Figure B.3 and Figure B.4, the LEAN method quickly starts to roughen the surface, while our method better conserves the cross-like highlight. Interestingly, the renders where the `Fitting` operator is constrained to produce *GMM*s with a maximum of 2 Gaussian elements are almost as good as the ones with a maximum of 8 Gaussian elements. This proved to be a real benefit in practice since it drastically reduces the memory cost on the grid, as well as the noise introduced by the stochastic sampling procedure in the light transport phase. Finally, Figure B.6 illustrates the failure cases of the LEAN method where the highlight becomes a single Gaussian shape. On the other hand, our method tends to preserve the cross-shaped highlight when using multiple Gaussian elements, even when accessing higher (coarser) levels in the mipmap. As expected, the result from our method using a single Gaussian element are similar to the ones produces with the LEAN method. Also, those tests show the importance of taking more than one iteration of the *GMM*-EM algorithm in the `Fitting` operator. Indeed, the quality of the results completely depends on this, especially when working with a larger number of Gaussian elements in the final *GMM*s.

4.10 Future work

While having explored new fitting methods, *GMM* simplification schemes and different blending strategies, there is a lot of work left to be done regarding reflectance filtering using Gaussian Mixture Models. Here is a short list of key ideas that would be worth investigating in the future:

- **Other statistical models**: we believe that by using more complex parametric distribution models in a similar framework, we could reduce the amount of parameters stored in the mipmap texture as well as on the *grid* in Manuka. 2-dimensional Skewed normal distributions only use two extra parameters but are much more expressive than regular Gaussian distributions. Therefore, we could reduce the number of elements in the mixture while keeping the same accuracy in the representation.

- *GMM* **NDF in current BSDF models**: as a replacement to the stochastic sampling of the Gaussian elements at the light transport phase, we could extend our BSDF models such that they work with more than a single Gaussian distribution as NDF.

- **Optimal Transport Expectation Maximization algorithm**: it would be interesting to derive a fitting algorithm based on the Earth-Moving distance metric introduce in this chapter. It would aim at minimizing the Earth-Moving distance rather than maximizing the log-lokelyhood during the parameters search. This could result in higher quality of the fitting from a reflectance perceptive standpoint.

- **Fitting discrete flakes parameters**: limited by its maximum number of Gaussian elements, the *GMM*s used in our framework still cannot properly represent the distribution of high-frequency detail normal map textures. It would be great to extend our fitting algorithm to approximate the density of those microstructures and feed those parameters to a discrete model like the one introduced in Chapter 3.

Chapter 5

Conclusion

In this thesis, we have presented three different methods aiming at solving different parts of the reflectance filtering problem. The first approach is brand new in the field since autoencoders have never been used before (to this day) to produce a compressed representation of the displacement information that is compatible with texture filtering methods. While it is still far from producing the level of quality needed to be usable in practice, we believe that this solution has a lot of potential and might inspire future research projects in computer graphics. The second method we introduced is a practical implementation of a discrete microfacet model in Manuka. The challenge for this method was to solve technical and engineering problems given the complexity and the architecture of the production renderer. Since this model was lacking in expressiveness, we investigated a third method that extends LEAN/LEADR using Gaussian Mixture Models. We developed an new optimization algorithm for fitting *GMM*s on larger *GMM*s and also suggested a novel simplification scheme for *GMM* based on the Earth-Moving distance. Those two algorithms combined allowed us to write a tool to efficiently generate mipmaps of Gaussian mixtures as a replacement to the LEAN maps. We then looked at different ways of combining sets of *GMM*s that could be used in conjunction with modern texture filtering methods. We put a lot of effort on a type of blending operator that proved to be incompatible with the architecture of our renderer. However, we believe that it could be used as a efficient solution when *GMM*s need to be bilinearly combined. Finally, we illustrated some results using our prototype implementation of this method in Manuka. While still a lot of improvements can be done on this method, we hope that one day in the near future it will replace the LEAN mapping method in Manuka and in other production renderers.

Until then, just keep rendering!

Appendix A

Bilinear blending strategies figures

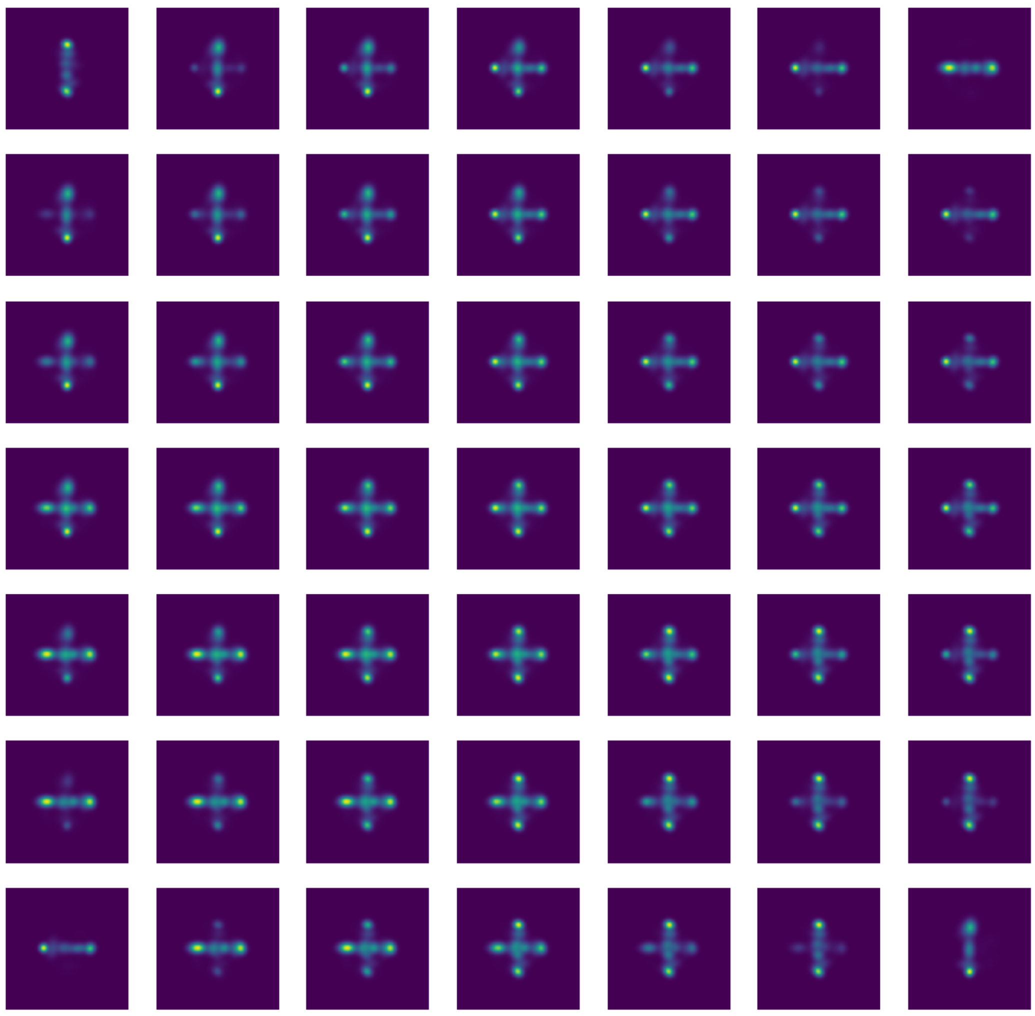


FIGURE A.1: `Blending` operator (ground truth)

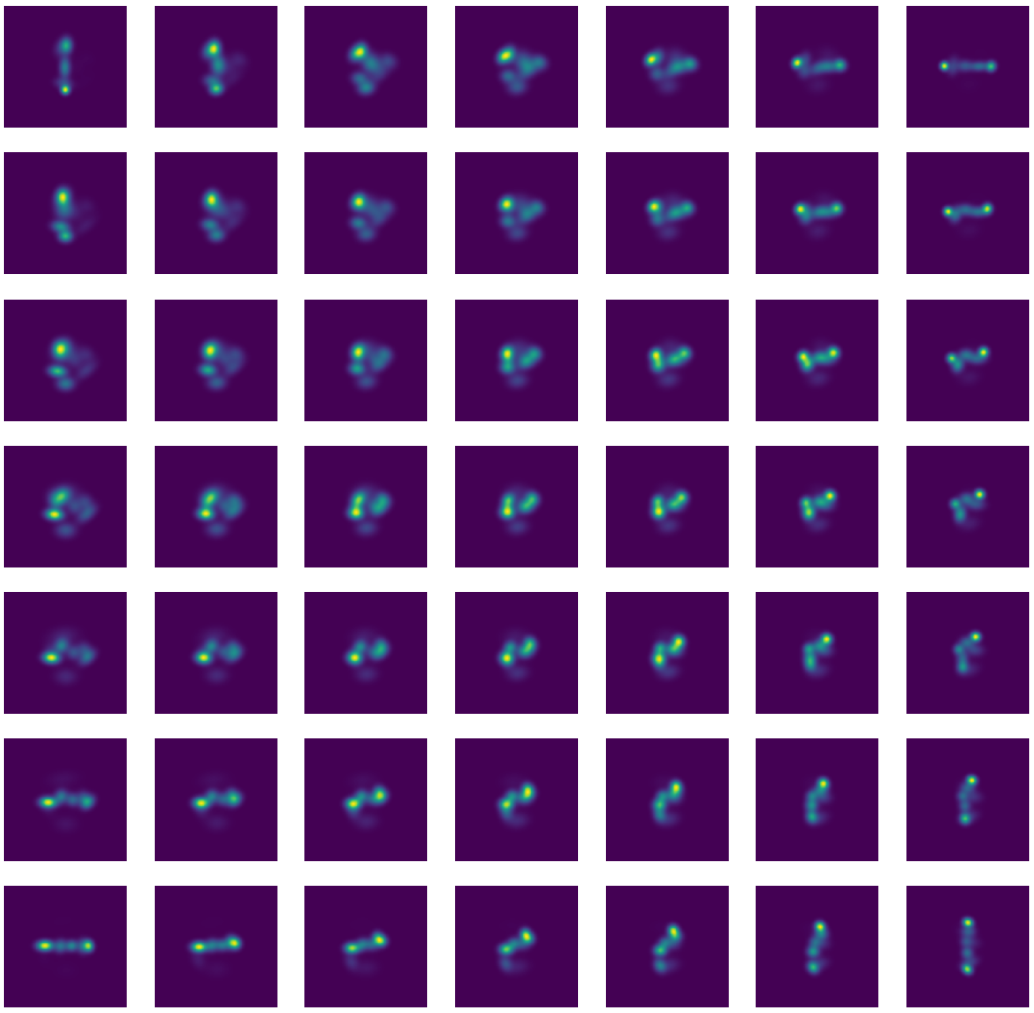


FIGURE A.2: `Interpolation` operator with aligned GMMs

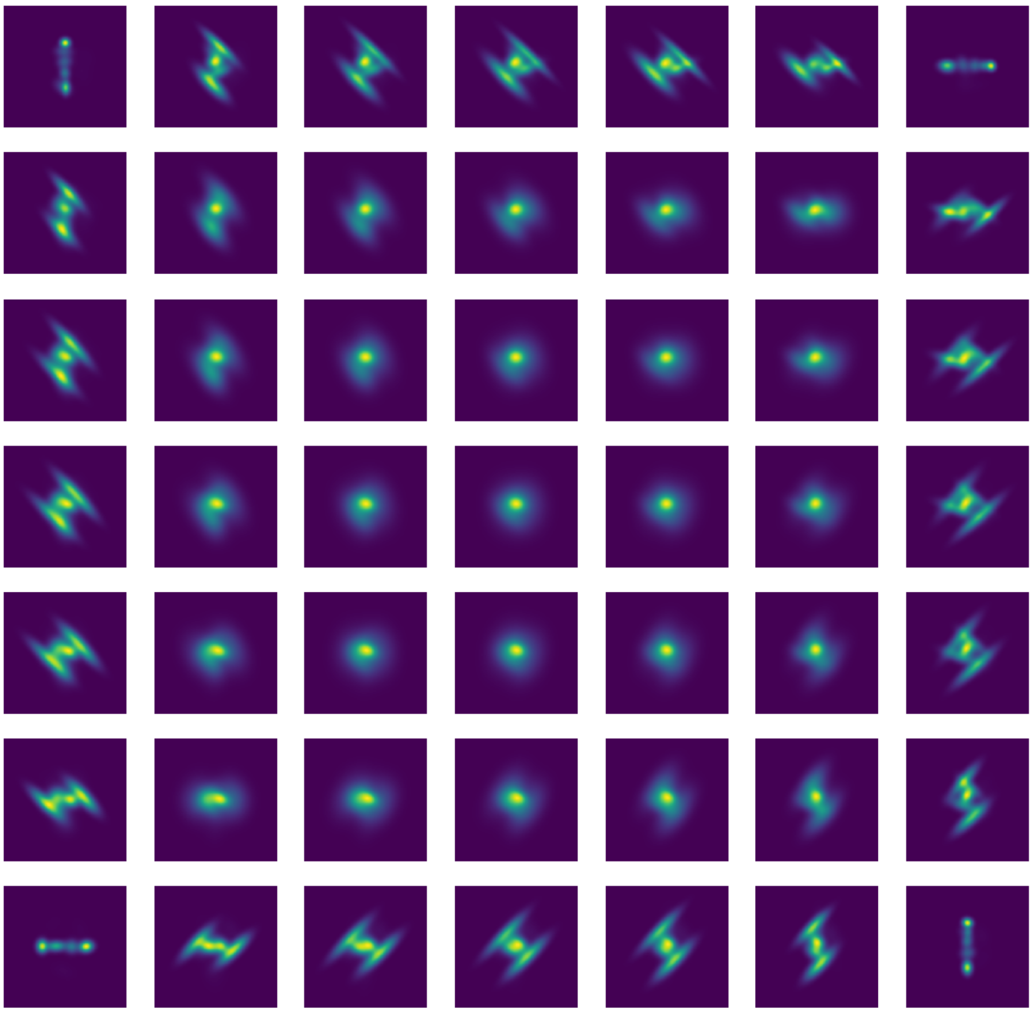


FIGURE A.3: `Filtering` operator with aligned GMMs

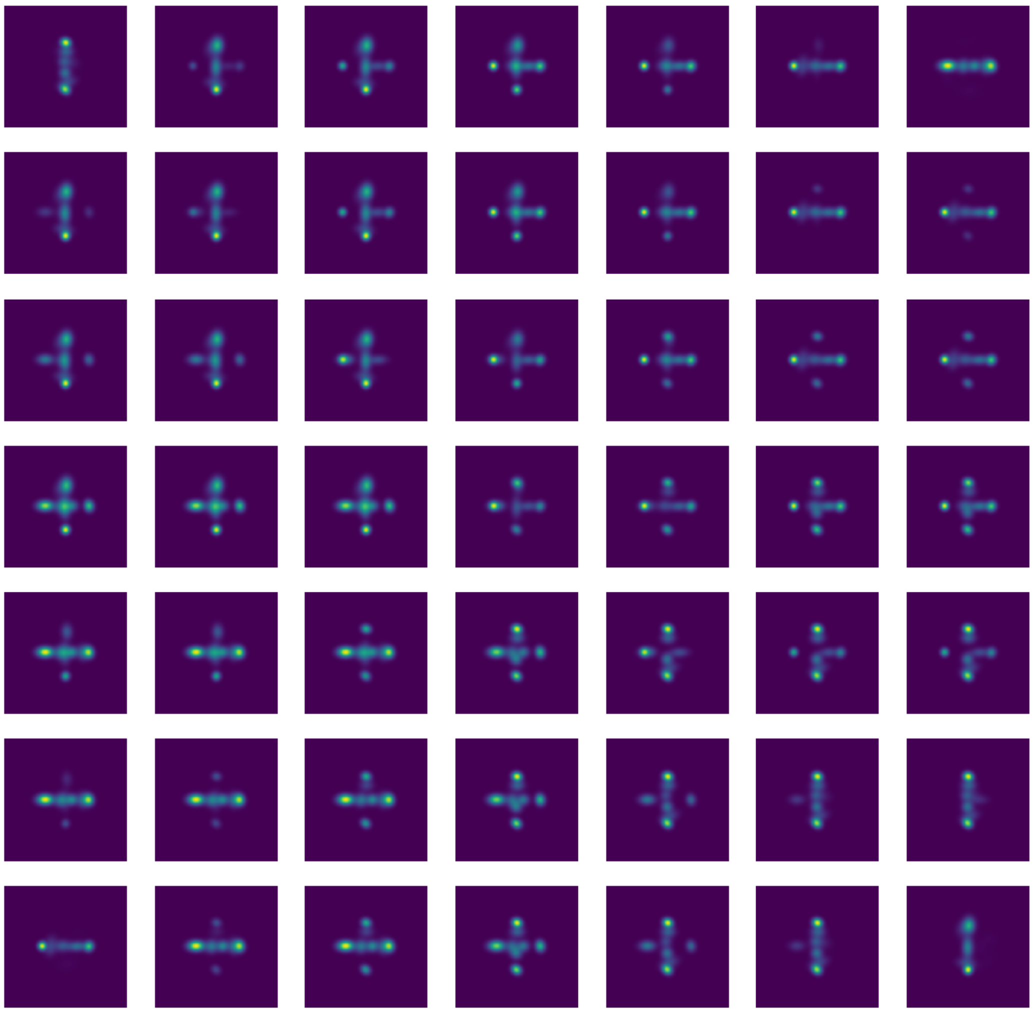


FIGURE A.4: `Max-Blending` operator

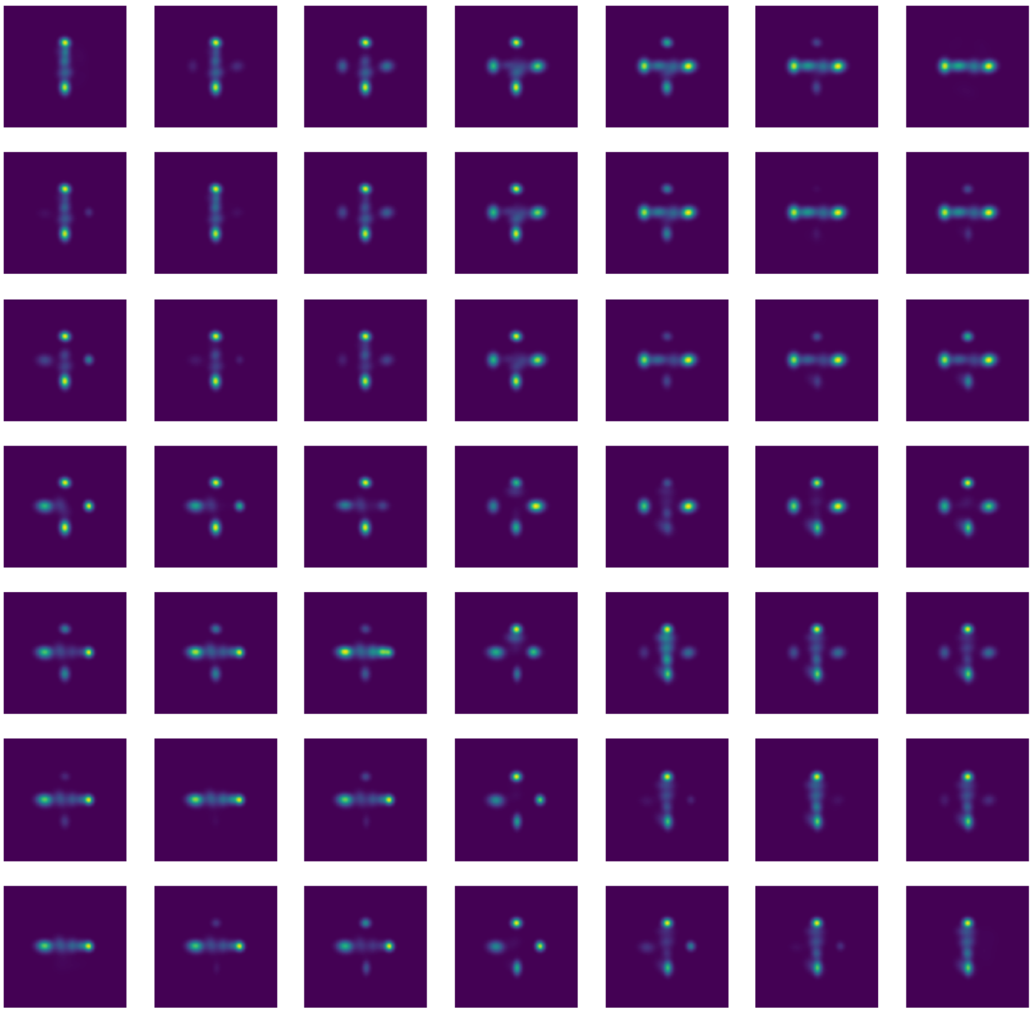


FIGURE A.5: `Smooth-Blending` operator

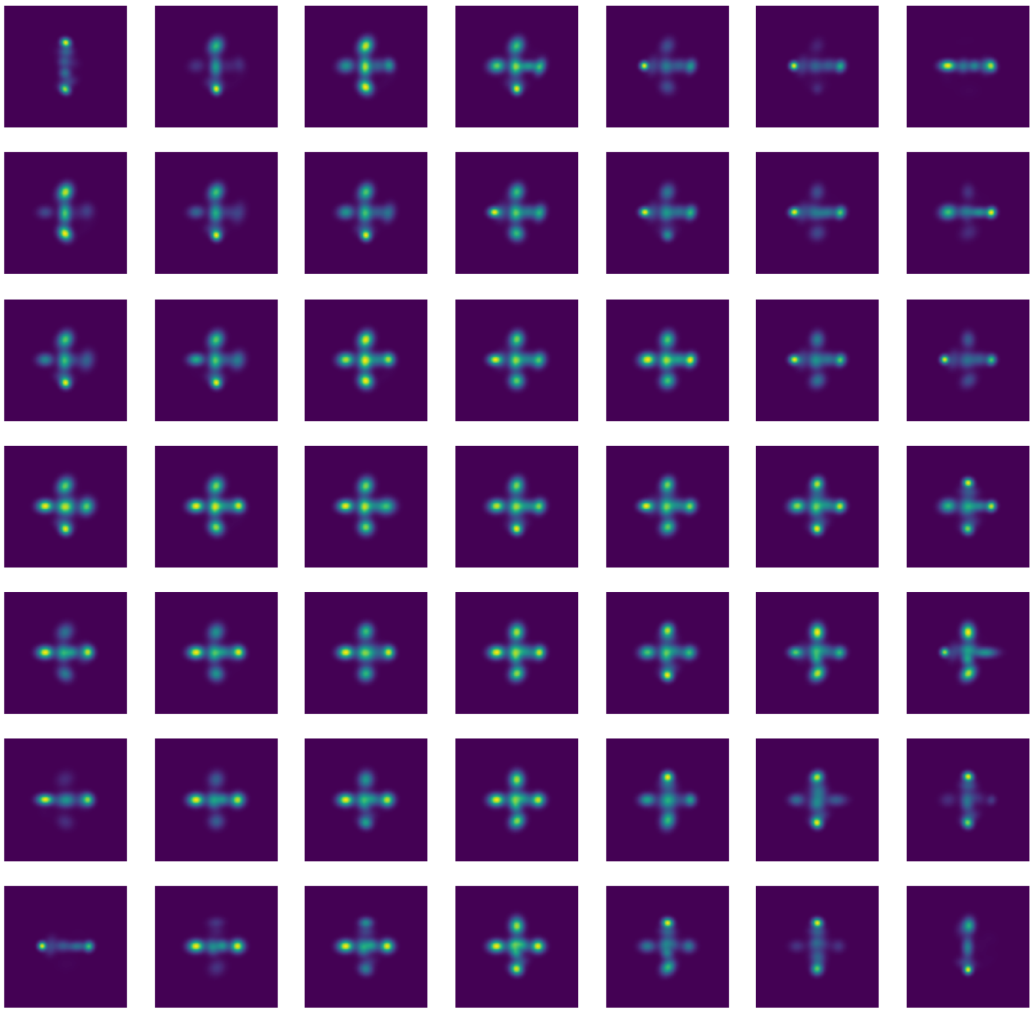


FIGURE A.6: `Fitting` operator

Appendix B

Results of the *GMM* reflectance filtering method

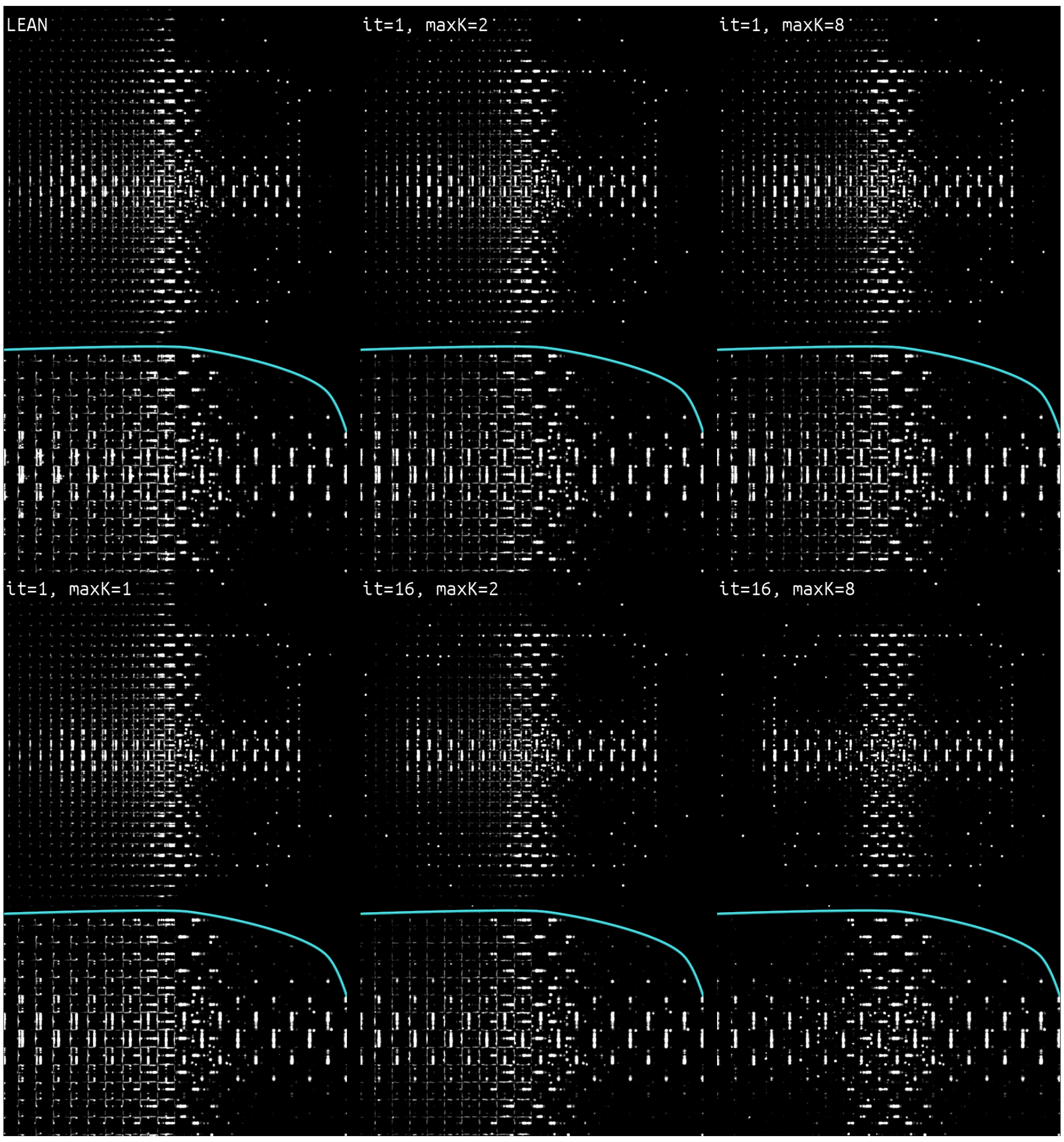


FIGURE B.1: Accessing mipmap level: 0

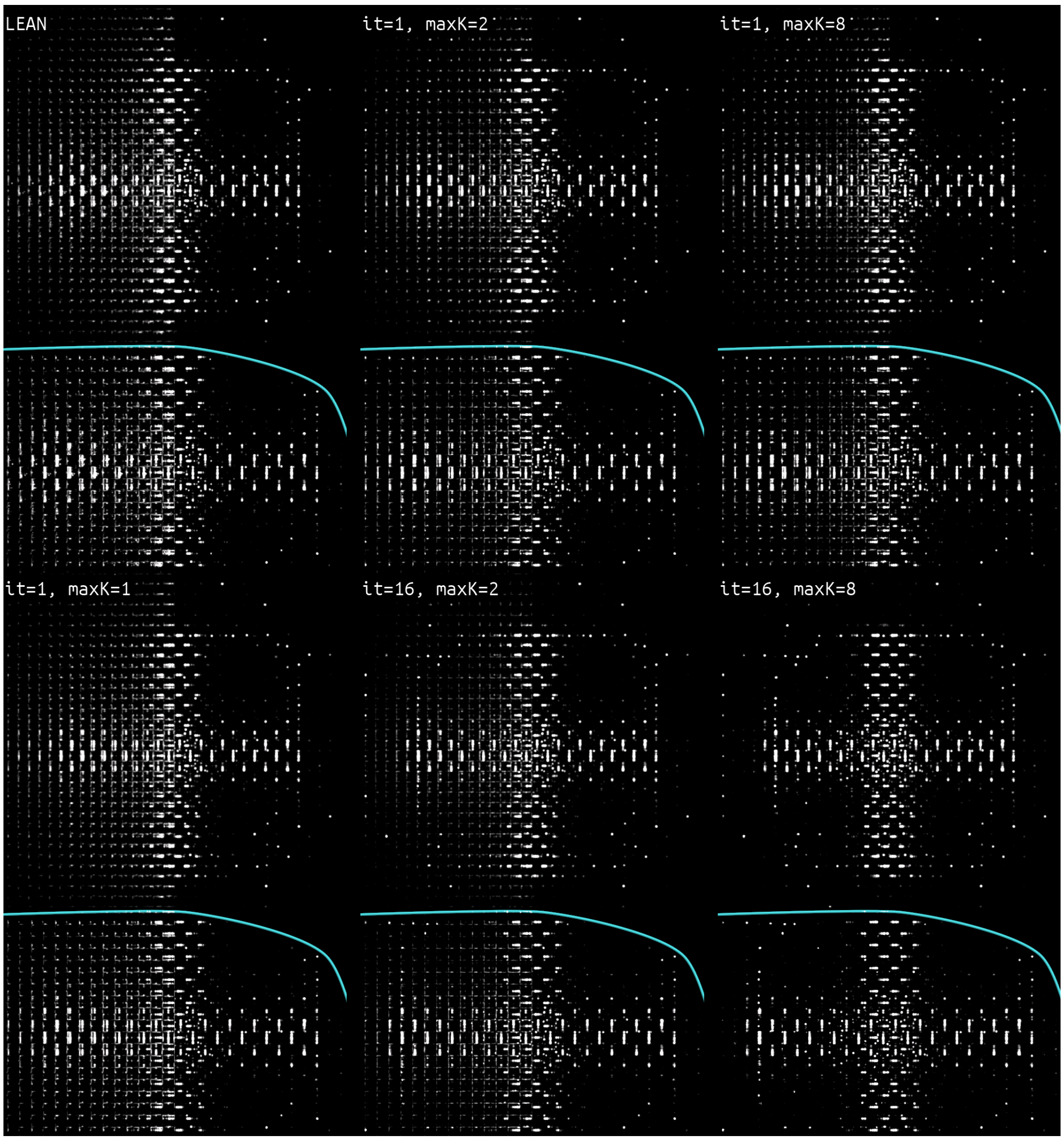


FIGURE B.2: Accessing mipmap levels: 0 & 1

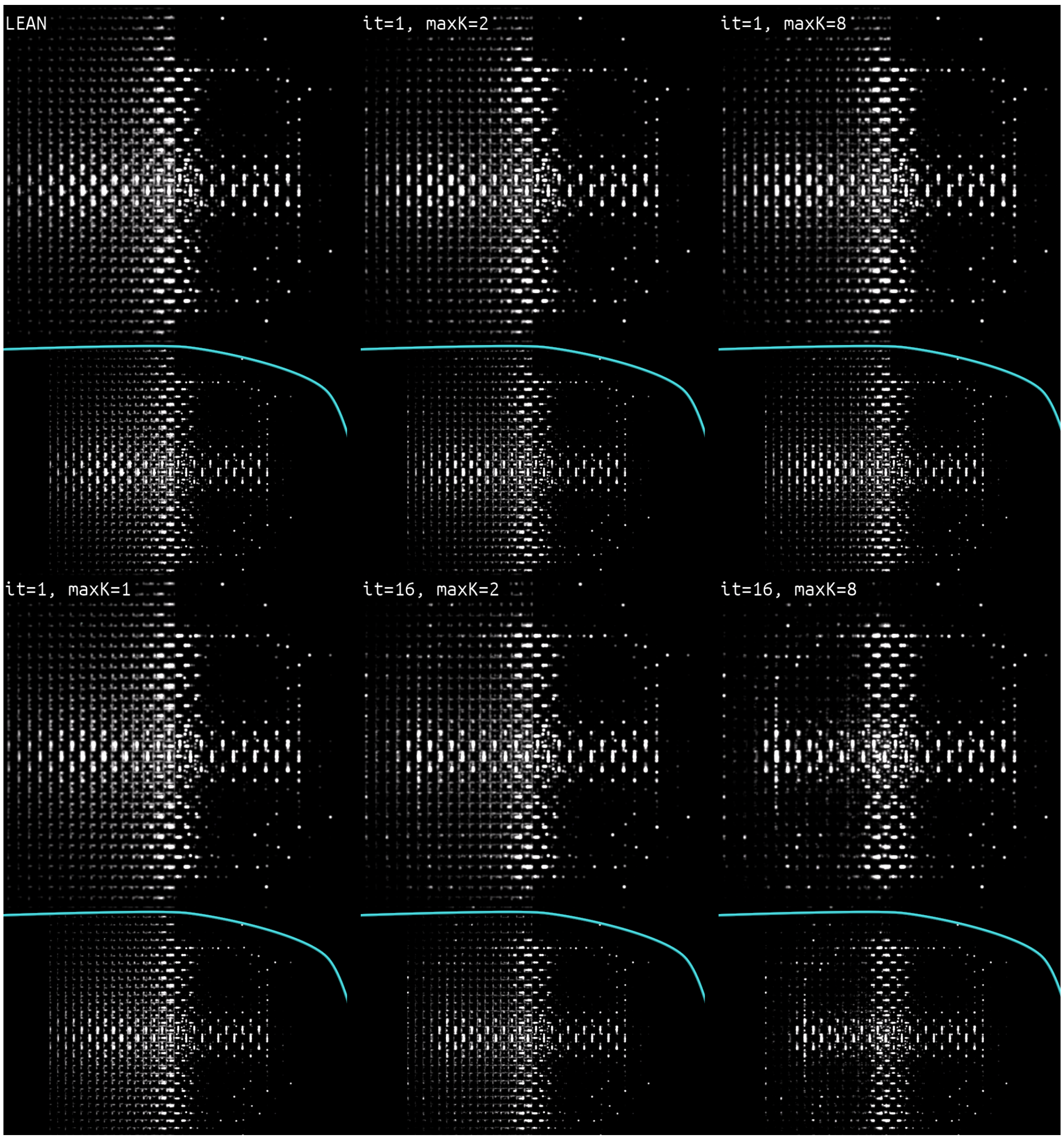


FIGURE B.3: Accessing mipmap levels: 1 & 2

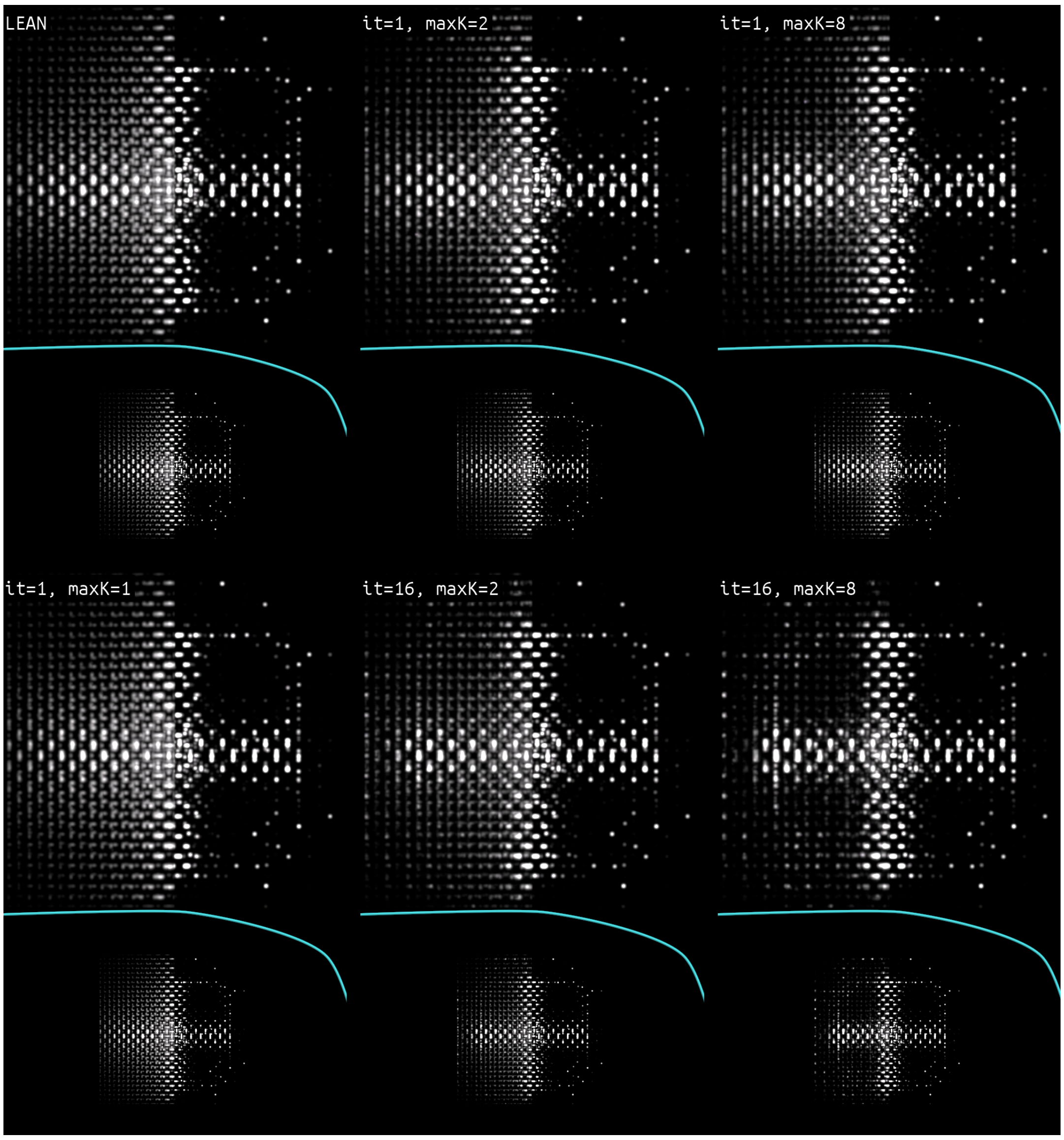


FIGURE B.4: Accessing mipmap levels: 2 & 3

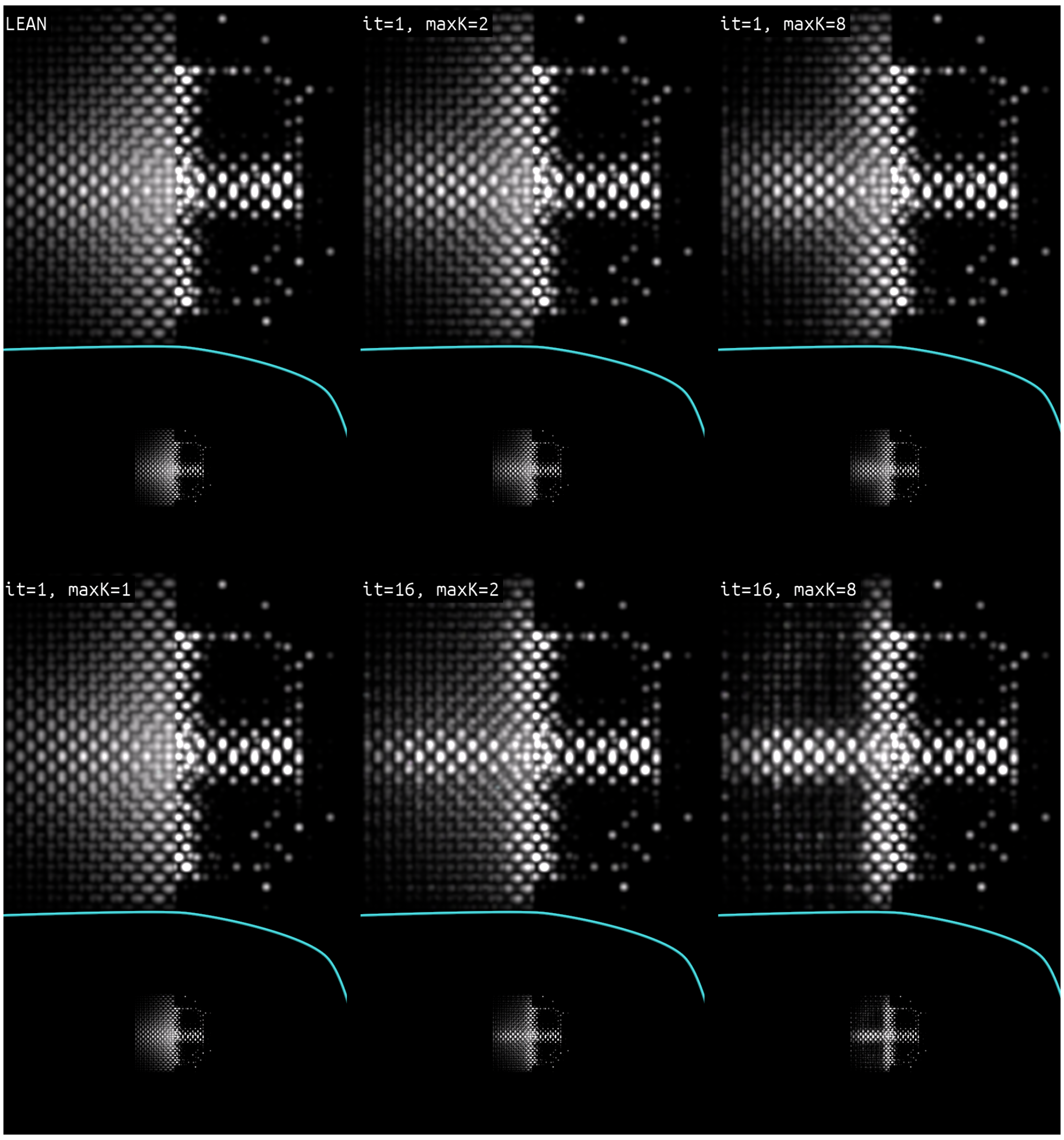


FIGURE B.5: Accessing mipmap levels: 3 & 4

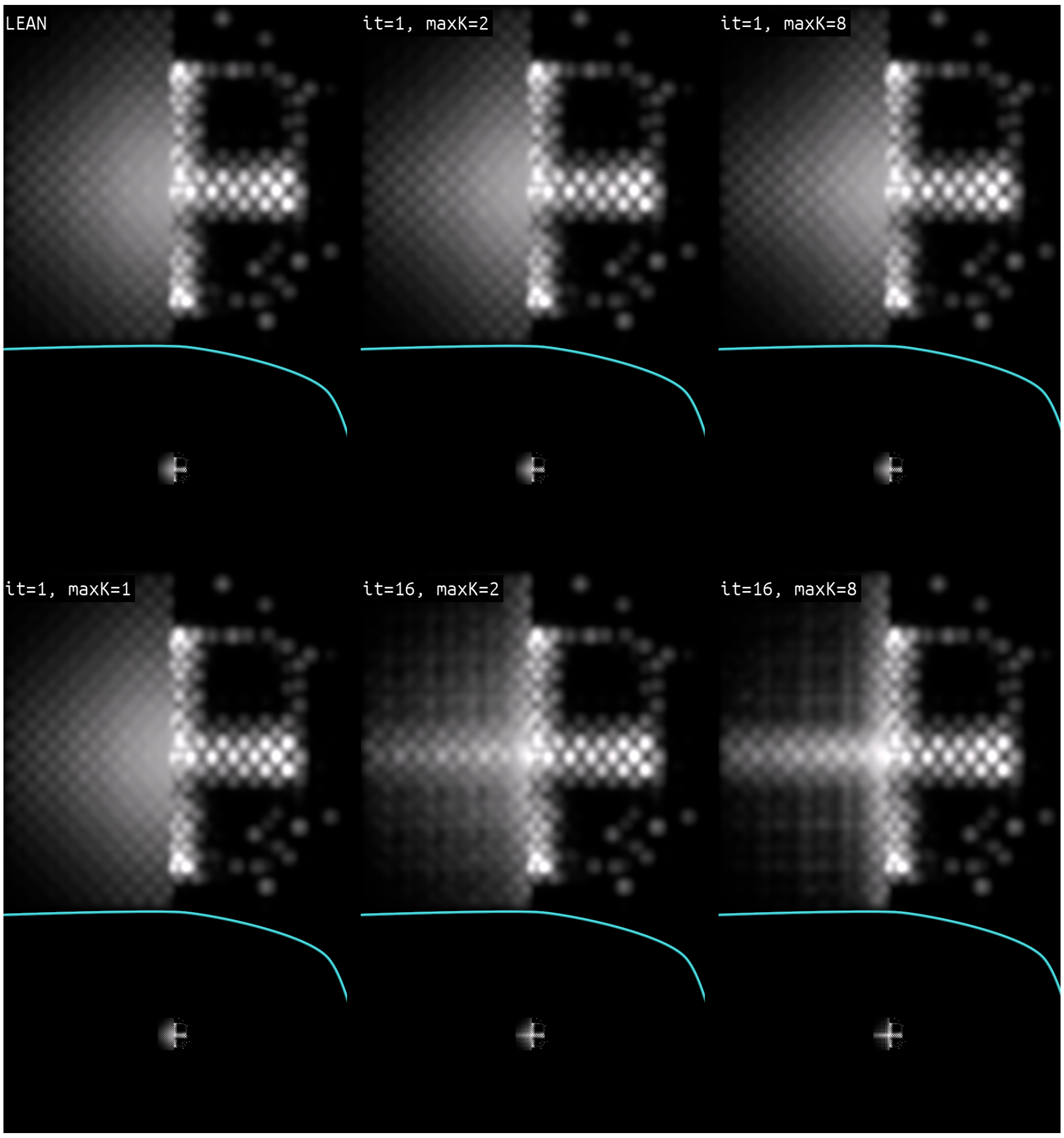


FIGURE B.6: Accessing mipmap levels: 4 & 5

Bibliography

ACM (2018). In: *ACM Trans. Graph.* 37.3. ISSN: 0730-0301.

Arthur, David and Sergei Vassilvitskii (2007). “K-means++: The Advantages of Careful Seeding”. In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '07. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, pp. 1027–1035. ISBN: 978-0-898716-24-5. URL: http://dl.acm.org/citation.cfm?id=1283383.1283494.

Atanasov, Asen and Vladimir Koylazov (2016). “A Practical Stochastic Algorithm for Rendering Mirror-like Flakes”. In: *ACM SIGGRAPH 2016 Talks*. SIGGRAPH '16. Anaheim, California: ACM, 67:1–67:2. ISBN: 978-1-4503-4282-7. DOI: 10.1145/2897839.2927391. URL: http://doi.acm.org/10.1145/2897839.2927391.

Ballard, Dana H. (1987). “"Modular learning in neural networks"”. In:

Blömer, Johannes and Kathrin Bujna (2013). “Simple Methods for Initializing the EM Algorithm for Gaussian Mixture Models”. In:

Christophe Hery Michael Kass, Junyi Ling (2014). *Geometry into Shading*.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). “Maximum Likelihood from Incomplete Data via the EM Algorithm”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 39.1, pp. 1–38. ISSN: 00359246. URL: http://www.jstor.org/stable/2984875.

Dong, Zhao et al. (2015). “Predicting Appearance from Measured Microgeometry of Metal Surfaces”. In: *ACM Trans. Graph.* 35.1, 9:1–9:13. ISSN: 0730-0301. DOI: 10.1145/2815618. URL: http://doi.acm.org/10.1145/2815618.

Dupuy, Jonathan et al. (2013). “Linear Efficient Antialiased Displacement and Reflectance Mapping”. In: *ACM Transactions on Graphics*. Proceedings of Siggraph Asia 2013 32.6, Article No. 211. DOI: 10.1145/2508363.2508422. URL: https://hal.inria.fr/hal-00858220.

Fascione, Luca et al. (2018). “Manuka: A batch-shading architecture for spectral path tracing in movie production.” In:

Garcia, Vincent, Frank Nielsen, and Richard Nock (2010). *Hierarchical Gaussian Mixture Model*.

Han, Charles et al. (2007). “Frequency Domain Normal Map Filtering”. In: *ACM Trans. Graph.* 26.3. ISSN: 0730-0301. DOI: 10.1145/1276377.1276412. URL: http://doi.acm.org/10.1145/1276377.1276412.

Heitz, Eric (2014). “Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs”. In: *Journal of Computer Graphics Techniques (JCGT)* 3.2, pp. 48–107. ISSN: 2331-7418. URL: http://jcgt.org/published/0003/02/03/.

Jakob, Wenzel et al. (2014). “Discrete Stochastic Microfacet Models”. In: *ACM Trans. Graph.* 33.4, 115:1–115:10. ISSN: 0730-0301. DOI: 10.1145/2601097.2601186. URL: http://doi.acm.org/10.1145/2601097.2601186.

Jolliffe, I.T. (1986). *Principal Component Analysis*. Springer Verlag.

Katzin, M. (1964). “The scattering of electromagnetic waves from rough surfaces”. In: *Proceedings of the IEEE* 52.11, pp. 1389–1390. ISSN: 0018-9219. DOI: 10.1109/PROC.1964.3413.

Kiefer, Joe and J Wolfowitz (1952). “Stochastic Estimation of the Maximum of A Regression Function”. In: 23.

Knight, P. (2008). “The Sinkhorn–Knopp Algorithm: Convergence and Applications”. In: *SIAM Journal on Matrix Analysis and Applications* 30.1, pp. 261–275. DOI: `10.1137/060659624`.

Olano, Marc and Dan Baker (2010). “LEAN Mapping”. In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’10. Washington, D.C.: ACM, pp. 181–188. ISBN: 978-1-60558-939-8. DOI: `10.1145/1730804.1730834`. URL: `http://doi.acm.org/10.1145/1730804.1730834`.

Pharr, Matt, Wenzel Jakob, and Greg Humphreys (2016). *Physically Based Rendering: From Theory to Implementation (3rd ed.)* 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., p. 1266. ISBN: 9780128006450.

Raymond, Boris, Gael Guennebaud, and Pascal Barla (2016). “Multi-Scale Rendering of Scratched Materials using a Structured SV-BRDF Model”. In: *ACM Transactions on Graphics*. DOI: `10.1145/2897824.2925945`. URL: `https://hal.inria.fr/hal-01321289`.

Tan, Ping et al. (2005). “Multiresolution Reflectance Filtering”. In: *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques*. EGSR ’05. Konstanz, Germany: Eurographics Association, pp. 111–116. ISBN: 3-905673-23-1. DOI: `10.2312/EGWR/EGSR05/111-116`. URL: `http://dx.doi.org/10.2312/EGWR/EGSR05/111-116`.

Torrance, K. E. and E. M. Sparrow (1967). “Theory for Off-Specular Reflection From Roughened Surfaces∗”. In: *J. Opt. Soc. Am.* 57.9, pp. 1105–1114. URL: `http://www.osapublishing.org/abstract.cfm?URI=josa-57-9-1105`.

Veach, Eric (1998). “Robust Monte Carlo Methods for Light Transport Simulation”. AAI9837162. PhD thesis. Stanford, CA, USA. ISBN: 0-591-90780-1.

Verbeek, Jakob, Jan R. J. Nunnink, and Nikos Vlassis (2006). “Accelerated EM-based clustering of large data sets”. In: 13, pp. 291–307.

Walter, Bruce et al. (2007). “Microfacet Models for Refraction Through Rough Surfaces”. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR’07. Grenoble, France: Eurographics Association, pp. 195–206. ISBN: 978-3-905673-52-4. DOI: `10.2312/EGWR/EGSR07/195-206`. URL: `http://dx.doi.org/10.2312/EGWR/EGSR07/195-206`.

Werner, Sebastian et al. (2017). “Scratch Iridescence: Wave-optical Rendering of Diffractive Surface Structure”. In: *ACM Trans. Graph.* 36.6, 207:1–207:14. ISSN: 0730-0301. DOI: `10.1145/3130800.3130840`. URL: `http://doi.acm.org/10.1145/3130800.3130840`.

Williams, Lance (1983). “Pyramidal Parametrics”. In: *SIGGRAPH Comput. Graph.* 17.3, pp. 1–11. ISSN: 0097-8930. DOI: `10.1145/964967.801126`. URL: `http://doi.acm.org/10.1145/964967.801126`.

Yan, Ling-Qi et al. (2014). “Rendering Glints on High-Resolution Normal-Mapped Specular Surfaces”. In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33.4.

Yan, Ling-Qi et al. (2016). “Position-Normal Distributions for Efficient Rendering of Specular Microstructure”. In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2016)* 35.4.

Yan, Ling-Qi et al. (2018). “Rendering Specular Microgeometry with Wave Optics”. In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2018)* 37.4.

Yongxin Chen Tryphon T. Georgiou, Allen Tannenbaum (2018). “Optimal transport for Gaussian mixture models”. In: