Normal Map Appearance Filtering using Linearly Transformed Cosines

Gaspard Nahmias

School of Computer Science

McGill University, Montreal, QC

October, 2022

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science (M.Sc.) in Computer Science

Résumé

De nombreux matériaux réels comme le plastique, la neige et le métal brossé, sont constitués de facettes microscopiques complexes qui diffusent la lumière. L'apparence de ces reflets haute fréquence est difficile à reproduire fidèlement dans une scène virtuelle, d'autant plus sous un éclairage polygonal. Le rendu d'un seul pixel sous éclairage polygonal nécessite une intégration spatiale sur la projection du pixel et une intégration angulaire de la réflectance à chaque microfacette pour calculer la contribution de la lumière. Un tel calcul est d'un coût prohibitif dans une pipeline graphique en temps réel, et les approximations plus rapides induisent des d'artefacts de crénelage. Nous visons à restituer des surfaces normales détaillées sous un éclairage de lumière polygonale à une vitesse interactive, tout en minimisant le crénelage et en préservant les détails spéculaires. Nous étudions l'utilisation des distributions "Linearly Transformed Cosines" (LTC) pour préfiltrer la carte de normales à différentes échelles et points de vue. Nous tabulons ensuite les données filtrées dans une hierarchie de textures, et les interpolons pour un rendu en temps réel. Bien que notre méthode soit sujette a des pertes d'énergie et du floutage mineur, elle produit des rendus sans artefact de crenelage et préserve les détails de la surface.

Mots clés: rendu en temps réel, rasterisation, anti-crénelage, filtrage de normales, distribution cosinus

Abstract

Many real-world materials like plastic, snow, and brushed metal, are covered in complex microscopic facets which scatter light. The appearance of these high frequency glints is difficult to reproduce faithfully in a virtual scene, even more so under area light illumination. Rendering a single pixel under polygonal illumination requires a spatial integration over the pixel footprint, and an angular integration of the reflectance at each microfacet to compute the contribution of the light. Such a computation is prohibitively expensive in a real-time graphics pipeline, and faster approximations suffer from shading aliasing artifacts. We aim to render detailed normal-mapped surfaces under polygonal light illumination at an interactive speed, whilst minimizing shading aliasing and preserving specular detail. We investigate the use of Linearly Transformed Cosine distributions (LTC) to pre-filter the normal map at different scales and view directions. We then tabulate the filtered data into a mip herarchy and interpolate it in a real-time renderer. Although our method is subject to minor energy loss and overblurring, it produced aliasing-free renders and preserves the normal map details.

Key words: real time rendering, rasterization, numerical methods, antialiasing, normal map filtering, linearly transformed cosine distributions

Preface and acknowledgments

This thesis contains the result of the work I have conducted in 2021 and 2022, between Paris and Montreal. The final experiments were conducted in the office at Ubisoft LaForge Montreal.

Many thanks to Assistant Professor Adrien Gruson for agreeing to be examiner for this thesis, and for his great feedback on the first version. I have taken all his comments in consideration and I hope that it produced a more complete and thorough paper.

I am very grateful for the opportunity offered by my supervisor Prof. Derek Nowrouzezahrai, to explore this domain of research in a hands-on way and to bring me out of my comfort zone. I would like to sincerely thank him for the ideas and the motivation, from the Realistic Rendering course, to our many meetings in the two years that followed.

I would like to thank my mother Axelle who I am dedicating this thesis to. She supported me significantly during my studies, and provided invaluable help in the past year, for which I am very grateful. Of course, many thanks to my father Patrice and my sister Carla who have been very encouraging and are always a phone call away. My adoptive family here in Canada was also incredibly kind and accomodating, and a special thank you to Martine!

I would finally like to express my gratitude to my two supervisors at LaForge, Jean-Philippe Guertin and Shahin Rabbani, whose insightful feedback always helped me stay on the right track. Everyone else that I met at LaForge was very welcoming, and I would like to thank them for making my internship a great time; notably Andrea, Olivier, Karl-Etienne, Damien and Arnaud.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In video games and in motion pictures, realistic image rendering is playing a major role as artists aim to improve the visual appearance of environments, and enhance the immersion for audiences. From a description of the geometry of the scene, of its materials, and of the lighting, the rendering engine computes for each frame the illumination hitting the surfaces and determines the color of each pixel. With the ultimate goal to make virtual images perfectly photorealistic, offline techniques can spend several minutes or hours to render each frame using expensive algorithms, for example by using high numbers of samples. On the other hand, real-time (or online) rendering methods are given a few milliseconds (or less) to display the frame to the screen. Viewers of real time media, like video games and virtual reality, are still expecting some level of visual realism. To achieve such performance, methods aimed at real-time engines avoid pure physical simulation of light, and prefer to make use of mathematical approximations along with GPU optimizations to bring visual realism into an interactive scene. Recent hardware improvements enabling real-time ray tracing bring us closer to interactive photorealism (figure 1.1), but many scene configurations are still immensely challenging to process interactively. Among those, we can mention volumetric media, bright emissive materials, transparency, soft shadows, caustics, and high frequency glints.

Figure 1.1: Image rendered interactively in the Unreal Engine 5, showcasing real time global illumination and volumetric media (Source: https://docs.unrealengine.com/5.0/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/).

This thesis focuses on the real time rendering of specular materials with a complex surface structure of microfacets. Materials of this type are omnipresent in the real world, including snow, glitter, glittery paint, brushed metals, textured plastics, and varnished wood. Their surfaces are composed of microscopic facets, which scatter light and can create a shimmery effect. This effect is most significant when illuminated from a small (or distant) light source, but is still noticeable with larger (or closer) sources. It is also more remarkable when the base material is more specular or shiny (as opposed to rough or diffuse) (figure 1.2). To represent such surfaces in virtual scenes, techniques describe the normal orientation and variation analytically (with statistical methods), or using a normal map texture. The higher the resolution of the normal map, the finer the details we can represent. In opposition to the statistical method, a normal map can convey important macroscopic features such as scratches or grain, which would match with the details on the albedo. Models that represent these surfaces also make use of a Bidirectional Reflectance Distribution Function (BRDF), a spherical function that describes the directions in which

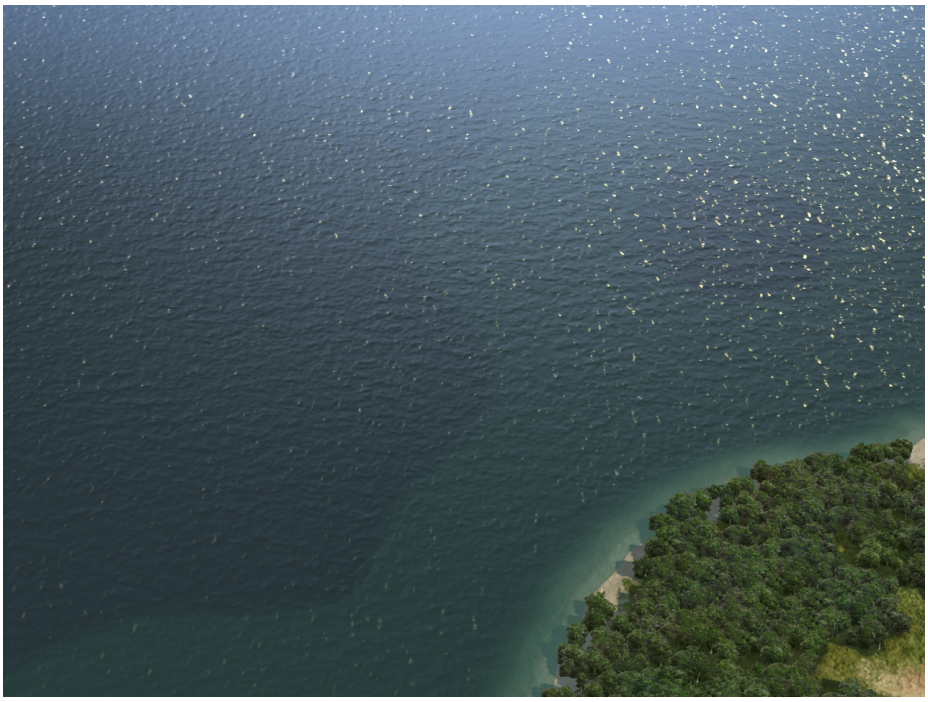


Figure 1.2: Real time render of a water surface illuminated by a distant sun using LEAN mapping [OB10]. We can see both diffuse illumination and specular glints (image taken from [OB10]).

light is scattered when reflecting off a material. Modern models such as GGX can be very elaborate, and have become widespread in realistic rendering.

We aim to render such materials in real time under polygonal light source illumination, while preserving the glinty effect of the specular material, and the normal map details at several viewing scales. To achieve it, we would require both spatial integration and angular integration inside each pixel. Indeed, we first need to check all the surface microfacets that can be seen by the pixel, whose count can range from one to several thousands depending on the display resolution, the view distance and the normal map resolution. Then, for each microfacet, we need to integrate the BRDF on the area covered by the polygonal light source, to compute its illumination.

Moreover, major complications can occur when a pixel projects over many microfacets. Each sample normal within the pixel footprint can yield very different shading results, and we cannot simply take normals and average them. Shading aliasing is a category

of undesirable rendering artifacts occuring when the sampling rate is lower than the frequency of the signal, in the context of computing the reflectance of a surface. This will almost always occur when several normals fall within a pixel footprint, and be manifested as flickering or the loss of small features. When the base BRDF is more specular, or when the light source is small, shading aliasing is more obvious. Aliasing is also more important when the density of texels increase, and it is a very current problem since texture resolutions can get arbitrarily large, while pixel density cannot increase as fast due to hardware limitations.

Thus, the difficulty of preventing aliasing of the higher frequency details on the surface is compounded with the problem of computing real time illumination from an arbitrary area light in real time. To tackle this, we aim to use an appearance pre-filtering technique on the normal map, which should reproduce features of any scale or frequency. In a signal processing framework, we can think of the method proposed here as pre-filtering the normal map at the bandlimit of the final result. The filtered result therefore depends on the number of texels inside the pixel footprint projected over the shading surface, as well as on the width of the main BRDF lobe, and the size/distance of the light source.

We contribute to this problem by presenting a prefiltering step where we fit the effective BRDF (which is the base BRDF convolved with the normal distribution function of the pixel) with a Linearly Transformed Cosine (LTC) distribution, for each view angles and scale. More specifically, our technical contribution is the following:

- A stochastic sampling algorithm for the effective BRDF of a pixel.
- Extension of anistropic LTC fitting in [KHDN22] to the effective BRDF of a normal map region, for tabulated view angles and scales. Filtering of the specular component and diffuse component separately.
- Runtime bilinear software interpolation of LTC matrices on top of MIP hardware trilinear interpolation.

The prefiltering ensures shading anti-aliasing, and once we render the surface at runtime, we only have to sample the filtered result once per pixel. The use of (LTC) allows us to integrate analycally over the area light. Our results show that we succeed in developing a technique that deliver aliasing-free and time-stable renderings of such surfaces under polygonal illumination. However, the two main limitations of our technique are the lack of high energy glints when the area light is small, and overblurring at grazing view angles.

1.1 Thesis overview

This thesis is divided into the background and related works chapter, the method chapter, the results, and finally the discussion, future works and conclusion. In the first part, we lay an overview of the fields related to this thesis, the works that came previously and that led to our research problem. We also provide the mathematical background required to understand our contribution. The method chapter is divided in two parts. The first part shows an overview of our contribution and the design decisions behind our algorithm. The second part, presented in a literate programming style, contains a detailed description of our technique, the tools used and the experiments conducted to evaluate the performance of our approach. It serves as documentation for the code base developed for that project, and shows the full implementation. In the results section, we provide measures and visual comparisons between the reference, the baseline method and our renders. Finally, we discuss the significance of the results, the improvements to bring to our technique, and future avenues of research.

Chapter 2

Background and related works

This chapter addresses related works and mathematical tools, useful to understand the method we developed. We first consider light transport and modeling surface illumination virtually; then we look at methods which, through approximations, allow us to render realistic scenes in real time; and finally we review filtering techniques designed to prevent shading aliasing.

2.1 Light transport and reflection

We are interested in modeling and approximating light when it hits a material on the surface of a 3D mesh, and reflects back into the eye or the camera. There are several models that describe light at some level of physical accuracy, and that have become ubiquitous. Spectral models, for example, use the spatio-temporal properties of light to display behavior such as dispersion and scattering. We favor geometric optics, a simpler model where light rays travel in straight lines in a vacuum, until they are reflected or transmitted through a material.

2.1.1 Shading point and coordinate system

A shading point is a point at the surface of a 3D mesh on which we need to perform lighting computations. To do this, a spherical coordinate system is usually defined, to parametrize the directions to and from a shading point x. The set of directions visible from x is a hemisphere based around the normal of the surface at x. We refer to it as the Ω_+ space, and we parametrize it using spherical coordinates: an elevation angle θ and an azimuthal angle φ (figure 2.1). The shading point can also be seen as a differential area of the shading surface (dA).

For the purpose of integrating energy on a spherical domain, we introduce the solid angle $d\omega$, which is the three dimensional equivalent of the planar angle. We define the solid angle subtended by a surface as the surface area of the surface's projection onto the unit sphere. The unit of the solid angle is steradians (sr).

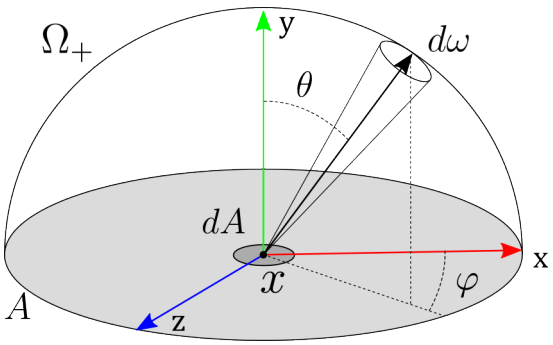


Figure 2.1: Spherical coordinates around the shading point.

We also introduce the projected area of a surface onto a plane, which is the product of the surface's area A and the cosine of the angle between the two surface's normals $\cos(\theta)$.

2.1.2 Radiometry

In light transport simulation, the important radiometric quantities to compute light energy are the irradiance and the radiance:

The irradiance,

$$E(x) = \frac{d\Phi}{dA(x)}, \tag{2.1}$$

is the radiant flux Φ per unit area dA received by a surface from all directions. It is measured in W/m^2.

The radiance,

$$L(x, w) = \frac{d^2\Phi}{d\omega\, dA(x)\, \cos(\theta)}, \tag{2.2}$$

is the light intensity reflected, received or emitted by a projected unit area $dA\, \cos\theta$, in the differentiable solid angle $d\omega$ of direction ω. It is measured in $W/m^2/sr$. In this case, θ is the angle between ω and the normal to the surface (figure 2.1).

The radiance reaching x from a given direction is called the incident radiance, and the radiance leaving x towards a given direction is the exitant/outgoing. In our geometric model, the radiance is invariant along a ray, since we assume that light travels in a vacuum and does not encounter a participating medium.

2.1.3 The Bidirectional Reflectance Function

The behavior of a ray of light interacting at the shading point is characterized by the Bidirectional Reflectance Function (BRDF). It is a simple model of surface reflectance, as it assumes that all incident light is either absorbed or reflected, and it does not account for subsurface scattering [Noe99].

Let's consider L_i, the radiance incident to x from direction ω_i, and dL_r the exitant radiance from x in direction ω_r, considering only the energy from L_i (figure 2.2). We call BRDF the ratio of dL_r to L_i scaled by a foreshortening term $\cos(\theta_i)$, where θ_i is the angle between the surface normal and ω_i:

$$f(x, \omega_r, \omega_i) = \frac{dL_r(x, \omega_r)}{L_i(x, \omega_i)\, \cos(\theta_i)\, d\omega_i}. \tag{2.3}$$

Thus, for a given incident direction, the BRDF is defined on Ω_{+^2} with real positive values everywhere ($\mathbb{R}+$).

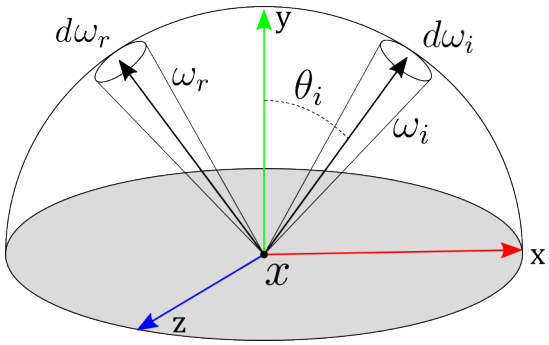


Figure 2.2: Incident and reflected radiance.

Since the reflection is a linear relation, the total light energy reflected by a surface in ω_r can be computed from an integration over the hemisphere of all radiance L_i. The terms can therefore be rearranged in the form of the reflectance equation (scattering equation),

$$L_r(x, \omega_r) = \int_{\Omega_+} f(x, \omega_r, \omega_i)\, L_i(x, \omega_i)\, \cos(\theta_i)\, d\omega_i. \tag{2.4}$$

The product of f and $\cos(\theta_i)$ is also known as the transfer function.

2.1.4 Properties of the BRDF

A BRDF is a specific spherical function following two interesting properties [Noe99].

First, the Helmholtz reciprocity principle (2.5) states that the value of the BRDF is identical if the incident and outgoing directions are swapped, for any direction on the hemisphere. That is,

$$\forall (\omega_i, \omega_r) \in \Omega_{+^2} f(x, \omega_r, \omega_i) = f(x, \omega_i, \omega_r). \tag{2.5}$$

Then, the BRDF follows the rule of conservation of energy (2.6), meaning that for any incoming direction, the integral of the foreshortened BRDF over all reflected directions is at most 1:

$$\forall \omega_i \in \Omega_+, \int_{\Omega_+} f(x, \omega_r, \omega_i)\ \cos(\theta_i)\, d\omega_r \leq 1. \tag{2.6}$$

This ensures that no energy is created when light reflects off the surface.

2.1.5 Types of BRDF

Isotropy

BRDFs can be either isotropic or anisotropic. In the more general anisotropic case, the reflection behavior of the material depends on the rotation of the surface around its normal (or on the relative incident azimuth of the light). This means that to capture the reflectance of the material comprehensively, we need to measure the reflected light for all incident light directions (all azimuth and elevation values). This is the case for materials with a specific microstructure such as polished materials and brushed metals. An isotropic BRDF, on the other hand, shows an identical reflection behavior regardless of the rotation of the material around its normal. We can therefore capture all of its reflectance patterns by choosing an arbitrary azimuth for the incident light, and varying its elevation. For concerns of storage space, the base BRDFs we will consider are isotropic.

Empirical BRDFs

Empirical BRDFs abide by the properties cited in the previous section, but they are not based on physical models. Among them, we can note the perfect mirror, the Phong and Blinn-Phong models, and Ward (for anisotropic BRDFs). The Lambertian BRDF (2.7), or perfect diffuse BRDF, is widely used for its simplicity, with only one constant parameter $\rho \in [0, 1]$ determining the reflectivity of the surface:

$$f(x, \omega_r, \omega_i) = \frac{\rho}{\pi}. \qquad \text{Lambertian BRDF} \tag{2.7}$$

In general, empirical BRDFs are simple to compute, but the disadvantage is that their parameters have no direct correspondence to physical phenomena, which makes it difficult to find the parameters that make the analytic BRDFs behave like real observed materials.

Physically based BRDFs

Physically based BRDFs such as Cook-Torrance and Beckmann, are based on the optical properties of real materials (unlike empirical BRDFs). They are microfacet reflectance models, which assume that the surface is covered in microscopic mirror-like facets. These models rely on aggregate statistical formulations of the normal distribution function (NDF) of the surface specular microfacets, we call D.

Today's physically based shading models largely rely on the Trowbridge–Reitz (GGX) microfacet BRDF [TR75] [WMLT07]. It is more accurate than previous models for rough transmissive materials, and generally considered the most realistic parametric BRDF [HMB+15]. Like the Cook-Torrance BRDF, GGX is composed of a geometric attenuation term G, a fresnel term F, and normal distribution D (2.8), such that

$$f(x, \omega_r, \omega_i) = \rho_d + \rho_s \left(\frac{D(x, \omega_h)\, F(\omega_i \cdot \omega_h)\, G(\omega_i, \omega_o)}{4\,(n \cdot \omega_i)(n \cdot \omega_o)} \right). \tag{2.8}$$

The isotropic GGX will be used as the base BRDF for our project.

Measured BRDFs

We could finally mention measured BRDFs, which require an experimental setup to measure and sample, and describe the reflectance behavior or real-world materials. They are presented in the form of tabulated measurements, for many different viewing or light angles.

2.1.6 Types of illumination models

The reflectance equation (2.4) allows us to compute the reflected light distribution at a shading point, given the distribution of incident light and a BRDF. It is therefore crucial to obtain an accurate representation of the incident light at the shading point. There are two types of models we can use for this.

Global illumination models take into account illumination from indirect light as it bounces around the scene. They are an important tool to reach photorealism, and modern GPUs are now able to compute some levels of global illumination in real time (figure 2.3).

Local/direct illumination models only take into account light sources and their interaction with surfaces. We choose to use a simple model of unshadowed direct illumination: it is the direct interaction between a light source and a surface, and we do not compute shadows cast from this light.

Figure 2.3: Global illumination scene from Assassin's Creed Unity rendered in real time in the AnvilNext graphics engine, using a baked light map (Source: https://www.nvidia.com/en-us/geforce/news/assassins-creed-unity-graphics-and-performance-guide/).

2.1.7 The rendering equation

The rendering equation, introduced in [Kaj86], is the fundamental equation to solve in light transport simulation. This equation describes how to compute, at any shading point x, the outgoing radiance L_o towards direction w_r (2.9). It is the sum of the emitted radiance L_e and the reflected radiance L_r towards this direction (described in (2.4)):

$$L_o(x, \omega_r) = L_e(x, \omega_r) + L_r(x, \omega_r). \tag{2.9}$$

In global illumination, we would consider the outgoing radiance from a shading point to other points in the scene, but from now on we work in a direct illumination framework. Thus, the only outgoing radiance direction we compute at a shading point is towards the eye, and ω_o refers to the view direction. Furthermore, non-emissive materials, which do not generate light, have a L_e term of zero : $L_o = L_r$.

As shown in (2.4), the total reflected radiance towards ω_r can be obtained by integrating the BRDF f over the hemisphere of incident directions ω_i. We simply substitute L_r in (2.9) by (2.4), and obtain the full rendering equation in its solid angle form,

$$L_o(x, \omega_r) = L_e(x, \omega_r) + \int_{\Omega_+} f(x, \omega_r, \omega_i)\, L_i(x, \omega_i)\, \cos(\theta_i)\, d\omega_i. \tag{2.10}$$

2.1.8 Monte Carlo integration

Monte Carlo (MC) integration is one of the most powerful tools commonly used in light transport algorithm design: it is a method that uses random sampling to estimate the values of integrals, such as the value of the rendering equation over the visible hemisphere [Vea98].

To estimate the value of an integrand over a space, MC throws random samples from a distribution with respect to a solid angle that has a known Probability Density Function p,

and computes an evaluation of the integrand at those points:

$$L_o(x,\omega_r) = \int_{\Omega_+} f(x,\omega_r,\omega_i)\, L_i(x,\omega_i)\, \cos(\theta_i)\, d\omega_i \tag{2.11}$$

$$\approx \frac{1}{N}\sum_{j=1}^{N} \frac{f(x,\omega_r,\omega_j)\, L_i(x,\omega_j)\, \cos(\theta_j)}{p(\omega_j)}. \tag{2.12}$$

On average, MC gives the correct estimate for the integrand, and given an infinite amount of time (and samples), it would converge to the true result. A great property of MC integration is that it does not suffer from the curse of dimensionality: the convergence rate of the estimator is independent of the dimensionality of the integrand.

The probability distribution p of the random samples has an important effect on the variance of the MC estimator. Indeed, if the samples are chosen carefully, so that p approximates the true integrand, the estimator will converge faster on average. Of course, the integrand is unknown because it is the quantity we are trying to approximate, but we can have good ideas about some of its components. A very popular family of techniques, called importance sampling [Vea98], throws samples according to known terms of the integrand such as the BRDF and the incident radiance L_i. Veach also introduces Multiple Importance Sampling (MIS), in which some samples are taken from one distribution, and some samples are taken from another [Vea98].

2.1.9 Polygonal light illumination

Point light illumination supposes that light is emitted from an infinitely small point in space, so incident radiance reaching x is from a single direction ω_L. From (2.4), we obtain

$$L_r(x,\omega_r) = \int_{\Omega_+} f(x,\omega_r,\omega_i)\, \delta(\omega_i - \omega_L)\, L_i(x,\omega_i)\, \cos(\theta_i)\, d\omega_i \tag{2.13}$$

$$= f(x,\omega_r,\omega_L)\, L_i(x,\omega_L)\, \cos(\theta_L). \tag{2.14}$$

It is a purely theoretical situation, used as a fast approximation when the light source is small or very far away.

Spherical light and polygonal lights are closer to what we could encounter in reality, and are commonly used lighting models. Estimating the total outgoing radiance $L_o(x, \omega_r)$ requires solving the reflectance equation (2.4) over directions ω_L in the solid angle subtented by the polygon P, also called polygonal domain. The integral is expressed as

$$L_o(x, \omega_o) = \int_P f(x, \omega_o, \omega_L)\, L_i(x, \omega_L)\, \cos(\theta_L)\, d\omega_L. \tag{2.15}$$

Even unshadowed area light illumination is much more challenging to compute in real time than shadowed point light illumination, since the integration usually cannot be solved in closed form.

We are more specifically interested in constant (or uniform) polygonal light illumination. This means that the radiance emitted by the polygon is the same in all directions ω_L, and is the same over the whole surface of the light. In other words, the radiance to the shading point is constant inside the solid angle subtented by the polygon, no matter the orientation of the shading surface or the light ($L_i(x, \omega_L) = L$). The radiance term can therefore be left out of the integral in equation (2.15), which yields

$$L_o(x, \omega_o) = L \int_P f(x, \omega_o, \omega_L)\, \cos(\theta_L)\, d\omega_L. \tag{2.16}$$

However, even with the simplest BRDFs, (2.16) can be very challenging to compute. Modern physically based models of BRDFs are sophisticated, having to support anisotropy and skewness, and their spherical integration over a polygonal domain is often undefined [HDHN16]. This is where Monte Carlo sampling techniques shine, such as Multiple Importance Sampling (section 2.1.8). With enough samples and time, the Monte Carlo estimate of the integral over the polygonal light converges towards ground truth, but it is either too slow or too noisy for our real-time application.

2.2 Interactive rendering techniques

We now cover real time techniques, where an approximation of the rendering equation integral (2.10) can be computed quickly. We give an overview of the real-time rendering pipeline, and we review analytic lighting techniques which can yield noise-free images under broad radiance (radiance coming from an area light or from the environment). Specifically, we look at Image Based Lighting, which models radiance coming from the whole domain of Ω_+, and we investigate real time polygonal illumination using Linearly Transformed Cosines [HDHN16].

2.2.1 The real-time rendering pipeline

In a conventional real-time graphics pipeline, data such as textures, shaders and geometry are first copied from system memory (RAM) to video memory. The Graphics Processing Unit (GPU) has quick access to the video memory, and it runs operations on its own cache memory [Fer04] (figure 2.4). More specifically, it is able to process scene vertices in parallel, and screen pixels in parallel through the use of shaders.

Vertex shaders are small programs that run in parallel on the chip of the GPU, to process each vertex in the scene and apply the appropriate transformations. The scene then undergoes rasterization, which has become the standard technique in real time rendering. For a thorough explanation of the rasterization steps, please refer to section 3.2.2 in the method chapter.

The real-time rendering pipeline relies on GPU hardware architecture specialized for this type of computationally intensive and highly parallelizable problem. More recently, new frameworks for general-purpose computing on GPUs (GPGPU) allowed the graphics programmers to leverage their general computational capability, in particular for large linear algebra operations. Notably, the CUDA platform and programming language enables the development of programs capable of running on thousands of GPU cores in parallel, but not tied to the fixed graphics pipeline [Kir07]. These frameworks have found

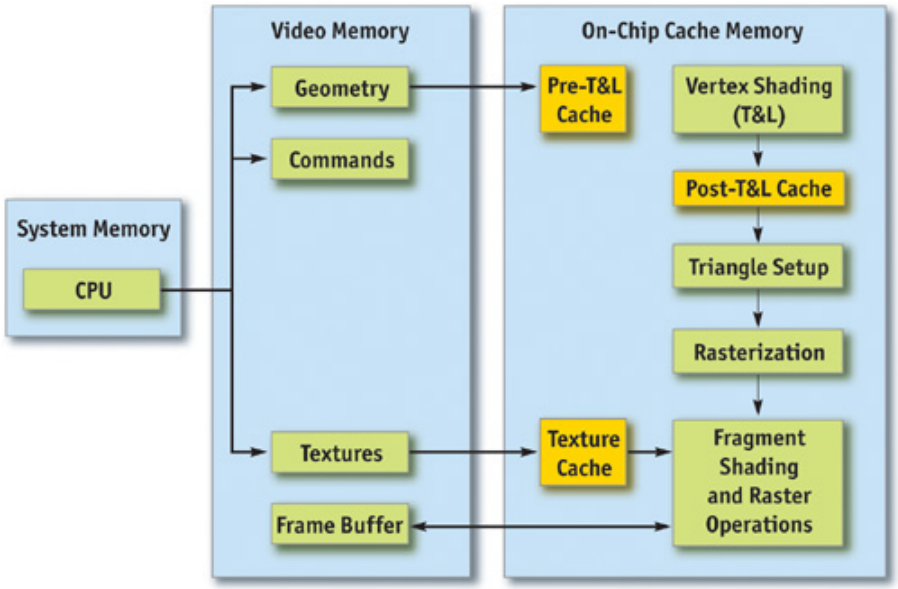


Figure 2.4: Real-time graphics pipeline and the transfers between system memory and GPU-accessible memory (Source: [Fer04]).

widespread adoption in applications beyond computer graphics such as deep learning and simulation. We use the CUDA plaform to leverage the parallel processing capabilities of the GPU, and we will not use conventional shaders of the real-time pipeline.

2.2.2 Pre-computation

The real-time rendering pipelines used video game engines allow for a limited time budget for vertex and fragment shaders (a few milliseconds per frame). Interactive rendering algorithms can therefore sometimes sacrifice storage space, to ensure that each frame can be rendered fast. Recent improvements in graphics hardware have even made Monte Carlo and ray-tracing techniques possible in real time, for example using reservoir sampling [BWP+20]. While such real-time stochastic approaches can be powerful for arbitrary materials, stochastic approximation of integration over broad incident radiance still leads to noisy results.

To leave the least amount of work for the real time/runtime part, a widespread strategy in real time lighting algorithms is to pre-compute a part of the rendering integral

offline, with slower algorithms, such as fitting, sampling and antialiasing operations. This can be done once and for all, or each time a scene is loaded, etc. The precomputed results can then be stored in a texture or a file, to be fetched at runtime, e.g., [Wil83][HSRG07][Kar13][HDHN16].

For instance, Light Mapping is a common technique used in video games (figure 2.5). It consists in pre-calculating the irradiance on the surface of static objects, and storing it in texture maps to use at render time. This only works for diffuse materials, as the Lambertian BRDF is view independent.

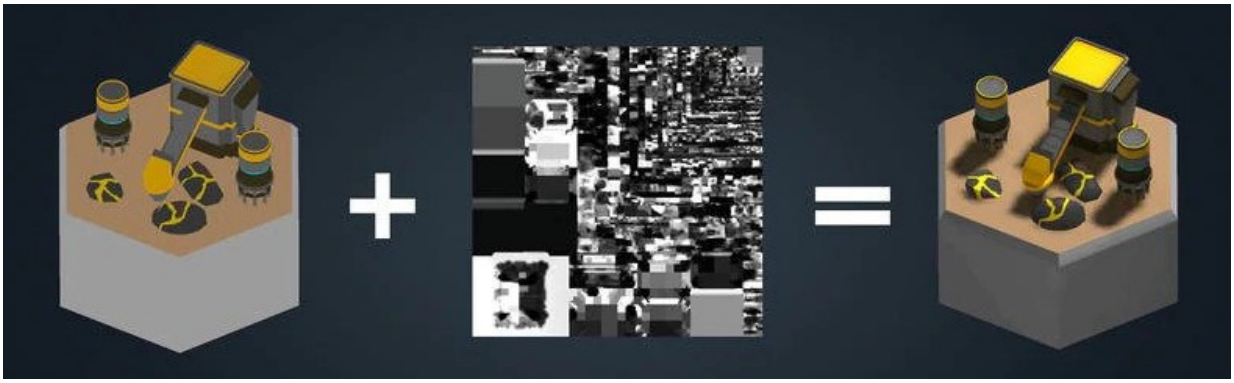


Figure 2.5: Unlit scene (left), light map (center), lit scene (right). Source: https://unity.com/how-to/advanced/optimize-lighting-mobile-games.

2.2.3 Environment maps and image based lighting

Environment mapping is a widely used category of techniques used to model radiance coming to the shading point from all directions on the sphere. Maps can be stored in a set of 6 textures forming a cube, or in spherical coordinates. With Reflection Mapping [BN76], textures are used to create real time reflections from a distant environment.

Environment maps can also be helpful for inexpensive image-based lighting (IBL) [Deb03], where the image represents the emitted radiance of each point in the environment towards the shading point (figure 2.6). Real time IBL techniques usually require precomputing an irradiance map from real photographs or from a capture of the virtual environment. For example, Ramamoorthi and Hanrahan convolve the incident radiance

with a lambertian kernel and fit it with spherical harmonics [RH02]. Spherical harmonic functions define a basis to represent functions on the spherical domain, analogous to the Fourier series in the 1-dimensional domain [RH02].

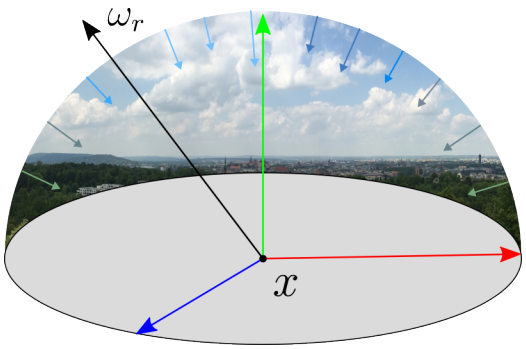


Figure 2.6: A shading point lit by an environment map.

We recall the rendering equation we want to solve over the domain Ω_+, and how we approximated it using Monte Carlo integration (2.11). Evaluating this formulation for IBL requires many samples per pixel for sufficient quality (depending on the nature of the environment map). The split sum technique [Kar13] enables highly performant real time IBL, by splitting the Monte Carlo formula into two parts. One with the radiance from the environment L_i and one with the foreshortened BRDF term :

$$L_o(x, \omega_r) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{f(x, \omega_r, \omega_j)\, L_i(x, \omega_j)\, \cos(\theta_j)}{p(\omega_j)} \tag{2.17}$$

$$\approx \left(\frac{1}{N} \sum_{j=1}^{N} L_i(x, \omega_j) \right) \left(\frac{1}{N} \sum_{j=1}^{N} \frac{f(x, \omega_r, \omega_j)\, \cos(\theta_j)}{p(\omega_j)} \right). \tag{2.18}$$

This approximation is exact for a constant $L_i(x, \omega_j) = L$, and the authors postulate that it is fairly accurate for most environments. We believe that it is the case because importance sampling the areas of high radiance leads to a low variance among L_i samples. It is probably less accurate for high contrast environments. The first sum, which does not take the view direction into account, can be precomputed once for each environment map and for each material roughness value, and stored in a prefiltered mipmap. The second sum can be precomputed, and reconstructed exactly at runtime

2.2.4 Analytic solutions to area lighting

To compute lighting from a polygonal or area light, we integrate the BRDF over the polygonal domain covered by the light (2.16). Real-time rendering techniques aim for a closed-form solution of this integral; thus, strategies generally fall into two categories: approximating the radiance, and approximating the reflectance. For instance, a commonly used technique in video game engines is to approximate the area light with a representative point light, e.g. [WLWF08]. Unfortunately, the results can be orders of magnitude away from the ground truth, and often yield unconvincing visuals [LDR14].

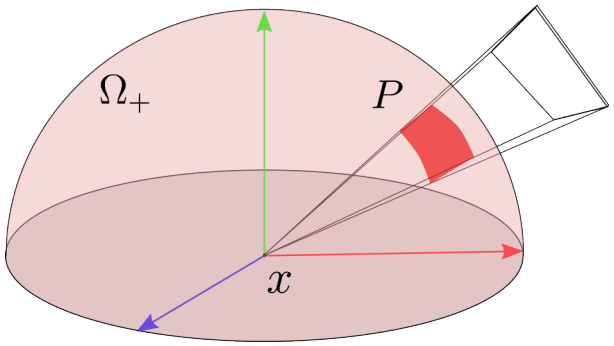


Figure 2.7: Integral of the polygon on the uniform hemispherical domain.

While the GGX BRDF cannot be analytically integrated over the polygon, we can find simpler spherical distributions that can. The Spherical Gaussian model [Fis53] and the Spherical Harmonics are both popular approximations for simple spherical functions as it can fit distributions of many frequencies. However, there is no analytic solution for their integration over spherical polygons [XCM+14]. The simplest case of a suitable function is the uniform spherical distribution, whose integral over a spherical polygon is, by definition, the solid angle of that polygon (Girard's theorem) (figure 2.7). Similarly, the hemispherical distribution's integral on the same domain is the solid angle of the polygon clipped to the hemisphere.

Noteworthy functions with analytical integration properties are the cosine-like distributions, the simplest being the ideal diffuse (Lambertian) distribution. The closed form of the cosine distribution's integral over a spherical polygonal domain [Lam60] was introduced

to graphics by Baum et al. [BRW89]. This was extended by Arvo to Phong distributions with arbitrary roughness [Arv95], with a drawback that the integration cost was proportional to the number of polygon vertices, and to the Phong exponent. This was fixed in [LDSM16] with a real-time solution for the integral that did not increase in cost with the sharpness of the distribution.

The current state of the art for analytic area light illumination, described in the next section, was introduced in [HDHN16], and has since been extended to linear lights and sphere/disk lights [HH17], and to anisotropic materials [KHDN22].

2.2.5 Linearly Transformed Distributions

Heitz et al. introduce Linearly Transformed Spherical Distributions (LTSD): by applying a linear transformation (3×3 matrix M) to its direction vectors ω_o, the original distribution can be reshaped (2.19) [HDHN16]. Reciprocally, we recover the original direction vector ω_o with the inverse transformation M^{-1} applied to ω (2.20):

$$\omega = \frac{M\,\omega_o}{\|M\,\omega_o\|} \tag{2.19}$$

$$\omega_o = \frac{M^{-1}\,\omega}{\|M^{-1}\,\omega\|}. \tag{2.20}$$

By definition, the original distribution D_o and the linearly transformed distribution D are related with this equality [HDHN16]:

$$D(\omega) = D_o(\omega_o)\,\frac{d\omega_o}{d\omega}. \tag{2.21}$$

For all M, the uniform spherical distribution Ω stays identical under linear transformation. Therefore, for all M, the norm of D_o over the spherical domain is identical to the norm of D:

$$\int_\Omega D(\omega)d\omega = \int_\Omega D_o(\omega_o)\,\frac{d\omega_o}{d\omega}\,d\omega = \int_\Omega D_o(\omega_o)\,d\omega_o. \tag{2.22}$$

Furthermore, the authors show that if the integration of D_o over a spherical polygonal domain has an analytic solution, this property extends to D [HDHN16]. Indeed, the

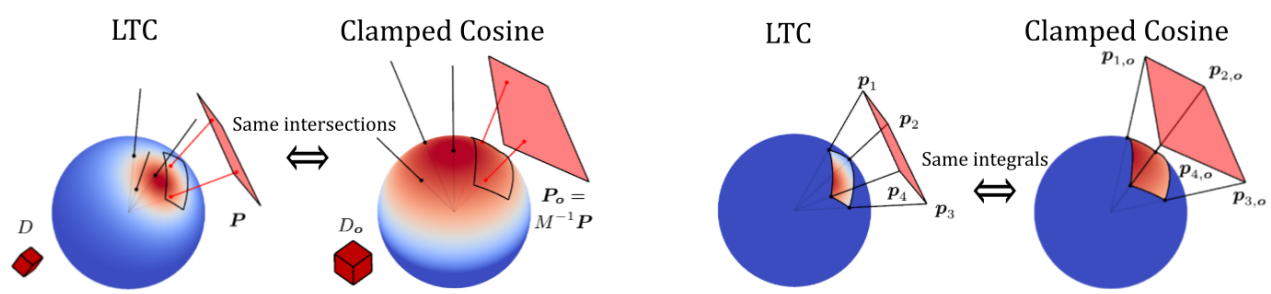


Figure 2.8: Samples of D have the same probability of intersecting P as samples of D_o with P_o. Figure from [HDHN16].

integral of D over a polygon P is the integral of D_o over the polygon $P_o = M^{-1}P$ (figure 2.8):

$$\int_P D(\omega)d\omega = \int_{P_o} D_o(\omega_o)\, d\omega_o. \tag{2.23}$$

2.2.6 Linearly Transformed Cosines

Heitz et al. also introduce Linearly Transformed Cosines (LTC) [HDHN16]. As a base distribution, they choose the normalized cosine (Lambertian) clamped to the upper hemisphere Ω_+:

$$D_o(\omega_o = (x, y, z)) = \frac{\max(0, z)}{\pi}. \tag{2.24}$$

From this base distribution and (2.21), we can generate a family of linearly transformed variants D we call Linearly Transformed Cosines. Since the Lambertian distribution is known to have an analytic integral over a polygon, all LTCs can be analytically integrated over a polygon.

Approximating GGX with LTC

LTCs can yield good approximations to physically based BRDFs,

$$D \approx f(x, \omega_r, \omega_i)\, \cos(\theta_i), \tag{2.25}$$

using the parameters of M to warp the space to support various roughness, anisotropy and skewness. Specficially, we can approximate the GGX microfacet BRDF [WMLT07] from a single well-fitted LTC lobe (figure 2.9). The BRDF is cosine-weighted and is view-evaluated for a given view elevation, since the reflectance distribution of isotropic material only varies with the view elevation.

For fitting, Heitz et al. use gradient descent to find the matrix parameters that minimize the L3 loss between the two distributions [HDHN16]. They rely on cleverly selected starting points, such as using the already optimized neighboring entries to fit the next entry, since L3 is prone to null gradients.

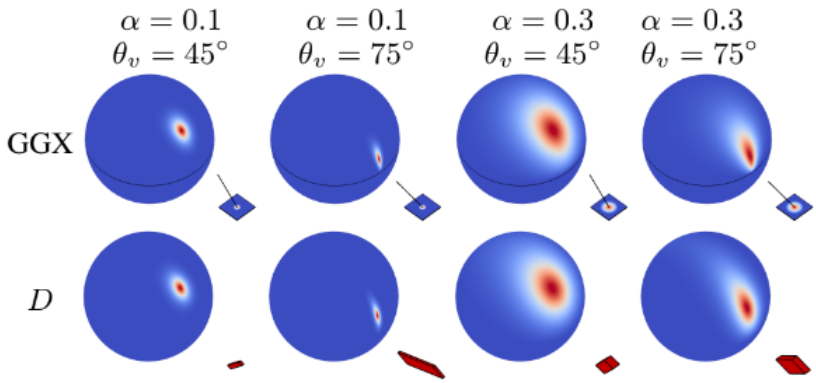


Figure 2.9: Linearly Transformed Cosine distributions D fit view-evaluated GGX BRDFs at different view elevations. Figure from [HDHN16].

Due to the planar symmetry of isotropic BRDFs and due to the scale invariance of LTCs, the authors show that they only need to optimize 4 matrix parameters to approximate most GGX shapes; and the other terms are known to be 0 [HDHN16]:

$$M = \begin{bmatrix} a & 0 & b \\ 0 & c & 0 \\ d & 0 & 1 \end{bmatrix}. \tag{2.26}$$

2.2.7 Anisotropic Linearly Transformed Cosines

Since 2017, real-time rendering engines have been using an LTC approximation of the isotropic GGX model, based on the method explained above. KT et al. extend this idea to anisotropic GGX so that they can support materials such as brushed metals [KHDN22]. Our own contribution is based on the technique presented in [KHDN22]; for more details please refer to the method chapter.

For more flexibility in the shape of the distribution, the autors compute a 8^4 look-up table (LUT) storing all 9 matrix parameters for each view elevation θ, view azimuth φ, and pair of anisotropic roughness coefficients (α_x, α_y). Indeed, the shape of anisotropic BRDFs also varies with φ, thus they fit LTCs to view-evaluated GGX BRDFs over several view azimuths. They leverage the 4-way azimuthal symmetry of the roughness coefficients, by only tabulating for $\varphi \in [0, \pi/2]$ (while the real range of φ is $[0, 2\pi]$). This reduces the storage size by a factor of 4.

Error metric

The authors solve the problem of the null gradients with the L3 error metric used in [HDHN16], especially when the starting distribution has no overlap with the target distribution [KHDN22]. They instead choose the Sliced Wasserstein (SW) loss [BRPP15] for fitting the LTC distribution to the target GGX. It is a sample-wise loss, approximating the optimal transport between the two distributions f and g, and which always provides smooth gradients. It is defined as

$$L_{SW}(f, g) = \mathbb{E}_{\omega \epsilon \Omega} \left[\int_0^1 |F_\omega^{-1}(u) - G_\omega^{-1}(u)| du \right] \tag{2.27}$$

$$\approx \lim_{n \longrightarrow \inf} \mathbb{E}_{\omega \epsilon \Omega} \left[\frac{1}{n} \sum_{i=1}^{n} |f_{i,\omega} - g_{i,\omega}| du \right]. \tag{2.28}$$

In other words, the integral of the difference between inverse CDFs can be approximated stochastically by using n sorted samples from the original distributions (2.28) (figure

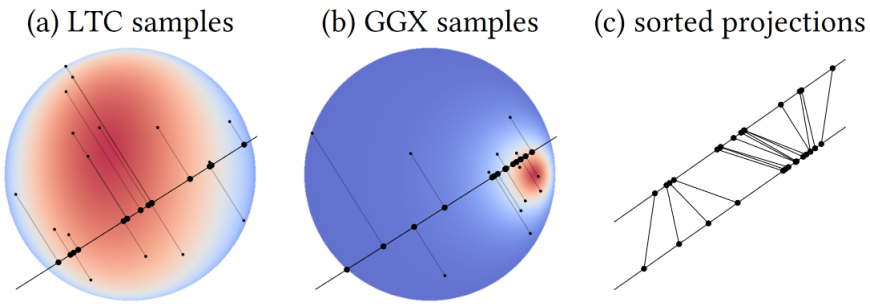


Figure 2.10: Illustration of the sliced Wasserstein metric employed in [KHDN22]. We importance sample the fitted distribution (a) and the reference distribution (b), and project the samples onto a random direction. Finally, we sort the projections and compute the absolute difference between them (c). Image from [KHDN22].

2.10). They propagate the gradient of the loss back to M and finish with a gradient descent step. In this work, we also use SW as a measure for loss.

Other concerns

Finally, we can note a few precautions taken by KT et al. when fitting LTCs in the anisotropic case [KHDN22].

The first problem is the artifacts that arise when several LTC matrices yield the same distribution (non-uniqueness of LTC). Indeed, the cosine distribution is invariant under rotations around the z-axis, and under x-y axis flipping. Thus, two adjacent matrices in the LUT could be fitted properly and still contain widely different parameters. This would yield incorrect parameters when interpolating between two matrices at runtime. Therefore, the authors use a minimization process on the parameters post-fitting to cancel rotations and flipping, and ensure coherence between adjacent matrices in the LUT.

They also fix certain values known to be 0 in certain configurations, which gives more stability to the generated lobes. For instance, when $\theta = 0$, the non-diagonal elements of M are 0, and the diagonal elements are identical regardless of φ.

2.2.8 Area light shading with LTCs

In [HDHN16] and [KHDN22], after fitting BRDFs with LTC distributions at a few discrete view directions, and storing the matrices in LUTs, we can linearly interpolate the LTC parameters to approximate BRDFs at unseen elevations. This allows both techniques to stay lightweight and fast at runtime.

We recall that the objective of the real time portion is to estimate (2.16) analytically, at each shading point. The final analytic polygonal light integration unfolds in the same way in both works. For a given shading point, the n vertices of polygon P are expressed in the referential of the shading point, and are transformed by M^{-1} to obtain P_o. Finally, they use [Lam60] to compute the irradiance of the clamped cosine distribution over the spherical polygonal domain of P_o.

Using the precomputed matrices

In the prefiltering, Heitz et al. stored the inverse of the LTC matrices M^{-1} in the LUT [HDHN16]. Thus in the real time part, they look up M^{-1}, determined by the material roughness and view elevation, and linearly interpolate between the closest tabulated values. It allows them to multiply the polygon whith M^{-1} without having to invert it first.

In contrast, KT et al. stored the regular matrices M in the LUT, which enables a coarser tabulation resolution [KHDN22]. They compute the view angle and interpolate matrix parameters for both the elevation and azimuth. The true range of φ is $[0, 2\pi]$, yet values were stored for $\varphi \in [0, \pi/2]$, so they extrapolate the values of M in other directions using symmetry. Finally, they invert the matrices which is an additional cost not present in [HDHN16], which directly stored the inverse.

Summary

In summary, the entire derivation of how to approximate the integral of the view evaluated BRDF over the polygonal domain using LTCs is

$$L_o(x,\omega_r) = \int_P f(x,\omega_r,\omega_i)\, L_i(x,\omega_i)\, \cos(\theta_i)\, d\omega_i \tag{2.29}$$

$$= L \int_P f(x,\omega_r,\omega_i)\, \cos(\theta_i)\, d\omega_i \tag{2.30}$$

$$\approx L \int_P D(\omega_i)\, d\omega_i \tag{2.31}$$

$$\approx L \int_{P_o} D_o(\omega_{io})\, d\omega_{io} \tag{2.32}$$

$$\approx L\, E(P_o). \tag{2.33}$$

D stands for the LTC distribution, and D_o stands for the clamped cosine distribution. In (2.33), the integral of D_o over the polygon P_o is the irradiance E of the polygon [BRW89].

2.3 Normal map appearance filtering

We now look at the aliasing problems encountered when rendering highly detailed surface microstructures, and we will review previous techniques that tackled these problems.

2.3.1 Microfacet models

Real world surfaces are not perfectly smooth: materials are composed of microscopic structures of varying scale, direction and roughness, which have the effect of scattering light. The appearance of those surfaces can be represented with microfacet models [CT82] [WMLT07], which model the complex surface with a simplified macrosurface, for which the BRDF matches the general scattering direction of the microfacets. These models, such as the GGX microfacet BRDF [WMLT07] presented in section 2.1.5 (2.8), assume that micro-surface details are too small to be seen individually by the viewer. To characterize the reflectance behavior of the surface, we are therefore interested in the statistical distribution of normals on the surface with respect to the underlying macrosurface (or geometric) normal n_g (figure 2.11). These models offer flexibility with a varying roughness parameter, and good results under distant views and under broad illumination.

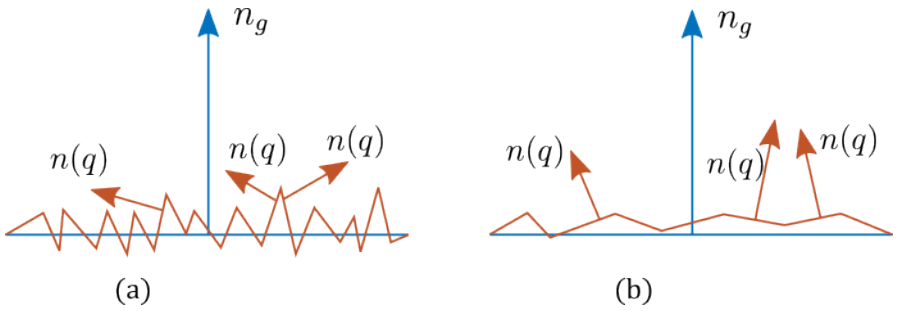


Figure 2.11: (a) A microfacet model with high roughness has a lot of variation among microfacet normals $n(q)$. (b) A smooth surface has little variation of microfacet normals (Image inspired by [HMB+15]).

However, surface details of all scales affect the appearance of a surface and certain larger features are individually discrenable. For instance, microstructures contained in

metallic paints, plastics and brushed metal, are a few micrometers in size, and give these materials their distinguishable glinty appearance (figure 2.12). These effects cannot be replicated by positing infinitely small microfacets, or by applying the same BRDF to the whole surface. The visually rich behavior caused by those microfacets is more noticeable when the light source subtends a small solid angle, which is either a distant or small light source. However, specular highlights which catch the surface detail are still noticeable under broad or global illumination [GGN18].

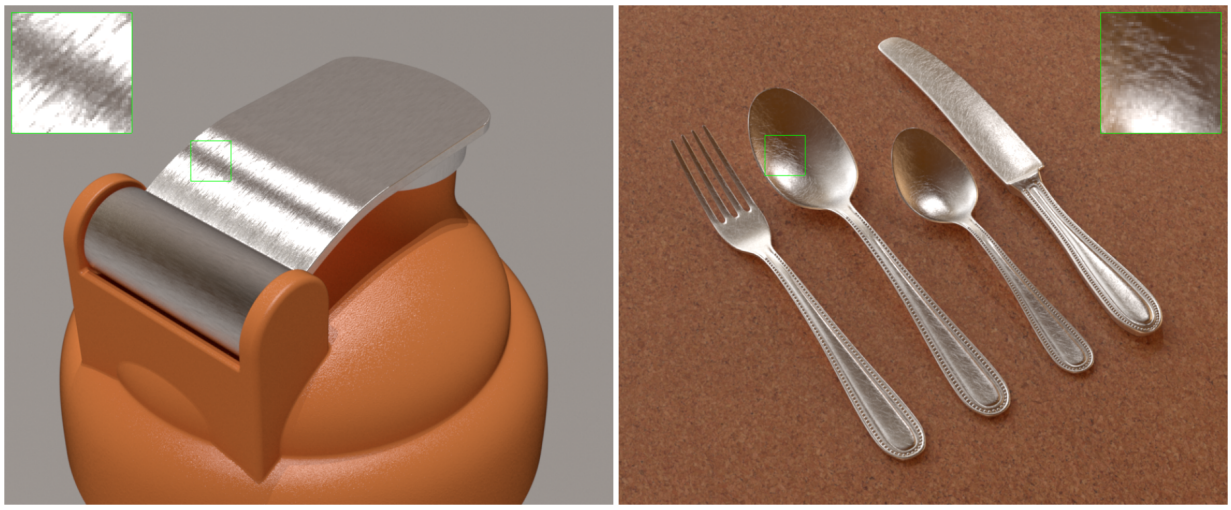


Figure 2.12: The surfaces presented in [YHJ+14] are covered in high-frequency microfacets, and the specular highlights catch the surface detail.

2.3.2 Normal mapping

Among techniques targeting accurate renderings of high-frequency normal variation, we can discern if the description of the underlying normals is implicit or explicit. Implicit, or procedural approaches, rely on statistical processes to describe the normal variation. For instance, Jakob et al. distribute discrete specular point scatterers randomly, to render high frequency glittery surfaces [JHY+14]. More recently, chermain et al. proposed a physically based procedural BRDF, using a compact representation for normal distributions inside of a mip hierarchy [CSDD20]. These methods lead to faster runtime performance and lower

storage requirements, as the normal distribution can be available analytically. However, this is at the cost of reduced control over appearance.

To explicitly represent geometry at a sufficient resolution to reveal the features that cause glints, the tool of choice is a high-resolution normal map. Normal mapping [Bli78] is the application of a texture to surface so that it specifies the normal direction at each surface point. It enables higher control over the surface appearance, but the downside of storing the orientation of each microfacet is larger memory requirements. On the left of figure 2.13 is a height map, also called bump map or height field, and it stores the apparent vertical offset from the surface normal. On the right is a regular normal map, which is the derivative of the height field. We choose to work with this explicit approach, so that we support any arbitrary normal map texture to represent the surface detail.

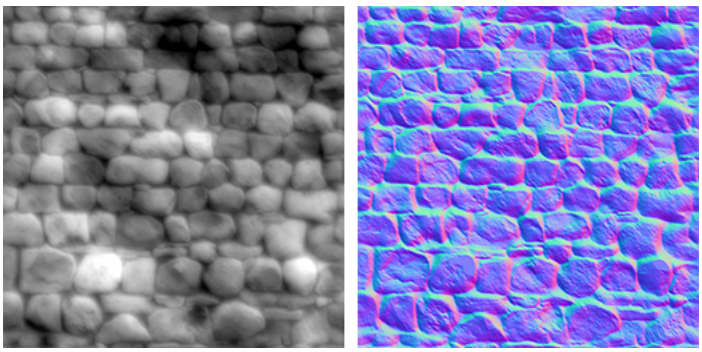


Figure 2.13: A height map (left) and its corresponding normal map (right). The RGB color value of each texture element (texel) in the normal map stores the (x, y, z) direction of the normal vector at that texel in tangent space (Source: https://docs.unity3d.com/2017.2/Documentation/Manual/StandardShaderMaterialParameterNormalMap.html).

2.3.3 Normal Distribution Function and effective BRDF

The rendering equation we have seen in (2.10) describes the outgoing radiance from a shading point towards the camera. However, when shading a pixel in a renderer, we really need to consider the outgoing light towards the camera from the whole surface covered by

this pixel. For this purpose, we define the footprint F of a pixel as the area subtended by its projection onto the shading surface (figure 2.14).

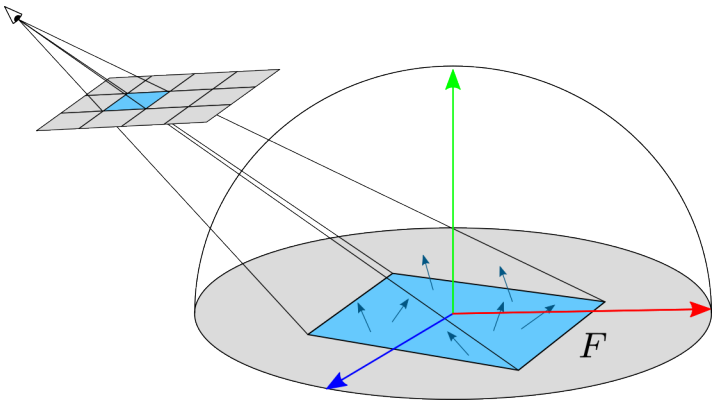


Figure 2.14: The pixel footprint F and the texel normals $n(q)$ that lie within F.

NDF

Let us now formalize a way to manipulate normals within a footprint. For a patch F of the shading surface, the Normal Distribution Function γ_F is the distribution of normals that lie within F [HSRG07]. For a discrete normal map of texels q with normals $n(q)$, and a direction $n \in \Omega_+$, $\gamma_F(n)$ is defined as

$$\gamma_F(n) = \frac{1}{N} \sum_{q \in F} \delta\left(n - n(q)\right), \tag{2.34}$$

where N is the number of normals in γ_F. The value of the Normal Distribution Function is $1/N$ when n is aligned with a texel normal within F. We note that each q is located at a different shading point inside F, but for simplicity we assume that for all normals in F, the view direction and incident radiance from a given direction are computed at the center x of F.

eBRDF

Now, if we add the base BRDF f to the model of the reflectance of F, we can define the effective BRDF (eBRDF) f^{eff} of F as the weighted sum of f evaluated at all texels q in

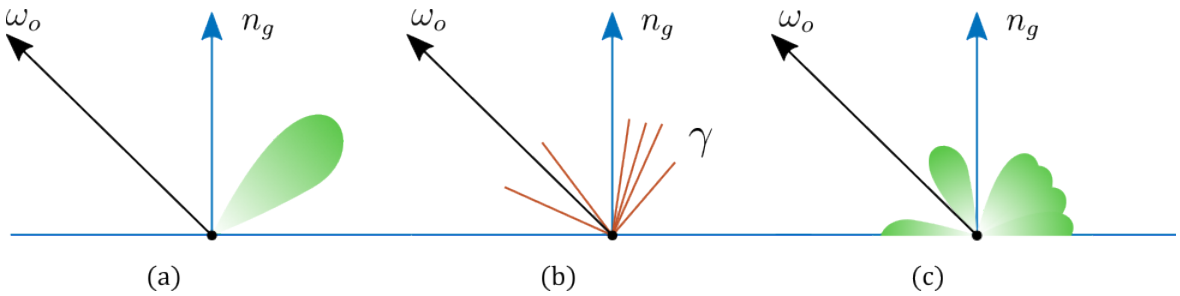


Figure 2.15: (a) The base BRDF of the macrosurface. (b) A discrete NDF of microfacets. (c) The effective BRDF is computed as the convolution of the previous two.

F [HSRG07] (2.36). More generally the eBRDF is an integral over the hemisphere of the base BRDF weighted by γ_F (2.35). Thus, the eBRDF of a patch can also be viewed as the convolution between the BRDF and the NDF of that region, as discussed in [HSRG07] (figure 2.15). The two ways of computing the eBRDF are

$$f^{eff}(x, \omega_o, \omega_i) = \int_{\Omega_+} f(x, R_n(\omega_o), R_n(\omega_i))\, \gamma_F(n)\, dn \tag{2.35}$$

$$= \frac{1}{N} \sum_{q \epsilon F} f\left(x, R_{n(q)}(\omega_o), R_{n(q)}(\omega_i)\right), \tag{2.36}$$

where the operator $R_n(\omega)$ rotates ω to a local texel frame where n is vertical. Indeed, the eBRDF is computed in the tangent frame of the macrosurface, but the base BRDF is computed in the local texel frame. We note that like the BRDF, the eBRDF respects the conservation of energy principle. If the patch contains a single normal, the eBRDF is the same as the base BRDF, but rotated to the local frame of the texel.

2.3.4 Shading aliasing

Shading the appearance of a high resolution normal map for a given pixel amounts to accurately evaluating the effective BRDF of that pixel's footprint. This means sampling every $n(q)$ in γ_F, evaluating the base BRDF at each $n(q)$, and averaging the results (2.36).

Inefficient sampling

However, sampling every texel normal in γ_F is prohibitively costly, as thousands of normals could lie in the patch (either when the camera is far, or when the normal map is high resolution). We would think that a stochastic method is preferred, but Monte Carlo integration over the pixel's footprint is revealed to be impractical [YHJ+14]. Indeed, the energy of the pixel will be concentrated in a few microfacets whose normals are at the half angle between the light and the view direction. These texels take up a minuscule fraction of the pixel, so uniform pixel sampling is ineffective at hitting these highlights.

We note that at this stage, it is the spatial integral over the pixel that is inefficiently sampled, not the BRDF integral over incident radiance; therefore MIS [Vea98] would not solve the issue [YHJ+14]. More effective strategies have been suggested, such as efficiently pruning the NDF normals that will contribute to the final shading, given the light and view directions [JHY+14][YHJ+14][YHMR16]. In short, these techniques find ways to efficiently query normals closely aligned with the half vector.

Non-linear shading

Futhermore, when sampling pixel points stochastically and computing the illumination of the surface, each sample will probably yield a very different shading result. In the final render, this would cause a category of undesirable artifacts we call shading aliasing. They tend to erase the recognizable macroscopic features of the normal map, and cause fireflies and flickering (we show examples of this in the result section). The artifacts are more pronounced under high frequency lighting, i.e. when the base BRDF is specular or when the light source is small. In the perspective of signal processing, aliasing occurs when we sample the signal at a lower frequency than what is required to reconstruct it. The artifacts therefore stem from the frequency difference between the pixel and the texture elements.

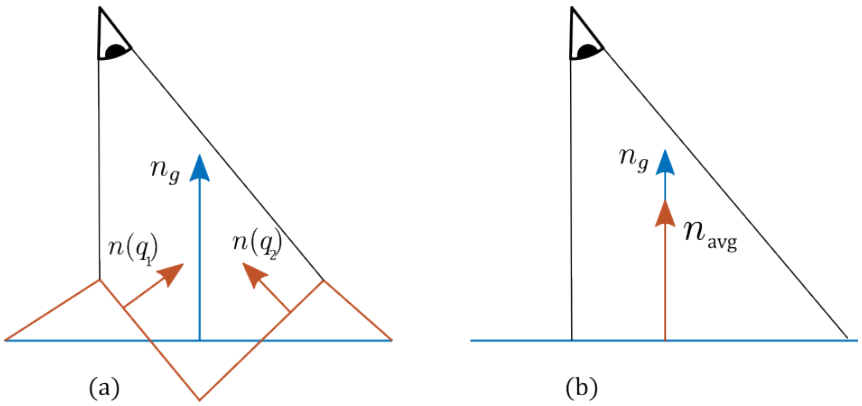


Figure 2.16: (a) The two normals of the surface lie in the NDF of the pixel's footprint. (b) The average of the two normals is equal to the geometric normal n_g, which yields a much different shading than with the two opposite normals.

Antialiasing normal maps is a difficult task because, unlike color, each normal does not contribute linearly to the shading [OB10]:

$$\int_{q \in F} f(q)\, dq \neq f\left(\int_{q \in F} q\, dq\right). \tag{2.37}$$

Thus, we cannot linearly average the normals within the footprint to simplify the task. As a major example of non-linear shading, we consider the v-groove geometry presented in [HSRG07] (figure 2.16).

2.3.5 Texture filtering

The main approach to address shading aliasing is a family of techniques called texture filtering, where we pre-process the texels that will be then mapped to screen pixels. In the case of anti-aliasing high-resolution normal maps, we use minification methods, which usually involves the implementation of a low pass filter. In other words, we pre-process the texels that occur at a higher frequency than what is required to display on screen [HSRG07].

Mip-mapping are widespread filtering techniques used for spatial anti-aliasing and for savings on memory/computation [Wil83]. These methods are effective at reducing

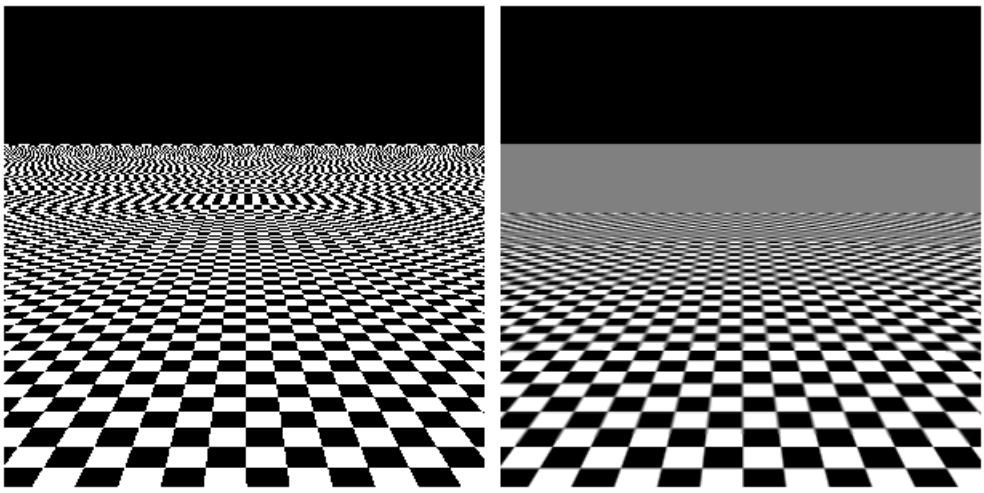


Figure 2.17: Left: the checkerboard texture is not filtered and shows serious aliasing in the form of moiré patterns. Right: a mipmap is applied, which removes the aliasing artifacts at the expense of blurring and loss of detail (Source: https://textureingraphics.wordpress.com/what-is-texture-mapping/anti-aliasing-problem-and-mipmapping/).

aliasing in image textures, at the expense of some blurring [EWWL98] (figure 2.17). A mip hierarchy is composed of a pyramid of textures where each texture is half the resolution of the previous. Each level is computed with simple bilinear interpolation from the level above. At runtime, the number of texels in the pixel footprint determines the MIP level to use to shade the point (i.e. the more texels per pixel, the higher the MIP level used). To determine the final color of the pixel, the most commonly used techniques are bilinear interpolation, and trilinear interpolation on adjacent MIP levels.

MIP-mapping image textures is commonplace in real time applications, and is even sometimes used for normal map filtering. However, as mentioned in 2.3.4, due to the nonlinearities in shading, we cannot average nearby surface normals to approximate the shading behavior of a patch [HSRG07]. It would either smooth out the texture so that we would loose the glinty effect we desire (figure 2.16), or it would manifest as flickering of specular highlights. Ideally, filtering techniques adapted to normal maps keep surface details throughout different scales and completely erase shading aliasing. The next parts

of the appearance filtering chapter will describe minification techniques that are useful for efficient shading anti-aliasing.

2.3.6 Real-time normal map filtering techniques

Initial work on antialiasing for explicit normal maps was led by the interactive graphics community, and the filtering methods cited in the following sections are improvements on mipmapping.

Normal map filtering techniques can deliver renderings free of aliasing in real time, by approximating the pixel's NDF by a single smooth distribution at each scale and by leveraging the hardware optimized for mipmaping. These approaches store the filtered normal map in a mip hierarchy separate from the original texture, and interpolate adjacent mip texels linearly such that larger patches of microfacets are approximated as flat surfaces with higher roughness. Although they all eliminate aliasing, these techniques differ in the amount of detail they can represent. Schilling [Sch97] fits the NDF using 2D covariance matrices, and Olano and North [ON97] use a single 3D Gaussian. Toksvig [Tok05] presents a more affordable method, especially useful for the diffuse component of shading. They show how the shortening occuring when interpolating unit normals inside an NDF can be used as a measure of normal variation, and can help minimize the effects of shading aliasing.

It can be argued that a single lobe is insufficient to fit complex NDFs [HSRG07]. For instance, Tan et al. [TLQ+05] fit a mixture of several 2D Gaussian lobes to a planar projection of the NDF at each texel. Han et al. [HSRG07] present a more accurate antialiasing method where the multi-scale NDFs are precomputed and fitted by a mixture of spherical harmonics for low frequency materials. For higher frequency materials, they use a small number of von Mises-Fisher distributions (spherical Gaussians).

Olano and Baker introduce Linear Efficient Antialiased Normal (LEAN) Mapping [OB10]. This method enables the filtering and combination of several layers of bump maps linearly, and is fast enough for complex time-varying bump effects. They represent bump

variation as the variance of gaussians, and store in textures the mean and second moment of the gaussian lobes. They are able to represent directional highlights spreading across bumps, using an anisotropic model based on [War92]. It has been widely adopted as it is efficient and easy to add to existing pipelines (figure 2.18).

Finally, Dupuy et al. present Linear Efficient Antialiased Displacement and Reflectance (LEADR) mapping, which is an extension to displacement maps [DHI+13]. Unlike techniques for flat normal maps, their technique accounts for view- and light-dependent effects such as masking and shadowing on the geometry.

All these real time applications fit a single or a few lobes to the NDF, and while they are able to yield anti-aliased results, they mostly fail to capture the anisotropies of multi-scale NDFS that would produce high frequency glints. Furthermore, fitering the NDF using broad lobes is only accurate if the illumination is itself low-frequency, which would make the complex features disappear anyway [YHJ+14].

The recent real-time technique presented in [CLS+21] improves the method from [CSDD20], by minimizing the aliasing of glints on surfaces with high curvature. They add a BRDF parameter to the model proposed in [CSDD20], making glint rendering compatible with normal map filtering and closer to explicit representations. Their method uses LEAN mapping to extract the roughness and the correlation factor of the normal map NDFs, but it is still limited to the filtering of specific NDFs, and it only works under punctual or directional lighting.

2.3.7 Offline normal map filtering techniques

Under high-frequency illumination, the normals that contribute to the glint are those aligned with the half-vector between the eye and the light direction, and it would be prohibitively inefficient to use importance sampling to find them. Thus, the following techniques offer significant improvements over Monte Carlo, and are much faster than supersampling all texels within the pixel. However, they are not aimed at a real time rendering pipeline as rendering an image takes seconds to minutes.

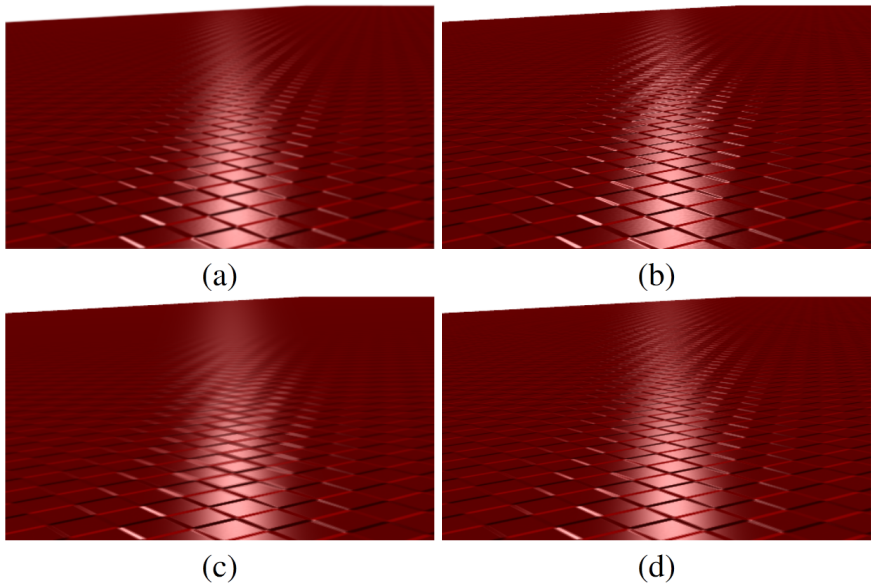


Figure 2.18: Antialiasing of checker grid bump map [OB10]. (a) ground truth computed with supersampling ($\times 64$). (b) mip-mapped normal map, showing shading aliasing artifacts at further distances (c) LEAN mapping with trilinear mip filtering, removing all shading aliasing (d) LEAN mapping with anisotropic mip filtering, which fixes some over-blurring.

Yan et al. [YHJ+14] target the accurate offline rendering of specular glints with a deterministic approach. They shade the surface patch seen through a pixel by evaluating the true pixel NDF for the half-vector, which can be done using the assumption of a Gaussian pixel filter. The normal map is tessellated into small triangular elements, as the integrals of Gaussians over triangles can be approximated.

Broad incident radiance

Until now, we reviewed filtering techniques that assume a fixed incident radiance from a point or directional light. If we shade a patch from a microfaceted surface under broad radiance, we must now compute an expensive double integration:

$$L_o(P, \omega_r) = \int_F \int_{\Omega_+} f\left(x, R_{n(q)}(\omega_o), R_{n(q)}(\omega_i)\right) L_i(x, \omega_i) \cos(\theta_i) \, d\omega_i \, dq. \tag{2.38}$$

The outer integration is the spatial integral of texels over the pixel footprint (NDF), and the inner one is the evaluation of the BRDFs over the incident radiance directions. This makes the appearance filtering problem much more difficult when considering environmental lights or area light sources (figure 2.19).

Using a discrete normal map, we can estimate (2.38) with

$$L_o(P,\omega_r) = \frac{1}{N}\sum_{q\in F}\int_{\Omega_+} f\left(x, R_{n(q)}(\omega_o), R_{n(q)}(\omega_i)\right) L_i(x,\omega_i)\,\cos(\theta_i)\,d\omega_i. \tag{2.39}$$

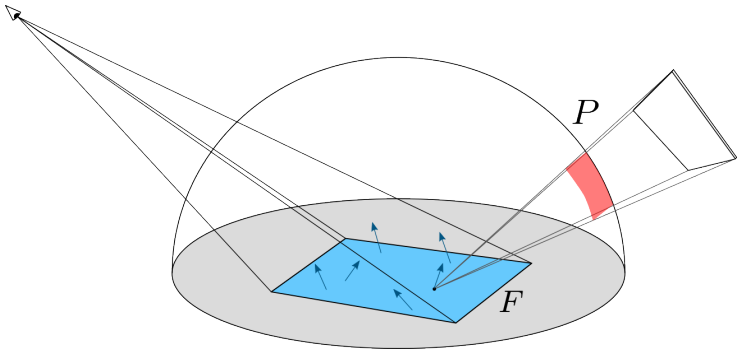


Figure 2.19: Shading a high resolution normal map under broad incident radiance (area light) involves solving a double integral: a spatial integral over the pixel's patch and an angular integral over the polygonal light.

The technique presented in [YHMR16] is more efficient than [YHJ+14] ($\times 100$ faster), and can also integrate over area lights. It involves fitting a mixture of 4D Gaussian elements to the NDF. The authors show that although the mixture can contain millions of Gaussians, and hundreds of Gaussians can contribute to a single NDF, a half-vector query will only depend on a few select lobes. They finally present an efficient gaussian query/pruning method for a given footprint and half-vector.

Belcour et al. [BYRN17] broaden the investigation around surface appearance aliasing to the context of indirect bounces, but is limited to high frequency lighting. Gamboa et al. [GGN18] extends the rendering of sparkles and glints to arbitrary incoming radiance such as global illumination and large area lights, and outperforms previous offline approaches in terms of speed. Indeed, previous works sample the NDF normals that contribute

to most of the pixel's energy for a given view and lighting direction, which becomes prohibitively expensive under broad illumination. The filtered model in [GGN18] uses spherical histogram accumulation to gather statistics of the normals within a given NDF. This allows for efficient numerical integration of the double integral (2.38).

In conclusion, the difficulty of preventing shading aliasing of the normal mapped surface is compounded with the problem of computing real time illumination from an arbitrary area light. This was tackled in offline rendering research, and to our knowledge, there are no existing techniques to solve the double integration in real time.

Chapter 3

Method

The first part of this chapter shows an overview of our contribution to the shading aliasing problem and the design decisions behind our algorithm. The following parts, written in a literate programming style [Knu84], will serve as documentation for the code base we developed for this project.

3.1 Our contribution

Our technique is explained in plain words (section 3.1.1), formally using equations (section 3.1.2), and visually using flow chart diagrams (section 3.1.3). The implementation and documentation surrounding our method can be found in section 3.4.

3.1.1 Overview

Our method aims to accurately render a detailed normal mapped surface under polygonal illumination in real time. In short, we tackle shading aliasing by pre-filtering the normal map and fetching the filtered result at runtime to integrate over the area light efficiently. To do this, we fit a LTC distribution to the view-evaluated effective BRDF of each normal map region at each scale.

When choosing a high frequency base BRDF prone to aliasing, we want to preserve as much of the high frequency components of the effective BRDF for the filtering step. Thus, we run the whole process for a specular GGX base BRDF, and for a diffuse lambertian base BRDF separately. The two components are added together at runtime, when compositing the final image.

From the base BRDF (either a highly specular GGX or a diffuse Lambertian), we generate an importance sampled view-evaluated effective BRDF (eBRDF) for the whole normal map at different MIP levels. We fit the eBRDF with a Linearly Transformed Cosine (LTC) distribution, for each view angle and scale. By filtering the view-evaluated eBRDF with this technique, we are approximating both the spatial integral over the pixel footprint and the angular integral over the hemisphere. The use of (LTC) allows us to integrate analycally over the area light in the real-time portion. In short, the prefiltering step ensures shading anti-aliasing, and once we render the surface at runtime, we only have to sample the filtered result once per pixel, by fetching the correct LTC matrix.

Our technical contribution is the following:

- A stochastic sampling algorithm for the view-evaluated effective BRDF of a pixel, using importance sampling on the underlying BRDF lobes.
- Extension of anistropic LTC fitting in [KHDN22] to specular and diffuse effective BRDFs separately, for different view angles and scales.
- Runtime bilinear software interpolation of LTC matrices on top of MIP hardware trilinear interpolation.

3.1.2 Formal description

We provide here a formal description of our approach, based on the equations seen in the related works chapter. The integrals in (2.38) can be reversed so that

$$L_o(F,\omega_r) = \int_{\Omega_+}\int_F f\left(x, R_{n(q)}(\omega_o), R_{n(q)}(\omega_i)\right) L_i(x,\omega_i)\,\cos(\theta_i)\,dq\,d\omega_i. \tag{3.1}$$

With a discrete normal map and a discrete NDF in the pixel footprint, we can write

$$L_o(F, \omega_r) = \int_{\Omega_+} \frac{1}{N} \sum_{q \in F} f\left(x, R_{n(q)}(\omega_o), R_{n(q)}(\omega_i)\right) L_i(x, \omega_i) \cos(\theta_i)\, d\omega_i. \tag{3.2}$$

If we choose to only illuminate the surface from a constant polygonal light P, from (2.16) we get

$$L_o(F, \omega_r) = L \int_P \frac{1}{N} \sum_{q \in F} f\left(x, R_{n(q)}(\omega_o), R_{n(q)}(\omega_L)\right) \cos(\theta_L)\, d\omega_L. \tag{3.3}$$

To filter the normal map, our contribution is to fit a single anisotropic LTC lobe to the effective BRDF of the pixel's footprint (similarly to (2.25) for a regular GGX). We obtain

$$L_o(F, \omega_r) = L \int_P f^{eff}(x, \omega_o, w_L) \cos(\theta_L)\, d\omega_L \approx L \int_P D(w_L)\, d\omega_L \tag{3.4}$$

which has been shown in (2.33) to be the irradiance of the inversely transformed polygon. The use of LTC therefore allows us to compute the angular integral over the solid angle subtended by the area light, in real time:

$$L_o(F, \omega_r) \approx L\, E(P_o) \tag{3.5}$$

We note that the approximation in (3.4) is highly dependant on the view direction ω_o, as well as the size of F. Filtering the texture for high resolution detail at one scale would not fix aliasing at other scales. This is the reason we filter the normal map by computing a fit at many scales and view angles in the style of an anisotropic mip hierarchy.

3.1.3 Diagrams of the algorithm

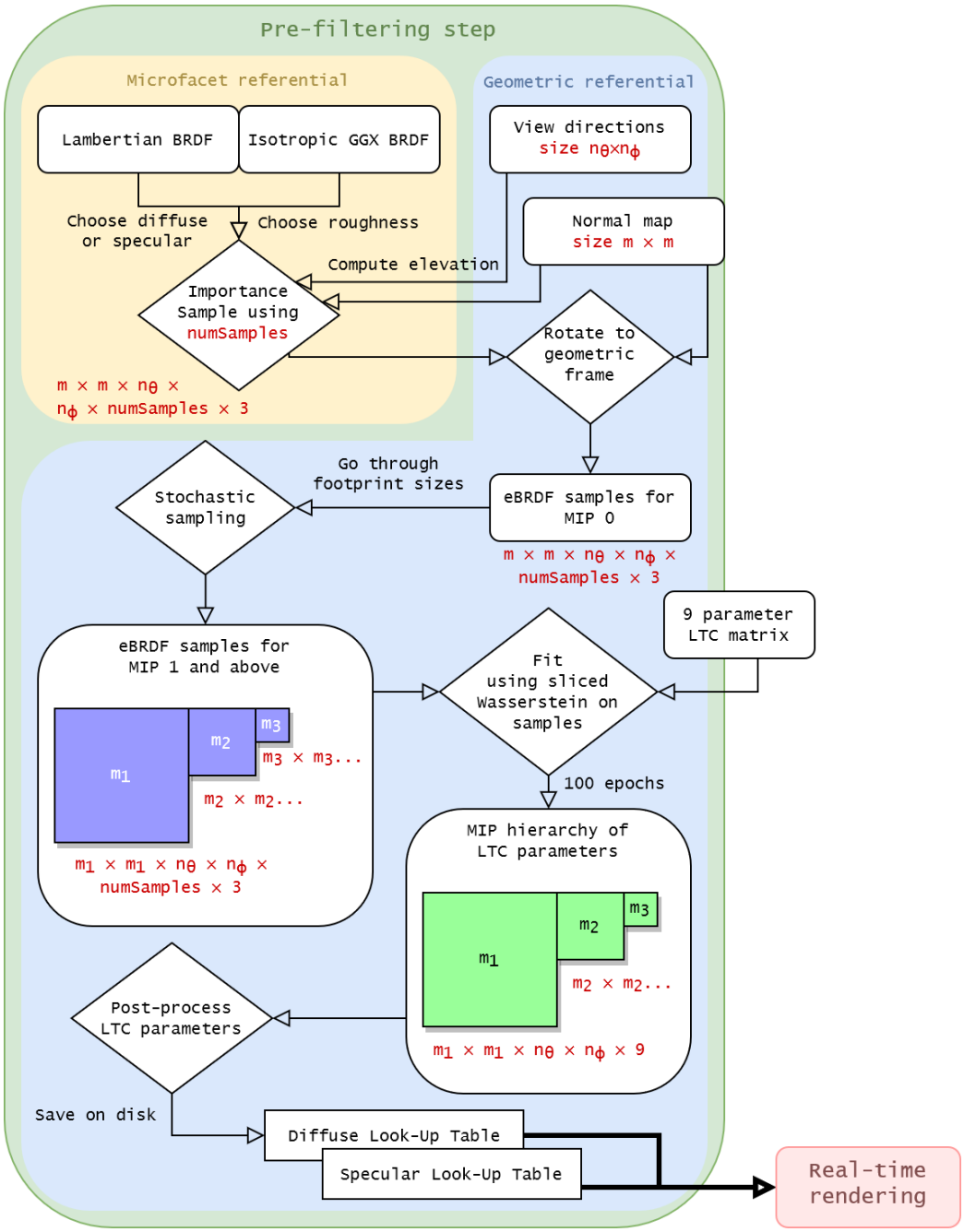


Figure 3.1: The main part of the algorithm is a pre-filtering step. The objects and tensors are indicated inside rounded rectangles, the main operations are inside diamonds, and the dimensions of the objects are indicated in dark red. The output of this step is 2 LUTs containing LTC matrices, one for the diffuse and one for the specular component.

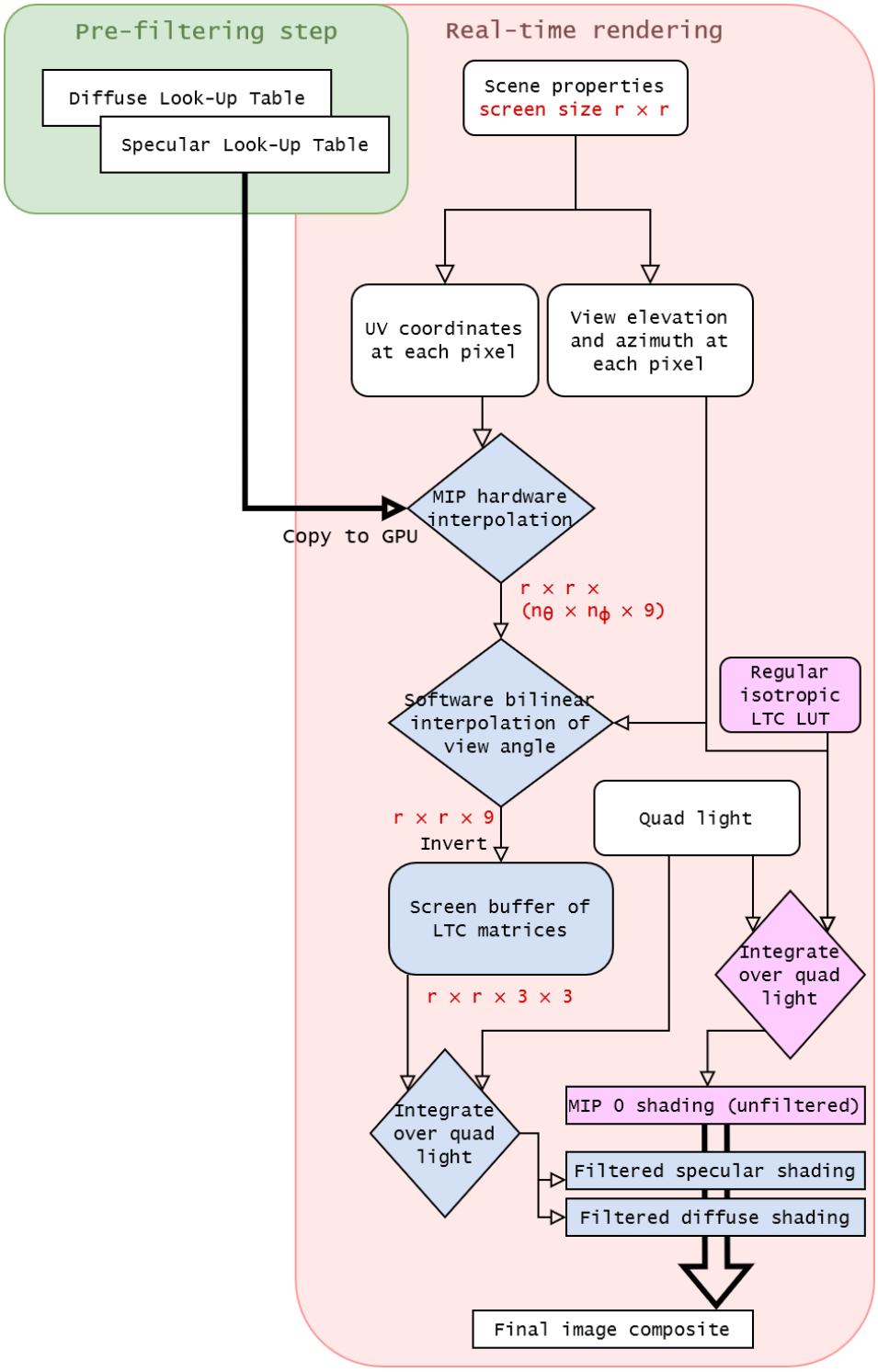


Figure 3.2: The second part of the algorithm happens during the real-time rendering step. The 2 LTC look-up-tables computed in the first step are copied to the GPU to be used. The output of this step is the final rendered image.

3.2 Presentation of the code base

In this second part, we present the code's structure and how to use it to render scenes, and we describe the implementation of our method. The documentation will be paired with fragments of code, but the full code is accessible at this github repository.

The code base developed for this project is constructed for experimentation and for fast prototyping in a rasterizing pipeline, based on the nvdiffrast library. It allows for the easy creation of scenes and their components, and provides a flexible set of options for GPU accelerated rendering of images. The end goal was to develop and present our contribution on top of this framework.

3.2.1 Repository structure

To start with, the code is based in python, with some parts in C++ and in CUDA. From now on, the `.` directory refers to the root directory of the project repository. The `./src` directory stores the source code, and the `./data` directory contains files to be used in the rendering of scenes:

- `./data/mesh` contains the mesh descriptions in PLY formats
- `./data/tex` contains image textures and normal maps
- `./data/LUT` contains precomputed LTC lookup tables

3.2.2 Rasterizing with the nvdiffrast library

Rasterization pipeline

Like many real time graphics applications, we use a variant of the rasterization pipeline, in contrast with a path tracing pipeline, for example. During rasterization, we first project the vertices making up the 3D geometry onto the image plane, and then determine the pixels they cover (figure 3.3(b)). Then, at each pixel, the vertex attributes are interpolated across

triangles (figure 3.3(c)). Finally, fragment or pixel shaders determine the final color of the screen pixels using texture and lighting information. For efficiency, these programs are run independently and concurrently so that any given shader does not know the state or the output of another. The output of the fragment shaders is sent to a frame buffer, which can then be displayed to the screen or used in another computation (figure 2.4).

Nvdiffrast and pytorch

We base our primitive graphics functionalities on NVidia's nvdiffrast library [LHK+20]. It is a fast and low level PyTorch library (also supporting TensorFlow), relying on a CUDA kernel and on the graphics pipeline hardware for GPU acceleration. We implemented every scene attribute, renderer and pre-filtering algorithm on top of the nvdiffrast framework; and following the structure of the library, all our screen buffers and textures are stored in pytorch float tensors. For fast parallel operations, we ensured that all of this tensor data was stored in the GPU. nvdiffrast is presented with a python API containing 4 main functions, and these functions carry their gradients over, for use in differentiable rendering:

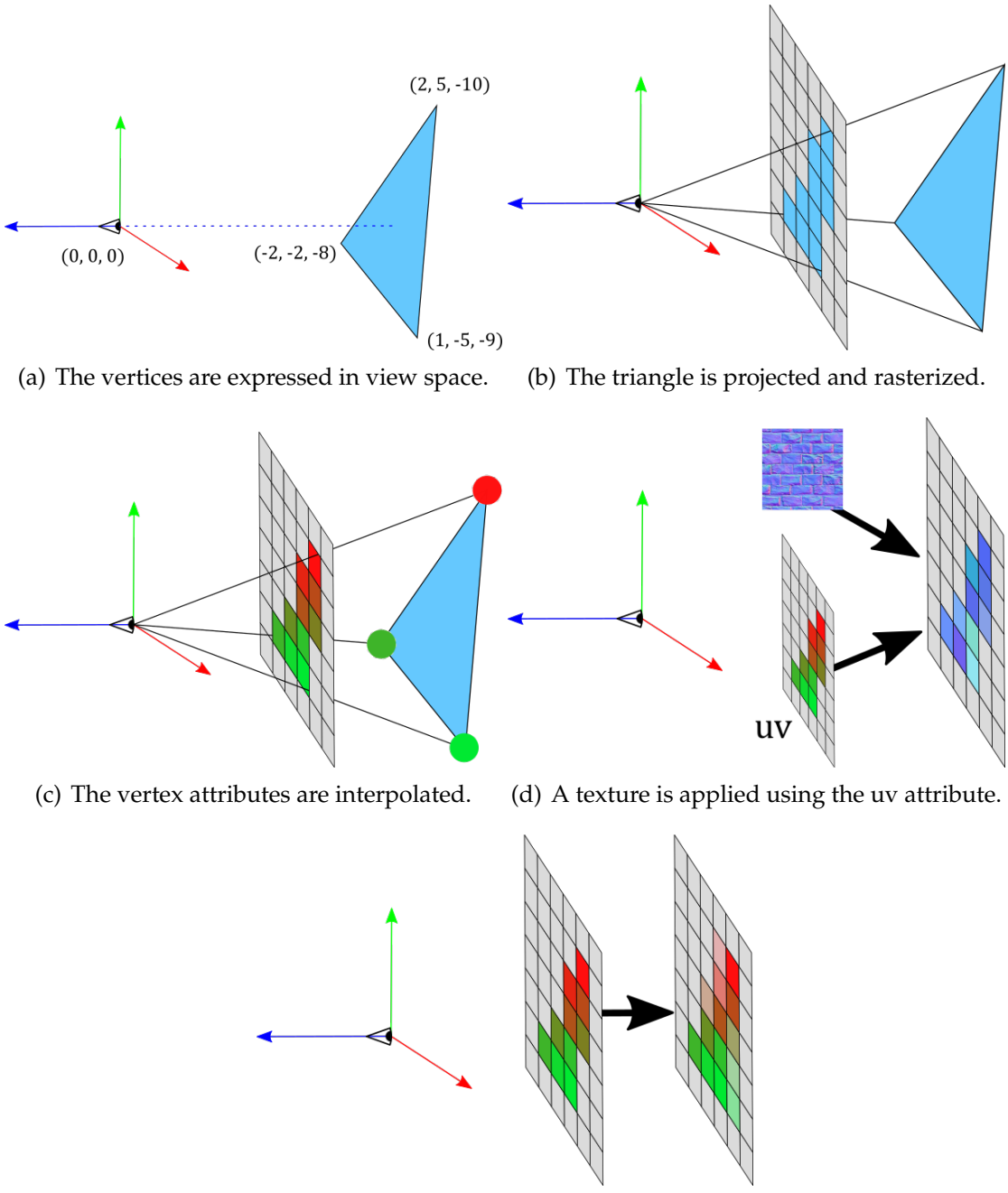
- `nvdiffrast.torch.rasterize()` rasterizes triangles. It takes as argument a set of vertices expressed in view space. The camera is therefore assumed to be at the origin and looking down the z axis (figure 3.3(a)). The triangles are then projected to screen space so that we can compute the pixels they overlap (figure 3.3(b)). The function outputs, for each screen pixel, the depth of its projection, triangle ID, and image-space derivatives of barycentrics. By default, the rasterizer uses point sampling which is prone to aliasing on the edge of triangles.

- `nvdiffrast.torch.interpolate()` linearly interpolates a set of attributes across the pixels covered by the triangles (figure 3.3(c)). This tool takes as input a tensor where each scene vertex is mapped to any number of attributes, and the outputs from the rasterize() operation. It outputs an image-sized buffer where the parameters are interpolated over all pixels.

- `nvdiffrast.torch.texture()` applies a texture to the pixels on screen (figure 3.3(d)). It takes as argument an image texture as a tensor, and the interpolated per-pixel uv coordinates. It can optionally load and apply a MIP hierarchy with hardware MIP interpolation. Finally, it outputs a screen buffer where the texture is applied over the pixels.

- `nvdiffrast.torch.antialias()` returns an anti-aliased copy of the screen buffer, using the output from the rasterizer. The anti-aliasing performed is a linear interpolation at the edges of triangles, which blurs and erases 'stepping' effects (figure 3.3(e)).

Nvdiffrast comes with a few other useful tools, such as `nvdiffrast.torch.RasterizeGLContext()`, a fast way to create new openGL rasterizer contexts. It can also rasterize multiple depth layers using the `nvdiffrast.torch.DepthPeeler()` class. Finally, it provides a mip-map constructor (`nvdiffrast.torch.texture_construct_mip()`) which takes an image texture and outputs a hierarchy of downsampled versions of the same texture.



(a) The vertices are expressed in view space.

(b) The triangle is projected and rasterized.

(c) The vertex attributes are interpolated.

(d) A texture is applied using the uv attribute.

(e) The edges are antialiased.

Figure 3.3: Main steps of the rasterizing pipeline applied to a single triangle.

3.3 Main functionalities and rendering a scene

3.3.1 The Scene class

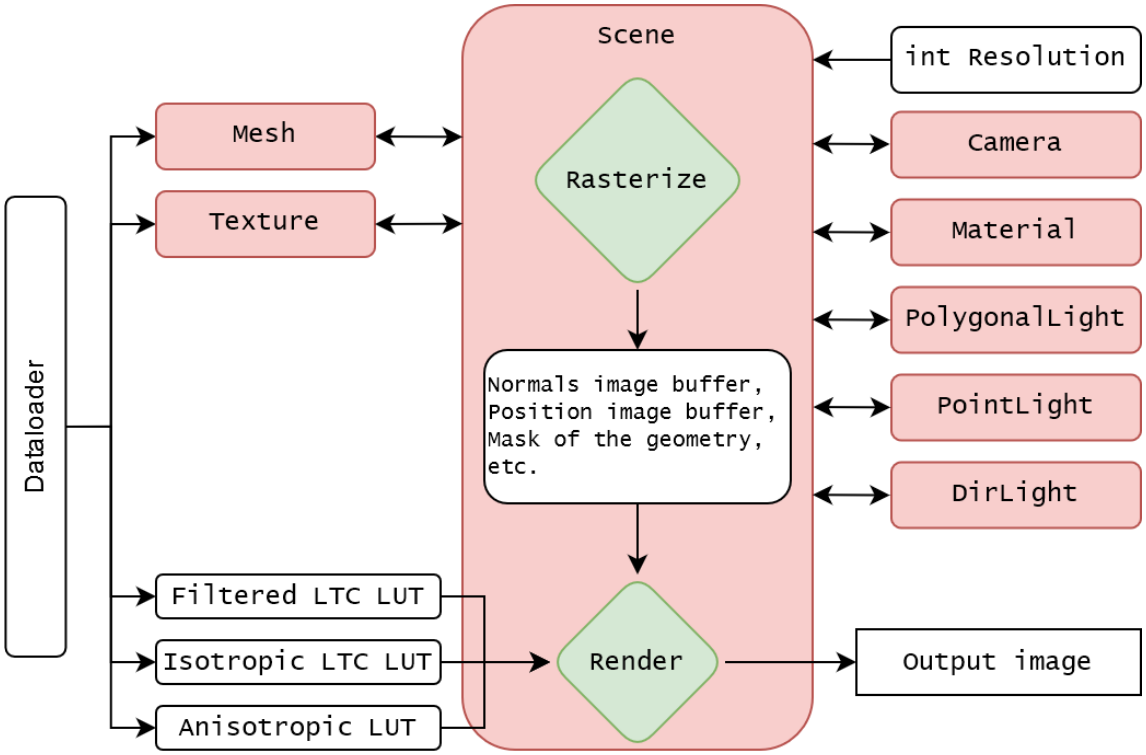


Figure 3.4: This diagram shows an overview of the class structure, and the main operations that are possible with our system. The classes are shown in red, and the two main operations are shown in green diamonds. On the left are the scene attributes loaded from the disk with the dataloader, and on the right are scene attributes built on-the-fly.

The `Scene` class, located in `./src/scene.py`, contains all objects, textures and rendering parameters, and also contains the various buffers creating during rasterization and rendering (figure 3.4). Inspired by the logic of a scene graph, our scene creator is object based, in which each scene attribute is a class instance.

The `Scene` attributes are instances of classes in `./src/parameters.py`. This includes `Camera`, `Texture`, `Mesh`, `PolygonalLight` and `Material`. For creating baselines, we also included `DirLight` and `PointLight`.

Another attribute created at the same time as the scene instance is the `scene.resolution` (set using `scene.setResolution()`). This integer determines the size of one side of the rasterized image, when the scene gets renders.

Below is an example of a standard scene, located in `./src/sampleScenes.py`. To create a scene, we first create the attributes:

```
def standardScene(resolution = 1024, name = "LTCscene"):
    cam = Camera(camera_distance=20)
    mesh = Mesh(f'{datadir}/mesh/planeflat.ply', cam, ty=-2.0, rx=-np.pi/2.0)
    material = Material(roughness=0.05, diffuseR=1.0, diffuseG=1.0, diffuseB
        ↪ =1.0)
    quad = PolygonalLight(cam)
    quad.offsetIdx(mesh.numVertex)
```

And then load them in the `Scene` instance:

```
    scene = Scene(resolution, cam, mesh, material, quad, name)
```

We can finally add special attributes such as LTC lookup tables:

```
    scene.setIsotropicLUT(f'{datadir}/LTC-bin/ltc_parameters_works.bin', f'{
        ↪ datadir}/LTC-dds/ltc_amp32_test.dds', LUTsize=32)
    return scene
```

3.3.2 The main pipeline

We will now go over the main pipeline: how to create a scene, rasterize it and render it. The main file `./src/main.py` contains sample methods showcasing the functionalities of our framework: loading a scene and then launching a real time render.

The function `run()` takes as argument a scene instance, rasterizes it, and and renders it using the chosen rendering algorithm. Finally, it displays the result to the screen:

```
glctx = nvdiffrast.RasterizeGLContext()
scene = normalMapSceneVertical(resolution = 512, superSample=False, enable_mip
    ↪ =True, normalMap='stone256.jpg')
run(scene)
```

```
def run(sc. display = True):
    img = sc.render(glctx, 'isotropicLTC')
    display = img[0].cpu().numpy()[::-1]
    if display:
        util.display_image(display, size=sc.resolution, title='Render output')
    util.writeImage(display)
```

`runLoop()` also render the scene to the screen, but for a certain number of frames (taken as argument). It allows for scene parameters to be updated between frames (mesh, camera and light transformations, material properties, etc.). `runCompare()` is similar to `runLoop()`, but renders two or more scenes simultaneously at each frame and displays them side-by side.

Finally, the function `runOptimize()` function is a sample workflow to leverage the differentiable rendering functionality of nvdiffrast. It renders and displays two scenes side-by side and optimizes the parameters of one of them using gradient descent on image space loss, at each frame. This functionality is detailed in the last part of the method section.

3.3.3 Scene attributes

This section covers the various classes and attributes to attach to a `Scene`.

Camera

It can be loaded to the scene with `scene.setCamera()`. It takes as argument the camera distance and the frustum parameters x, n and f; and sets its projection matrix with the frustum parameters, either using an orthographic projection or a perspective projection. We do not use a view-to-world matrix as we assume that view space is world space (figure 3.3(a)).

```
class Camera:
```

```
    def __init__(self, camera_distance = 20, x = 1.0, n = 2.0, f = 400.0):
        self.camera_distance = camera_distance
        self.setProj(x, n, f)
    def setScene(self, scene):
        self.scene = scene
    def setDistance(self, camera_distance):
        self.camera_distance = camera_distance
        self.scene.rasterized = False
    def setProj(self, x = 1.0, n = 2.0, f = 50.0, orthographic = False):
        self.x = x
        self.n = n # near frustum
        self.f = f # far frustum
        if orthographic: self.proj  = util.torch_projection_orthographic(x=x,
            ↳ n=n, f=f)
        else: self.proj = util.torch_projection(x=x, n=n, f=f)
        try: self.scene.rasterized = False
```

Mesh

It can be loaded to the scene with `scene.setMesh()`. The Mesh takes as argument the name of the mesh file, the camera, the rotation and translation, the mesh scale and the uv scale. We start by extracting the object space vertex coordinates from a PLY file using `dataloader.dataFromPLY()`, and we note that we only support triangular meshes. The PLY format also contains the triangle id, normal and uv at each vertex.

```
class Mesh:
    def __init__(self, filename, camera, scale = 100.0, uvscale = 1.0, rx=0.0,
        ↳  ry=0.0, rz=0.0, tx=0.0, ty=0.0, tz=0.0):
        self.camera = camera

        #load mesh data
        self.basepos, self.npidx, self.normals, baseuv = dataloader.
            ↳ dataFromPLY(filename)
        self.uv = baseuv * uvscale
```

```
        self.pos = self.basepos * scale

        self.numVertex = self.pos.shape[0]
        self.numTriangles = torch.tensor(self.npidx.shape[0], device=torch.
            ↪ device('cuda:0'))
```

We then converts the vertex and triangle properties (in object space) to torch GPU tensors.

```
        self.idx = torch.from_numpy(self.npidx.astype(np.int32)).cuda()
        self.vtx_pos = torch.from_numpy(self.pos.astype(np.float32)).cuda()
        self.vtx_pos4 = torch.cat([self.vtx_pos, torch.ones([self.numVertex,
            ↪ 1]).cuda()], axis=1)
        self.vtx_uv  = torch.from_numpy(self.uv.astype(np.float32)).cuda()
        self.vtx_n  = torch.from_numpy(self.normals.astype(np.float32)).cuda()
        self.vtx_n_transpose = torch.transpose(self.vtx_n, 0, 1)

        if self.pos.shape[1] == 4: self.pos = self.pos[:, 0:3]
```

We note that we keep a vec3 version of the vertex positions to compute the object T,B,N matrix, as well as a vec4 version in homogeneous coordinates to multiply with the modelview matrix.

The next step is to set the local modelview matrix and calls `setWorldPos()` to compute the vertex world position, as well as the clip position of each vertex (using the camera's projection matrix).

```
        self.setTranslationRotation(rx, ry, rz, tx, ty, tz)

    def setTranslationRotation(self, rx, ry, rz, tx, ty, tz):
        self.rx=rx; self.ry=ry; self.rz=rz
        self.tx=tx; self.ty=ty; self.tz=tz
        self.rotation = torch.matmul(torch.matmul(util.torch_rotate_z(rz),
            ↪ util.torch_rotate_y(ry)), util.torch_rotate_x(rx))
        self.translation = util.torch_translate(tx, ty, tz-self.camera.
            ↪ camera_distance)
        self.setWorldPos()
```

The tangent directions on the mesh can be later accessed through the computation of the Tangent, Bitangent, Normal matrix (TBN) of each vertex in object space:

```
    def objectTBN(self):
        return normal_util.objectTBN(self.vtx_pos, self.vtx_uv, self.idx, self
            ↪ .vtx_n, self.numVertex)
```

`PointLight` and `DirLight`

They are loaded with `scene.setLight()`. Directional and point lights are used for testing BRDFs and generating baselines. They take as argument the intensity/color of the light, and the properties of direction or position.

```
class DirLight:
    def __init__(self, intensity = 10.0, dirx=0.0, diry=0.0, dirz=0.0):
        lightdirnp = np.asarray([dirx, diry, dirz])
        lightdirnp /= np.linalg.norm(lightdirnp)
        self.dir = torch.from_numpy(lightdirnp.astype(np.float32)).cuda()
        self.intensity = torch.tensor(intensity).cuda()
```

```
class PointLight:
    def __init__(self, intensity = 10.0, x=3.0, y=3.0, z=-3.0):
        pointlightposnp = np.asarray([[x, y, z]])
        self.pos = torch.from_numpy(pointlightposnp.astype(np.float32)).cuda()
        self.intensity = torch.tensor(intensity).cuda()
```

`PolygonalLight`

It can be loaded to the scene using `scene.setQuad()`. In the current version of the code base, we only support quadrilaterial lights. The `PolygonalLight` takes as argument the light intensity, which is constant over the mesh:

```
class PolygonalLight:
    def __init__(self, camera, intensity=1.0, twoSided = False, clip = True,
            ↪ xscale=2.0, yscale=2.0, rx=0.0, ry=0.0, rz=0.0, tx=0.0, ty
            ↪ =1.0, tz=0.0):
```

```
        self.camera = camera
        self.intensity = torch.tensor(intensity, device=torch.device('cuda:0')
            ↳ )
        self.quadcolor = torch.tensor([self.intensity, self.intensity, self.
            ↳ intensity], device=torch.device('cuda:0'))
```

We note that in the webGL fragment shader provided by [HDHN16], the intensity of the polygonal light is set to 4.0 by default. The polygonal light also inherits the geometric properties of a mesh, such as scale, position, translation and normal:

```
        self.xscale = xscale
        self.yscale = yscale
        self.quadpoints0 = torch.tensor([[-self.xscale, self.yscale, 0.0,
            ↳ 1.0], [self.xscale, self.yscale, 0.0, 1.0], [self.xscale, -
            ↳ self.yscale, 0.0, 1.0], [-self.xscale, -self.yscale, 0.0,
            ↳ 1.0]]).cuda()
        self.setTranslationRotation(rx, ry, rz, tx, ty, tz)
        self.idx = torch.tensor([[0,1,2], [2,3,0]], dtype=torch.int32).cuda()
        self.quad_n = torch.tensor([[0.0, 0.0, 1.0], [0.0, 0.0, 1.0], [0.0,
            ↳ 0.0, 1.0], [0.0, 0.0, 1.0]]).cuda()
        self.quad_n_transpose = torch.transpose(self.quad_n, 0, 1)
```

It finally allows to set `twoSided = True` if we want the light emitted on both sides, and to set `clip = True` if the polygonal light is clipped to the upper hemisphere of each shading point. Currently, our polygonal light clipping algorithm only supports rectangular or triangular lights.

```
        self.twoSided = constants.ZERO
        if twoSided:
            self.twoSided = constants.ONE
        self.clip = clip
```

Material

It can be loaded to the scene with `scene.setMaterial()`, and gets applied to the surfaces of the `Mesh`. A `Material` takes as arguments the roughness parameter, a BRDF name, a diffuse color and a specular color.

```
class Material:
    def __init__(self, roughness, BRDF = 'ggx', diffuseR=1.0, diffuseG=1.0,
            ↳ diffuseB=1.0, specularR=1.0, specularG=1.0, specularB=1.0):
        self.setDiffuse(diffuseR, diffuseG, diffuseB)
        self.setSpecular(specularR, specularG, specularB)
        self.setRoughness(roughness)
        self.brdf = BRDF
```

In our implementation, the reflectance of the material is the sum of a specular BRDF (GGX in this case), and a diffuse Lambertian BRDF. The specular color is used as a factor for the specular BRDF, and the diffuse color is mulitplied with the Lambertian BRDF.

Three base BRDFs are implemented in the renderer: Blinn-Phong (`blinnphong`), Cook-Torrance (`cooktorrance`) and GGX (`ggx`). As the `Roughness` parameter does not have a true physical interpretation, its appearance varies depending on the BRDF chosen. For example, for a realistic result, the range for `Roughness` is between $[20, 200]$ for Blinn-Phong, $[0.05, 0.4]$ for Cook-Torrance, and $[0.2, 0.8]$ for GGX. If we render the scene with LTC, the range of `Roughness` is $[0.01, 1]$.

Texture

It can be loaded as a normal map into the scene with `scene.setNormalMap()` or as an image texture with `scene.setTexture()`. It can also be loaded into a prefiltering algorithm to generate a filtered version. `Texture` takes as argument the name of the image file, the option to enable or disable the mip mapping, the maximum mip level constructed and the option to supersample at render time.

In the current version of the code base, we only use textures as normal maps, but they can easily be used as albedo maps or roughness maps with little reworking of the code. We therefore keep the name of the class as `Texture`.

```
class Texture:
    def __init__(self, filename, enable_mip = False, superSample = False,
        ↳ max_mip_level = None):
        self.setTexture(filename)
        self.enable_mip = enable_mip
        if self.enable_mip:
            self.setMip(max_mip_level, superSample)
        else:
            self.mip_object = None
            self.filter_mode='linear' # default for non mip-mapped normal maps
            self.superSample = False
```

`Texture.setTexture` loads the data from a file using `dataloader.dataFromImg()`, which acceps all formats supported by the pillow library. Since textures can be used for normal maps, `Texture.setTexture` also shifts the values to the $[-1, 1]$ range (to allow for negative normals).

```
    def setTexture(self, filename):
        self.data = dataloader.dataFromImg(filename)
        self.torch_tex = torch.from_numpy(self.data.astype(np.float32)).cuda()
            ↳  * 2.0 - 1.0 # allow for negative normal coordinates
        self.torch_tex = torch.nn.functional.normalize(self.torch_tex, dim=2)
        self.dimension = float(self.torch_tex.size(dim=0))
```

If mip-mapping is enabled, `nvdiffrast.torch.texture_construct_mip()` is called to create the mip hierarchy.

```
    def setMip(self, max_mip_level = None, superSample = False):
        self.max_mip_level = max_mip_level
        self.mip_object = dr.texture_construct_mip(torch.unsqueeze(self.
            ↳ torch_tex,0),max_mip_level=self.max_mip_level)
        self.filter_mode='linear-mipmap-linear'
```

```
        self.superSample = superSample
```

3.3.4 Loading LTC parameters

A scene can also take as argument lookup tables (LUTs) containing LTC parameters, that will be used to shade over the polygonal light at rendertime. All LUTs are copied as GPU tensors when they are loaded, similarly to how a conventional rendering pipeline sets a texture to be GPU accessible.

- `IsotropicLUT` (loaded with `scene.setIsotropicLUT()`) is organized in the same way as the method presented in [HDHN16]. It is a 2 dimensional LUT tabulating over n_θ view elevations, n_α material roughness, and storing 4 matrix parameters at each point. The LTC parameter texture and the LTC magnitude texture can either be from binary files or from textures such as Microsoft's DDS format. Our current implementation sets $n_\theta = n_\alpha = 16$.

- `AnisotropicLUT` (loaded with `scene.setAnisotropicLUT()`) comes as a numpy array and has a structure based on the implementation of [KHDN22]. It tabulates over view elevations, view azimuths, and optionally over material roughness, 9 matrix parameters at stored at each point.

- `FilteredLUT` (loaded with `scene.setFilteredLUT()`) is the prefiltered LTC look up table from our contribution, presented in section 3.4.

3.3.5 The rasterizer

After the scene is created, we can call the function `scene.render()` to display it. If this is the first frame or if scene geometry has moved since the last frame, the attribute `scene.rasterized` is set to `False`, meaning we first call the rasterizer.

```
    def render(self, glctx, mode, antialias = False, renderOption = False):
        if not self.rasterized:
```

```
            self.rasterize(glctx)
```

In the rasterizer, we start by concatenating the vertex positions in clip space, and triangle ID for the meshes:

```
    def rasterize(self, glctx):
        self.all_clip = torch.cat((self.mesh.pos_clip, self.quad.quad_clip),
            ↳ 1)
        self.all_idx = torch.cat((self.mesh.idx, self.quad.idx), 0)
```

Then, we rasterize the triangles using the function included in nvdiffrast;

```
        self.rast_out, self.rast_out_db = dr.rasterize(glctx, self.all_clip,
            ↳ self.all_idx, resolution=[self.resolution, self.resolution])
```

This creates the screen buffer using the `scene.resolution` parameter, and we can see that our current implementation generates a square image.

We create a mask for the polygonal light, and a mask for all the triangles, if we need to render them on a different layer. This is possible because we added an offset to all the vertex IDs belonging to the polygonal light triangles:

```
        self.lightMask = torch.clamp(self.rast_out[..., -1:] - self.mesh.
            ↳ numTriangles, ZERO, ONE)
        self.triangleMask = torch.clamp(self.rast_out[..., -1:], ZERO, ONE)
```

We interpolate pixel attributes that will be useful in our renderers, specifically the normal direction and positions in world space at each pixel. First, we create 2 buffers containing the 3D position and the view directions of each surface point:

```
        self.all_positions = torch.cat((self.mesh.world_positions, self.quad.
            ↳ world_quadpoints), 0)
        self.position, _ = dr.interpolate(self.all_positions[None, ...], self.
            ↳ rast_out, self.all_idx)
        self.viewdir = torch.nn.functional.normalize(-self.position, p=2.0,
            ↳ dim=3) #camera is always at 0,0,0
```

Finally, the rasterizer creates screen buffers of the tangents and normal directions at each pixel.

```
        self.computeNormalAndTangentBuffers()
```

This function `computeNormalAndTangentBuffers()` either interpolates UVs and apply a normal map, or just interpolates the surface normals using the vertex attributes. If we need to compute the view azimuth in the renderer, we also ensure that we interpolate a tangent direction at each pixel (using the object TBN matrix). We finally give the option to enable normal map superSampling, and in this case each pixel will store all the normals in its footprint.

```
    def computeNormalAndTangentBuffers(self):
        if self.hasFilteredLUT: # get the geometric normal and tangent buffers
            self.surfacenormalimage, self.surfacetangentimage = normal_util.
             ↪ surfaceNormalsAndTangents(self)
        elif self.computeTangents:
            self.normalimage, self.tangentimage = normal_util.
             ↪ surfaceNormalsAndTangents(self)
        if self.hasNormalMap: # apply a normal map
            if self.normalMap.enable_mip: # uv image with mip map enabled
                self.uvimage, self.uvimaged = dr.interpolate(self.mesh.vtx_uv[
             ↪ None, ...], self.rast_out, self.mesh.idx, rast_db=self.
             ↪ rast_out_db, diff_attrs='all')
            else: # uv image with mip map disabled
                self.uvimage, _ = dr.interpolate(self.mesh.vtx_uv[None, ...],
             ↪ self.rast_out, self.mesh.idx)
            if self.normalMap.superSample:
                self.normalimage, self.normalsPerPixel = normal_util.
             ↪ exhaustiveNormalMap(self)
            else:
                self.normalimage = normal_util.normalMap(self)
        else: # do not apply a normal map and do not compute tangent
            self.normalimage = normal_util.surfaceNormals(self.rast_out, self.
             ↪ mesh.mv_33, self.mesh.vtx_n_transpose, self.mesh.idx)
```

We finally set `scene.rasterized` to `True`, and we can proceed to a renderer. The choice of renderer is given to `scene.render()` via the String argument `mode`.

3.3.6 The baseline renderers

Rendering algorithms have been implemented in the `./src/renderer.py`, and play the same role as fragment shaders in a regular rendering pipeline. To emulate this in our pytorch framework, a renderer gets pixel attributes (such as position, normal, tangent) in GPU tensors representing image buffers. For example, if the scene resolution is $r \times r$, the renderer receives the pixel normal directions in a buffer of size $r \times r \times 3$, and output a $r \times r \times 3$ tensor (3 color channels). This allow every pixel to be handled concurrently on the GPU.

The renderers take as argument the `Scene` instance and use the attributes necessary to compute the color at each pixel. This includes the output from the rasterizer, the normal direction at each pixel, the 3D position of each vertex and light in the scene, etc.

The most basic type of renderer we implemented is `renderPointLit()` and it is called with the mode `pointlit`. It computes unshadowed direct illumination from a single point light with a BRDF.

We start by computing the direction and squared distance from the light to each point.

```
def renderPointLit(sc):
    lightdir = torch.nn.functional.normalize(sc.pointlight.pos.sub(sc.position
            ↳ ), p=2.0, dim=3)
    dsquared = torch.squeeze(torch.pow(torch.cdist(sc.position, sc.pointlight.
            ↳ pos), 2), 3)
```

Then, we compute the amplitude of each BRDF in the direction of the light (in this case, we chose GGX as our specular BRDF):

```
    diffuse = BRDF.lambertianEval(sc.normalimage, lightdir)
    specular = BRDF.evaluate(sc.mat.brdf, sc.mat.roughness, sc.normalimage,
            ↳ lightdir, sc.viewdir)
```

And finally we sum the energy contribution from the ambient, diffuse and specular terms.

```
color = torch.mul(torch.div(specular + diffuse, dsquared), sc.pointlight.
        ↳ intensity)
colorRGB = torch.stack((color, color, color),3)
return colorRGB
```

The other basic renderer is `renderDirLit()` (called with the mode `dirtlit`) and computes the illumination at each point from a fixed lighting direction. Some debugging renderers are also available, to display various properties on the geometry; for example `normals`, `tangents`, `texeldensity`, etc.

3.3.7 LTC renderers

Some renderers in `./src/renderer.py` are designed to render a scene under polygonal light illumination using a lookup table of LTC parameters. Depending on the mode argument provided to `scene.render()`, the corresponding LTC renderer is executed.

```
def render(self, glctx, mode, antialias = False):
    if not self.rasterized:
        self.rasterize(glctx)
```

We have the option of rendering a scene with our filtered method (described in section 3.4.2):

```
    if mode == 'filtered': # Our method
        if self.hasFilteredLUT:
            self.RGBcolor = renderer.renderLTCLUT(self)
```

or with the baseline methods:

```
    elif mode == 'isotropicLTC': # Eric Heitz (2016) method
        if self.hasIsotropicLUT:
            self.RGBcolor = renderer.renderLTCIsotropic(self)
    elif mode == 'anisotropicLTC': # Aakash KT (2022) method
        if self.hasAnisotropicLUT:
            self.RGBcolor = renderer.renderLTCAnisotropic(self)
```

The method `renderer.renderLTCIsotropic()` is a re-implementation of [HDHN16], adapted to our framework, and `renderer.renderLTCAnisotropic()` is a re-implementation of [KHDN22]. These two renderers are used to generate ground truth and naive baselines with which to compare our own method.

Finally, `renderer.renderLTCAnalytic()` does not load a LUT, but enables the rendering of the scene using an analytic formula for the LTC parameters. Using differentiable rendering, we can optimize an analytic function that take as argument the view elevation and/or the material roughness, and outputs the 4 LTC parameters we need to reconstruct the lobe. It is an experimental rendering mode but shows promising results, as can be seen on the main github repository.

```
        elif mode == 'analyticLTC': # Analytic experimental method
            if self.hasAnalyticLTC:
                self.RGBcolor = renderer.renderLTCAnalytic(self)
```

In the last part of `scene.render()`, we add ambient lighting (if any):

```
        if self.ambient:
            self.RGBcolor = self.RGBcolor + ONE * self.ambientColor
```

we also mask out the geometry to replace background color, and we mask out the polygonal light to display its color:

```
        self.RGBcolor = torch.where(self.triangleMask == ZERO, ZEROPOINTFIVE,
             ↪ self.RGBcolor) # mask background
        self.RGBcolor = torch.where(self.lightMask == ONE, WHITE, self.
             ↪ RGBcolor) # mask light quad
```

and finally, we antialiase the image if enabled:

```
        if antialias:
            self.RGBcolor = dr.antialias(self.RGBcolor, self.rast_out, self.
             ↪ all_clip, self.all_idx)
        return self.RGBcolor
```

Render using isotropic LTC [HDHN16]

As an example, we present here a high level overview of our implementation of the renderer in [HDHN16], provided in this webGL fragment shader. We start by computing the cosine of the view elevation at each point:

```
def renderLTCIsotropic(sc):
    costheta = torch.sum(sc.normalimage * sc.viewdir, dim=3)
```

We fetch and interpolate the 4 LTC parameters in the 2D lookup table, for each fragment. In the texture, the u direction is the material roughness, and v is the view elevation.

```
    uv = torch.nn.functional.pad((torch.acos(costheta)/HALFPI).unsqueeze(3),
            ↳ (1, 0), mode='constant', value=sc.material.roughness)
    uv[:,:,:,0] = uv[:,:,:,0] * (sc.LUT_size - 1.0)/sc.LUT_size + 0.5/sc.
            ↳ LUT_size
    uv[:,:,:,1] = uv[:,:,:,1] * (sc.LUT_size - 1.0)/sc.LUT_size + 0.5/sc.
            ↳ LUT_size
    LTC_parameters_image = dr.texture(sc.LTC_param_tex[None, ...], uv,
            ↳ filter_mode='linear')
```

We then reconstruct the full inverse matrix M^{-1} from the 4 parameters, by filling the rest of the matrix with 0:

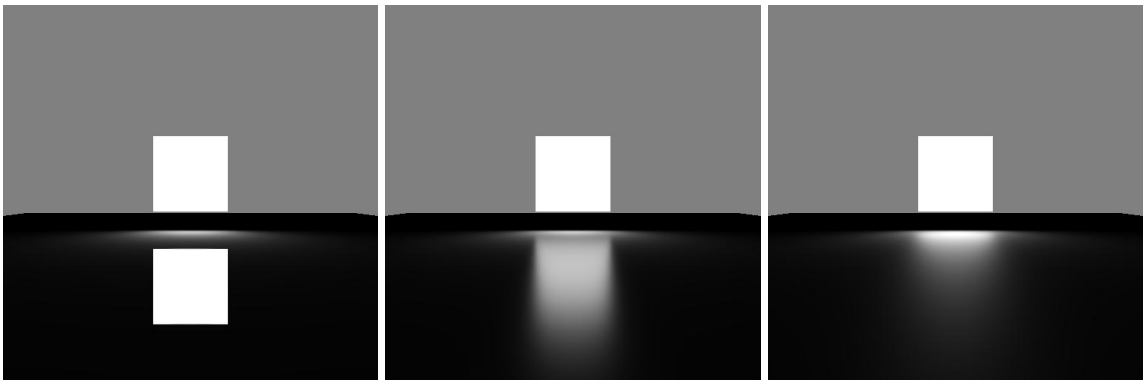
```
    Minv = LTC.transformationMatrixFromFourParameters(LTC_parameters_image)
```

And we compute the coordinates of the polygonal light points, with respect to each surface point, along with the transformation matrix associated with each surface point:

```
    L = LTC.computeLocalLightPoints(sc)
    localmat = LTC.computeLocalMat(sc, costheta)
```

Finally, we integrate the LTC lobes over the polygonal domain (details are omitted):

```
    diff = LTC.renderLTC(sc, sc.diffmatrix, localmat, L) * sc.material.
            ↳ diffusecolor
    spec = LTC.renderLTC(sc, Minv, localmat, L) * sc.material.speccolor
    return spec + diff
```



(a) (b) (c)

Figure 3.5: Isotropic LTC renders using the standard Scene in `./src/sampleScenes.py`. We set the roughness parameters to (a) 0.01 (b) 0.3 (c) 0.5. The resolution of each render is 512×512.

To see the implementation of other renderers such as `renderLTCAnisotropic()`, please visit `./src/renderer.py`.

3.4 Implementation of our method

3.4.1 Normal map prefiltering

The offline prefiltering step, located in `./src/fitEBRDF.py`, can be ran once for each normal map and each material roughness. It is based on the fitting algorithm presented in [KHDN22].

Loading a normal map

Our model is explicit, meaning that we support arbitrary normal map inputs. The first step is to load an image texture as a GPU tensor, and normalize each texel so that it represents a tangent space normal vector. We denote m the number of texels on one side of the square normal map.

```
normalMap = Texture(f'{datadir}/tex/stone128.jpg').torch_tex
```

Normal maps of size up to $m \times m = 128 \times 128$ can be loaded in the pre-filtering algorithm (figure 3.6).

Since we want to leverage parallel GPU computation, we want to fit all data in video memory. We are filtering the normal map in texture space, and using the pytorch framework, we process all the texels of the base texture in parallel. When $m = 128$, the base BRDF sampling buffer (described in section 3.4.1) occupies $(128 \times 128) \times (16 \times 8 + 2 \times 4) \times 128 \times 3\times$ FLOAT32 = 3.22 GB. When $m = 256$ (we want to use powers of 2), the same buffer occupies $(256 \times 256) \times (16 \times 8 + 2 \times 4) \times 128 \times 3\times$ FLOAT32 = 12.88 GB. It makes it impossible to fit the buffer in GPU memory for $m \geq 256$, the limitation being our access to 8 GB of video memory. This is the main reason why we do not filter larger normal maps.

The textures we use are seamless, we can therefore visualize larger normal maps by tiling the map at runtime. Furthermore, absolute normal map resolution is not crucial when working on aliasing; the important quantity is the ratio of texel to pixel.

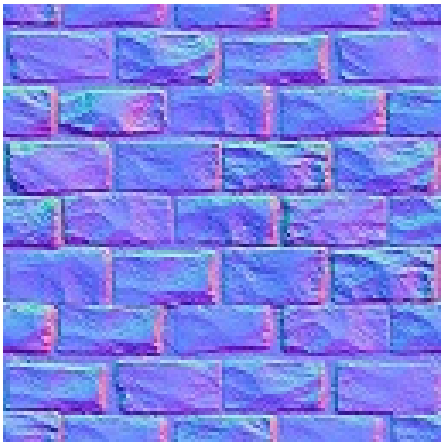


Figure 3.6: STONE normal map (128×128). Source: Sergün Kuyucu (https://www.artstation.com/artwork/kbnmy).

View tabulations

We prepare an array of n_θ view elevations θ and n_φ view azimuths φ. These view directions are the final angles at which we will tabulate our results, therefore we express them in the surface tangent frame, i.e. with respect to the geometric normal n_g (figure 3.7).

```
def getThetaPhis(nTheta, nPhi):
    thetaList = []
    phiList = []
```

The range of θ is $[0, \pi/2]$, computed from n_g.

```
    if nTheta == 1: thetaList.append(0.1)
    else:
        for theta in range(nTheta):
            t = theta / (nTheta-1.0) * 0.999 * math.pi/2.0
            thetaList.append(t)
```

The effective BRDF will be highly assymmetric due to the normals in the pixel footprint, and the reflectivity will change radically when rotating the view around the geometric normal. Thus, we do not limit the range of φ to $[0, \pi/2]$ range like in [KHDN22] (we do not have any symmetry to leverage). Instead, we tabulate our view azimuths over the whole circle $[0, 2\pi]$.

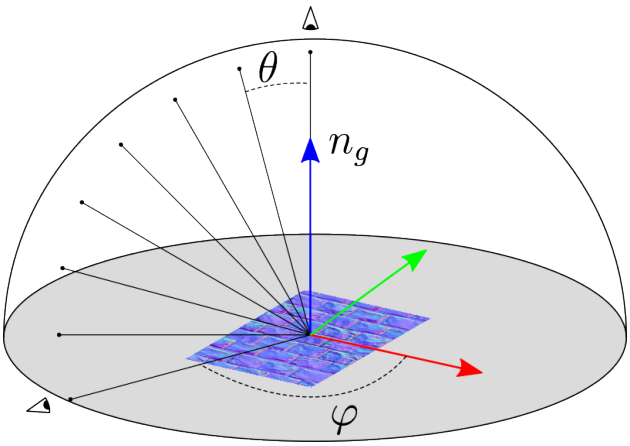


Figure 3.7: 8 tabulated view elevations θ with the same azimuth φ, with respect to the geometric/surface tangent frame. For visual simplicity, we do not show the true magnitude of the x, y and z axis, which is 1.

```
    if nPhi == 1: phiList.append(0.1)
    else:
        for phi in range(nPhi):
            p = phi / (nPhi-1.0) * math.pi * 2.0
            phiList.append(p)
    return thetaList, phiList
```

Finally, we found that having the azimuthal angular resolution being half the elevation angular resolution yielded acceptable results, so we chose $n_\theta = 8$ and $n_\varphi = 16$.

Sampling the base BRDF

The base BRDF we sample is a GGX microfacet model, where we choose the roughness parameters $\alpha_x, \alpha_y = 0.01$ (highly specular). If we want to try a different base BRDF roughness, it must be changed here (and the pre-filtering process has to be run again).

We importance sample this function for `samplesPerEBRDF` directions at each texel belonging to the full resolution normal map (MIP 0), for all views v (θ, φ). We choose `samplesPerEBRDF = 128` samples in each GGX lobe, which provides a good compromise between fitting accuracy and lightweight storage.

```
baseSamples = generateBaseGGXsamples_GPU(elevations, azimuths, nTheta, nPhi,
            ↳ samplesPerEBRDF, normalMap, alphax, alphay, diffuse=diffuse)
```

First, for each tabulated view v and each MIP 0 texel q, we compute the view elevation θ_q with respect to the microfacet normal $n(q)$. We clamp this elevation to positive values since we cannot shade a texel if the view is underneath the microfacet.

```
def generateBaseGGXsamples_GPU(elevations, azimuths, nTheta, nPhi,
            ↳ samplesPerEBRDF, normalMap, alphax, alphay, diffuse = False):
    with torch.no_grad():
        ng = torch.tensor([0.0, 0.0, 1.0], dtype=torch.float, device=torch.
            ↳ device('cuda:0')) # geometric normal
        ngimage = ng.repeat(texSize, texSize, 1)
        samplesPerEBRFMore = int(samplesPerEBRDF * 1.4)
        allGGXSamples = []
        for elevation in elevations:
            for azimuth in azimuths:
                vg = getEyeSpherical(elevation=elevation,azimuth=azimuth,
            ↳ localdevice='cuda:0') # geometric view direction
                thetaLocal = torch.dot(vg, normalMap)
                thetaLocal = torch.acos(torch.where(thetaLocal > ZEROZEROONE,
            ↳ thetaLocal, ZEROZEROONE))
```

Then, we generate the GGX distribution in the local texel frame using a vertical normal $n_0 = (0, 0, 1)$, and a view v_0 with $\theta_0 = \theta_q$ and $\varphi_0 = 0$ (figure 3.8(a)). This is possible because the base GGX BRDF we chose is isotropic (unlike the effective BRDF which is highly anisotropic). We importance sample this distribution using rejection sampling at `samplesPerEBRDF` directions, giving us the shape of the GGX at θ_q.

```
                v0 = getEyeSphericalBatch(thetaLocal, ZERO)
                samples = utils.rejection_sampling_torch(samplesPerEBRFMore,
            ↳ v0, alphax, alphay, texSize=texSize, sampleCount=
            ↳ samplesPerEBRDF)
```

The GGX rejection sampling method is directly taken from [KHDN22].

Expressing the base BRDF in the geometric frame

The GGX samples we generated in the local microfacet frame (figure 3.8(a)) have to be expressed in the geometric frame, so that all base BRDFs in a patch are expressed in the same referential.

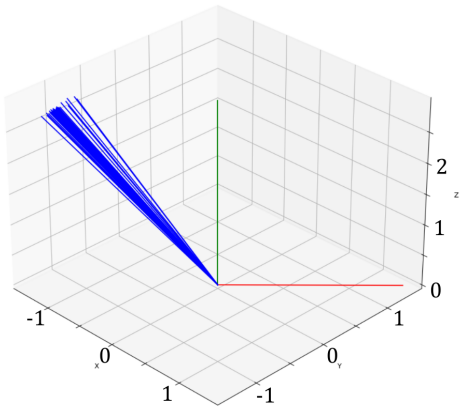
For that reason, we compute a 3D rotation R_0 so that $R_0(n_0) = n(q)$. We apply R_0 to v_0 to obtain v_r, and apply R_0 to all base BRDF samples (figure 3.8(b)).

```
cosThetaNormal = torch.sum(ng * normalMap, dim=2).unsqueeze(2)
sinThetaNormal = torch.sqrt(ONE - (cosThetaNormal * cosThetaNormal))
E = torch.nn.functional.normalize(torch.cross(ngimage, normalMap), dim=2)
vr = rotateAroundTorch(pivot=E, vector=v0, cosAngle=cosThetaNormal, sinAngle =
    ↳  sinThetaNormal)
samples = rotateAroundTorch(pivot=E.unsqueeze(2).repeat(1,1,samplesPerEBRDF,1)
    ↳ , vector=samples, cosAngle=cosThetaNormal.unsqueeze(3),
    ↳ sinAngle = sinThetaNormal.unsqueeze(3))
```

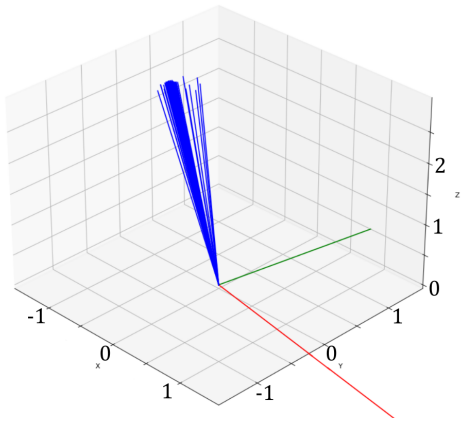
The second 3D rotation R_1, rotating around $n(q)$ so that $R_1(v_r) = v$. We also apply R_1 to all samples (figure 3.8(c)).

```
pr = torch.nn.functional.normalize(vr - torch.sum(vr * normalMap, dim=2,
    ↳ keepdim=True) * normalMap, dim=2)
pg = torch.nn.functional.normalize(vg - torch.sum(vg * normalMap, dim=2,
    ↳ keepdim=True) * normalMap, dim=2)
cosThetaNormal = torch.sum(pr * pg, dim=2, keepdim=True)
sinThetaNormal = torch.sqrt(ONE - (cosThetaNormal * cosThetaNormal))
E = torch.nn.functional.normalize(torch.cross(pr, pg), dim=2)
samples = rotateAroundTorch(pivot=E.unsqueeze(2).repeat(1,1,samplesPerEBRDF,1)
    ↳ , vector=samples, cosAngle=cosThetaNormal, sinAngle=
    ↳ sinThetaNormal)
```

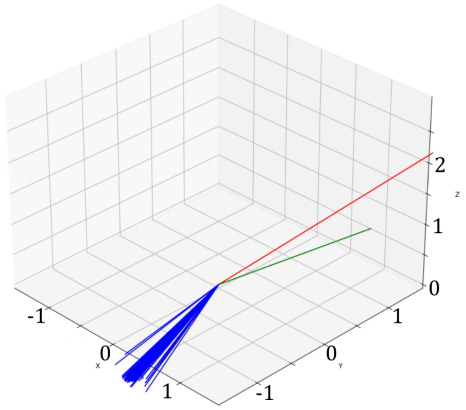
After the two rotations, some samples can have $z < 0$, but we do not clamp them at 0. We now have a list of $n_\theta \times n_\varphi$ tensors of length $m \times m \times$ samplesPerEBRDF $\times 3$, describing the GGX distribution at each texel, epressed in the geometric frame.



(a) Samples in the local microfacet frame, for a given view θ and for $\varphi = 0$. The normal n_0 is vertical.



(b) Samples after applying R_0. The normal matches the microfacet normal $n(q)$.



(c) GGX samples after applying R_0 and R_1. Both the normal and the view directions match their real direction in the geometric frame.

Figure 3.8: 128 GGX samples (blue), the view direction (red) and the normal (green), rotated to different frames. The tool we developed to generate these plots is included in our code base, in `./src/viz.py`.

Sampling the effective BRDF

As seen previously, the effective BRDF (eBRDF) can be formally written as the weighted sum of its underlying BRDFs (2.36). We have generated samples for the base BRDFs at each MIP 0 texel, and expressed them in the surface geometric frame; therefore they can be used to obtain an importance sampled eBRDF for different scales and footprints.

We note that, at the finest MIP level (MIP 0), the eBRDF of a pixel is the same as the base BRDF (but expressed in a different frame). We can therefore save storage space by not generating the eBRDF at MIP 0, and instead using the technique from [HDHN16] at runtime. We provide details about this step in section 3.4.2.

```
MIPSamples, LUT_sizes = getEBRDFsamples(nTheta, nPhi, baseSamples,
    ↪ samplesPerEBRDF, elevations, azimuths)
```

We start by iterating over footprint sizes (mip levels):

```
def getEBRDFsamples(nTheta, nPhi, baseSamples, samplesPerEBRDF, elevations,
    ↪ azimuths):
    with torch.no_grad():
        pixelSize = 12 # XYZ, float32
        allEBRDFSamples = []
        LUT_sizes = []
        viewCount = nTheta*nPhi
        size = baseSamples[0].shape[0]
        tileSize = 2
        while tileSize <= size:
            numTiles = int(size/tileSize) # number of mip texels in the mip
            LUT_sizes.append((numTiles, numTiles, nTheta, nPhi))
            numFacets = tileSize*tileSize
```

We want to use `samplesPerEBRDF` samples to represent the eBRDF, to keep the memory footprint reasonable. Thus, if there are `tileSize` MIP 0 texels in our current footprint, we stochastically select `samplesPerEBRDF/tileSize` samples in each underlying BRDF 3.9(a)). We do not need a random seed as the base samples were already generated randomly.

```
            samplesPerFacet = math.ceil(samplesPerEBRDF/numFacets)
            currentMIP_samples = torch.zeros(viewCount, numTiles, numTiles,
                ↪ samplesPerEBRDF, 3, dtype=torch.float, device='cpu')
            for elevation in range(nTheta):
                for azimuth in range(nPhi):
                    viewIndex = nPhi * elevation + azimuth
```

```
                    sampleBuffer = baseSamples[viewIndex]
                    sampleBuffer = sampleBuffer[:,:,:samplesPerFacet, :]
```

The MIP 0 texels belonging to each patch are then isolated using square axis-aligned footprints in texture space.

```
                    c = np.lib.stride_tricks.as_strided(sampleBuffer,
                    shape=(numTiles, numTiles, tileSize, tileSize,
            ↪ samplesPerFacet, 3),
                    strides=(tileSize * size * samplesPerFacet * pixelSize,
                            tileSize * samplesPerFacet * pixelSize,
                            size * samplesPerFacet * pixelSize,
                            samplesPerFacet * pixelSize,
                            pixelSize,
                            4))
```

In higher mip levels, we will have `samplesPerEBRDF < numFacets`, meaning that we cannot sample from all texels in the patch. In this case, we stochastically select `samplesPerEBRDF` microfacets using uniform sampling, and keep one sample from each.

```
                    if samplesPerEBRDF < numFacets:
                        c = c.reshape(numTiles,numTiles,numFacets,3)
                        perm = torch.randperm(numFacets, device='cpu')[:
            ↪ samplesPerEBRDF]
                        perm = perm.repeat(numTiles,numTiles,1)
                        perm = torch.stack([perm,perm,perm], dim=3)
                        c = torch.gather(c, 2, perm)
```

At these scales, the eBRDF will be highly multi-modal and our stochastic sampling will lead to a rougher surface, which is desirable [OB10]. We trust that we can capture some of the complexity of the eBRDF with the many view azimuths. Furthermore, the effective BRDF is a lower frequency signal that the NDF, and it gets blurrier as the filtering footprint gets larger (figure 3.9(c)).

Finally, we rotate all eBRDF samples about n_g, so that $\varphi = 0$ no matter the view azimuth. This ensures that the LTCs we fit to the eBRDF are all computed in a frame where the view azimuth is 0, which simplifies the transformations in the real-time part.

```
samples = rotateAroundTorch(pivot=ng, vector=samples, cosAngle=math.cos(-
    ↪ azimuths[azimuth]), sinAngle=math.sin(-azimuths[azimuth]), cpu
    ↪ =True)
```

At that point, for each filtering footprint at each scale, and for each view angle v, we have stored `samplesPerEBRDF` directional samples on the view-evaluated eBRDF. For a given mip level l containing $m_l = m/\texttt{tileSize}$ texels, the data is stored in a tensor of dimensions $m_l \times m_l \times n_\theta \times n_\varphi \times \texttt{samplesPerEBRDF} \times 3$.

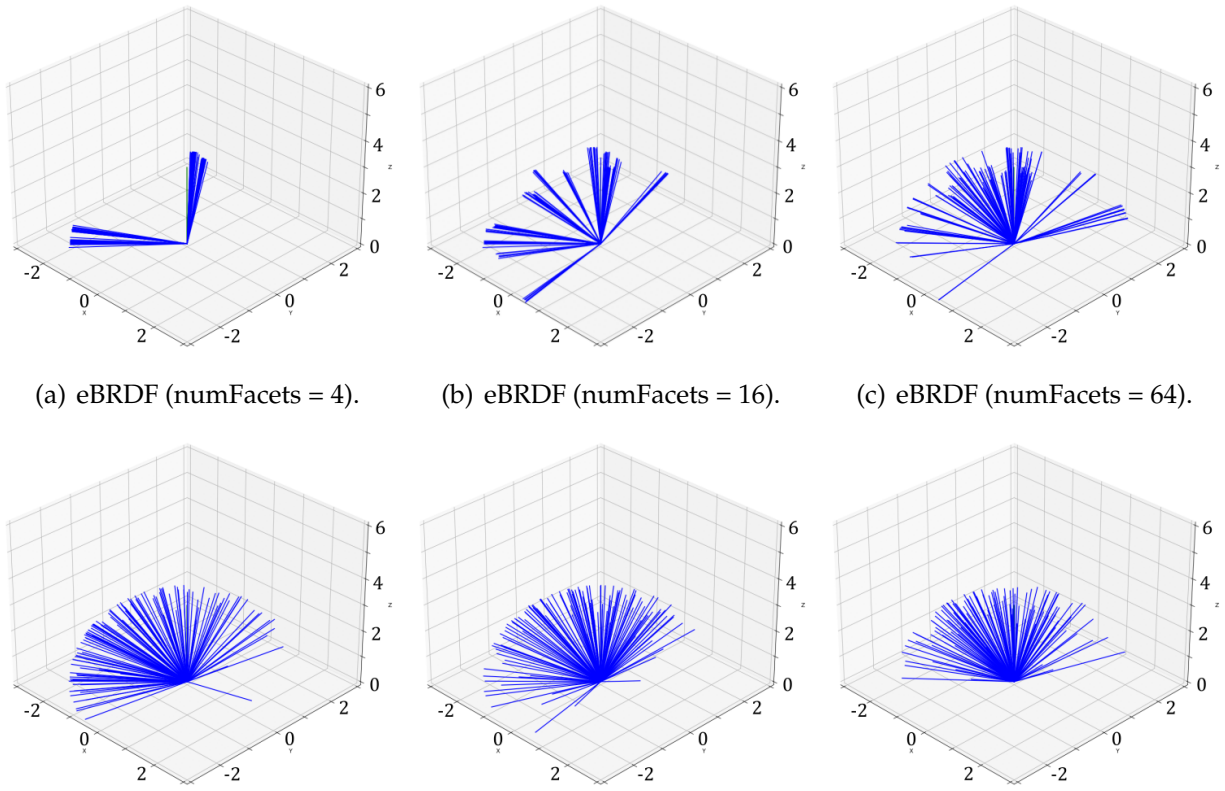
Fitting LTCs

In this step, we filter the effective BRDF at each footprint and view angle, using a single LTC lobe (figure 3.9). We initializing a model of anisotropic LTC for each mip level, which will contain all LTC parameters for a given mip level. This means that we optimize all 9 parameters of the LTC matrix. This procedure is based on the code KT et al. [KHDN22].

```
for MIP_level in range(1, max_MIP+1):
        model = ltc.LTCAnisotropic(lut_size=LUT_sizes[MIP_level-1], type=torch
            ↪ .float32)
        MIP = fitLTC_EBRDF(MIPSamples[MIP_level-1].cuda(), model, MIP_level,
            ↪ LUT_sizes[MIP_level-1], samplesPerEBRDF)
```

We start by setting the parameters for a Stochastic Gradient Descent optimizer.

```
def fitLTC_EBRDF(MIPSamples, model, MIP_level, LUT_size, samplesPerEBRDF):
    epochs = 100
    lr = 1.0
    eps = 0.1
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.0)
    model.train()
    model.optimize_mat()
```



(a) eBRDF (numFacets = 4). (b) eBRDF (numFacets = 16). (c) eBRDF (numFacets = 64).

(d) fitted LTC for (a). (e) fitted LTC for (b). (f) fitted LTC for (c).

Figure 3.9: 128 effective BRDF samples and the corresponding fitted LTC samples, for a given filtering region, at different scales. For the sake of simplicity, v is vertical (aligned with n_g). For more visualization options and to interact with the 3-dimensional plots, please use the functions in `./src/viz.py`.

We then divide our samples into batches for faster backpropagation of loss, and set the parameters for our batch backpropagation. For example, for a 64×64 mip level, $n_\theta = 8$ and $n_\varphi = 16$, we have to fit 524288 ltc matrices in 32 batches of `mul_fac` = 16384 matrices.

```
NUMBER_OF_DIRECTIONS = 32
view_count = LUT_size[0]*LUT_size[1]*LUT_size[2]*LUT_size[3]
batch = max(min(int(view_count/16),32),1)
mul_fac = int(view_count / batch)
LTCSamples = samplesPerEBRDF
```

For each training epoch, we generate `LTCSamples` from the LTC model. On the first epoch, the LTC matrix is initialized as the 3×3 identity matrix. This is because our

optimization technique method always ensure a non-zero gradient. Also, initializing with guesses from optimized neighbors is useless since neighboring regions of eBRDF can be arbitrarily different. Thus, the model begins by returning samples from a cosine distribution (LTC with identity matrix), for all filtering footprints and all view angles. The following code is from KT et al. [KHDN22]:

```
    for epoch in range(epochs):
        with torch.enable_grad():
            wi_cos = utils.sample_cosine_batch(view_count, LTC_samples, dtype=
             ↳ torch.float32)
            ltc_wi = model.transform_to_target(wi_cos)
```

For each batch, we first compute some random directions from the standard normal distribution.

```
            for idx in range(0, batch):
                opt.zero_grad()
                directions = torch.randn(mul_fac, NUMBER_OF_DIRECTIONS, 3,
             ↳ device='cuda:0')
                directions = directions / torch.norm(directions, dim=2,
             ↳ keepdim=True)
```

We then compute a stochastic estimator of the Sliced Wasserstein (SW) distance [BRPP15] between the LTC distribution and the eBRDF samples, as shown in (2.28). The samples from both distributions are projected on the random directions we just generated, and this is done only for the view angles and footprints in the current batch. This code is from KT et al. [KHDN22]:

```
                lower = idx*mul_fac
                higher = (idx+1)*mul_fac


                feat_op_projected = torch.einsum('abc,aec->aeb', ltc_wi[lower:
             ↳ higher].cuda(), directions)
                feat_gt_projected = torch.einsum('abc,aec->aeb', MIPSamples[
             ↳ lower:higher], directions)
```

```
                feat_op_projected = torch.sort(feat_op_projected, dim=2)[0]
                feat_gt_projected = torch.sort(feat_gt_projected, dim=2)[0]
```

We finally compute the mean L1 loss of the projections over the random directions, and backpropagate the loss gradient to the LTC matrix. The benefit of using SW is that the gradient is not Null, regardless of the initial LTC distribution.

```
                loss = (feat_gt_projected - feat_op_projected).abs()
                loss = torch.mean(torch.mean(loss, dim=2), dim=1)
                loss = eps * torch.sum(loss)
                loss.backward(retain_graph=True)
                opt.step()
            eps *= 0.999
```

At the end of the fitting epochs, we return the LTC matrix parameters M.

```
    ltc_mat = model.LUT.detach().cpu().numpy()
    return ltc_mat
```

For each MIP level (starting at MIP 1), we write a compressed numpy table file `.npy` containing the fitted parameters of the LTC matrices. Adjacent filtering footprints are stored next to each other for fast interpolation at runtime. Each MIP level is fitted and written to disk separately; we can therefore fit each of them on a different machine if needed. The details about storage space usage can be found in the next part.

```
writeLUT(MIP, f'MIP{MIP_level}',isotropic=isotropicLTC, directoryName=
            ↳ directoryName)
```

Filtering the diffuse component

We filtered the specular component and ignored the diffuse component of the reflectance until now. Although aliasing is much more obvious with high frequency BRDFs (like the specular GGX we chose), it is still significant with the diffuse component. Directly filtering a sum of GGX and Lambertian BRDFs would be inefficient, as the effective BRDF would

lose its high frequency components and we would not achieve specular reflections in the render.

We therefore run the same sampling in `generateBaseGGXsamples_GPU()` using an ideal Lambertian base BRDF instead of GGX.

```
samples = utils.sample_cosine(samplesPerEBRDF).unsqueeze(0).unsqueeze(0).
            ↳ repeat(texSize,texSize,1,1)
```

The fitting procedure is then identical; which means that we also fit the diffuse effective BRDF with the standard LTC model. We note that whilst the Lambertian BRDF might be view independant, rotating it to the referential of the microfacet makes it view dependant (and hence, the effective BRDF too). To make storage lighter and because the filtering is at much lower frequency, we tabulate the diffuse LUT for $n_\theta = 2$ and $n_\varphi = 4$.

Post-processing LTCs

Whilst the LTC matrices may be correctly fitted to the eBRDFs, different matrices can produce the same LTC distribution [KHDN22]. Thus, interpolation between two adjacent matrices in the LUT may sometimes be ill-defined and cause artifacts. We use the parameter alignment technique presented in [KHDN22], which minimizes the average squared distance between the original cosine distribution samples, and the LTC samples. The implementation can be accessed in `processLTCs()` in `./src/fitEBRDF.py`. This ensures that we store the 'simplest' LTC matrix and guarantees the well-defined interpolation of parameters in the real time part.

Storage

For a mip level l, the final dimension of the FLOAT32 parameters tensors is $m_l \times m_l \times n_\theta \times n_\varphi \times 9$. With a normal map of size 128×128, we start storing MIP1 with $m_l = 64$, and every higher MIP level until $m_l = 1$. We store the specular component at $n_\theta \times n_\varphi = 8 \times 16$ tabulations and the diffuse component at $n_\theta \times n_\varphi = 2 \times 4$. Thus, the total size of the LUTs on disk is $(64 \times 64 + 32 \times 32 ...) \times (8 \times 16 + 2 \times 4) \times 9 \times$ FLOAT32 = 26.7 MB. This is evidently

substantial compared to the sum of a mip-mapped normal map and a regular LTC LUTs (from [KHDN22] or [HDHN16]).

When we increase the resolution of a 2D texture, its size on the disk increases $O(m^2)$, m being the length of one side of the texture. Our LUTs are 5-dimensional, and if we filter a normal map of side length m, their size on the disk also follow $O(m^2)$.

3.4.2 Real time rendering

We load the stored mip hierarchies into a scene using `scene.setFilteredLUT()` (section 3.3.4), and copy them to GPU memory. The fact that the whole hierarchy fits in video memory at once is important for performance.

Our real-time algorithm consists of applying our LUT as a texture and correctly interpolating the LTC parameters. However, we do not need to manually interpolate between mip levels of our hierarchy: we leverage nvidia's mip hardware and automatically interpolate tri-linearly between adjacent mip texels and between mip levels. Therefore, even if our texture cannot really be 'applied' yet to the surface, each region gets the correct LTC parameters fitted for that normal map location. This works because we use the same surface uv coordinates used to apply the regular normal map.

Preparation

We start by computing image buffers for the view elevation (theta) and azimuth (phi) with respect to the geometric frame (true surface normals)

```
def renderLTCLUT(sc, smoothDiffuse = False):
    costheta = torch.sum(sc.surfacenormalimage * sc.viewdir, dim=3)
    theta = torch.acos(torch.where(costheta>ZERO, costheta, ZERO))*MAXBOUND

    viewproj = torch.nn.functional.normalize(sc.viewdir - torch.unsqueeze(
            ↳ costheta, 3) * sc.surfacenormalimage, p=2.0, dim=3)
    cross = torch.cross(sc.surfacetangentimage, viewproj)
    sign = torch.sum(cross * sc.surfacenormalimage, dim=3)

    cosphi = torch.sum(sc.surfacetangentimage * viewproj, dim=3) * MAXBOUND
    phi = torch.acos(cosphi)
    phi = torch.where(sign > ZERO, phi, TWOPI - phi)
    phi = torch.remainder(phi - HALFPI, TWOPI)
```

We then compute the light point coordinates and local matrices, like in 3.3.7:

```
L = LTC.computeLocalLightPoints(sc)
localmat = LTC.computeLocalMat(sc, costheta, useSurfaceNormals=True)
```

Rendering the specular component

Then, we interpolate the uv coordinates and apply `LUT_spec` as a mip map. After trilinear mip interpolation, we are left with a buffer of size $r \times r \times (n_\varphi \times n_\theta \times 9)$ float32 values.

```
uv, dxy = dr.interpolate(sc.mesh.vtx_uv[None, ...], sc.rast_out, sc.all_idx,
            ↳ rast_db=sc.rast_out_db, diff_attrs='all')
M = dr.texture(sc.LUT_spec.MIP1, uv, uv_da=dxy, mip = sc.LUT_spec.MIP,
            ↳ filter_mode='linear-mipmap-linear', max_mip_level=None)
M = torch.transpose(torch.reshape(M, (1, sc.resolution, sc.resolution, sc.
            ↳ LUT_spec.numTheta * sc.LUT_spec.numPhi, sc.LUT_spec.
            ↳ numChannels)), 3, 4)
```

For each pixel, we fetch the two closest azimuths and closest elevation and perform software bilinear interpolation of the LTC parameters. We are then left with a buffer of size $r \times r \times 9$, each pixel containing its 9 matrix parameters.

```
M = softwareInterpolate(M, theta, phi, sc.LUT_spec.numTheta, sc.LUT_spec.
            ↳ numPhi, sc.LUT_spec.numViews, sc.LUT_spec.numChannels,
            ↳ interpolate = True)
```

We can reshape the parameters into the 3×3 matrix M, and invert it to obtain M^{-1} at each pixel.

```
M = torch.reshape(M, (1, sc.resolution, sc.resolution, 3,3))
Minv = torch.inverse(M)
```

Finally, we run this buffer of matrices through the same LTC integrator as used in the baseline.

```
MIP_specular = LTC.renderLTC(sc, Minv, localmat, L) * sc.material.speccolor
```

Rendering the diffuse component

Rendering the diffuse component is the same as the specular component, only using the table `LUT_diff`, which contains different LTC parameters and yield a different `Minv`. Both `localmat` and `L` are identical as in the specular component.

```
MIP_diffuse = LTC.renderLTC(sc, Minv, localmat, L) * sc.material.diffusecolor
```

Rendering at MIP 0 and compositing the final image

We only filtered the effective BRDF for MIP 1 and above; and we will obtain blurry results if we use our technique when the shading surface is close to the camera, that is, when the number of texels per pixel is lower than a threshold. In practice, we check that a pixel's footprint contains less than two texels in width: $d_{Texel}/d_{Pixel} < 2$, and in this case, we switch to MIP 0 rendering. Another approach is to use d_{Texel}/d_{Pixel} as a weight to interpolate between MIP 0 and our technique.

Rendering at MIP 0 boils down to shading in texel space and using the method from [HDHN16]. In other words, we point-sample the center of the pixel, compute the elevation between the view and the texel normal, and fetch the appropriate LTC fitted to the base BRDF (section 3.3.7). We finally composite the final image with both techniques.

```
    RGBMIP0 = spec + diff
    RGBMIP = MIP_specular + MIP_diffuse
    duv = dxy * sc.LUT_spec.sizeMIP0_torch
    threshold = TWO
    MIP0_flag = (torch.sum((torch.abs(duv) > threshold), dim=3)>ZERO)
    colorRGB = torch.where(MIP0_flag, RGBMIP, RGBMIP0)
    return colorRGB
```

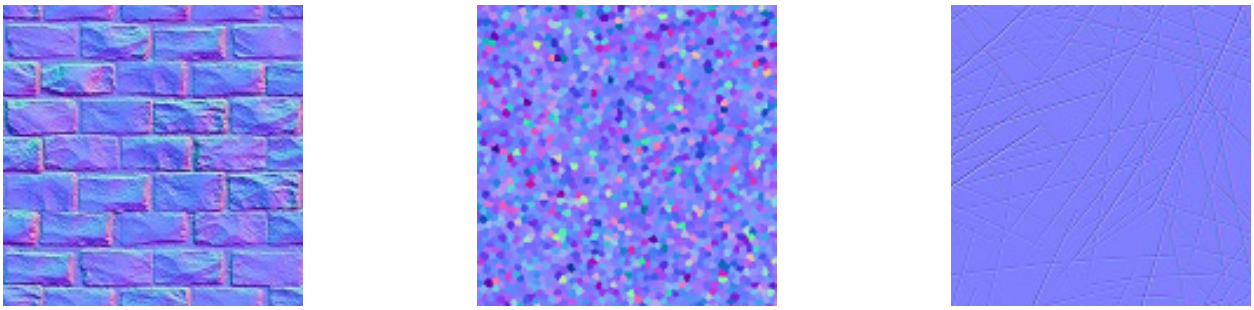
Chapter 4

Results

We first explain the error metrics, baseline and ground truth methods, as well as the parameters and scenes chosen to generate the results. We then present the renders generated with our system and evaluate them. Finally, we show some real time performance results.

4.1 Presentation of the results

To estimate the visual quality of our filtering method (described in section 3.1 and implemented in section 3.4), we compare it with a ground truth and with a naive baseline that also use LTC for area light integration. To the best of our knowledge, there are no real



(a) STONE (b) GLITTER (c) SCRATCH

Figure 4.1: 128 × 128 tangent space normal maps we used to test our technique.

time techniques that tackle the double integration of spatial pixel filtering and area light shading. Thus, we do not compare our results with other existing techniques.

For the ground truth, we shade in texel space by supersampling the pixel, and then average the results. This ensures that each texel in the patch contributes to the shading. For each normal, we evaluate the view elevation and use the standard technique from [HDHN16] to integrate over the light. Finally, we average the contribution of each sample to obtain the color of the pixel.

The naive baseline is a standard linear mipmap of the texture. We generate the mipmap using the constructor included in nvdiffrast, and sample it at runtime using trilinear interpolation. We do not show the results of sampling the normal map's MIP 0 (using point sampling or linear interpolation) since the aliasing has a similar appearance. After sampling a normal from the mip hierarchy, we shade the pixel using the standard technique from [HDHN16].

4.1.1 Error metrics

The first image error metric we use is the absolute Square Error (L2) between pixels, and we also compute the Mean Square Error of a whole region. MSE is suitable to evaluate the renders as it is sensitive to strong outliers such as fireflies, which are typical aliasing artifacts we are tying to remove.

We also use the FLIP difference evaluator presented in [ANAM+20]. It is a more perceptual image metric than MSE, as it reflects more accurately what a human viewer would consider different or similar. We set the FLIP algorithm with the default viewing parameter of 67 pixels per degree. We also compute the mean FLIP over whole images as a general similarity score.

4.1.2 Parameters chosen

We apply a normal map of dimensions 128×128 to a plane, and tile the texture 5×5 times to obtain a normal map of dimensions 640×640. We judged that it represented enough surface detail, especially at smaller scales, and thus we did not need to filter a larger texture. Furthermore, it allowed us to save on storage space as we could re-use each filtered texel 25 times during render. The normal maps used in these results are presented in figure 4.1.

We chose a sharp GGX as our specular base BRDF (roughness of 0.1), since it is more prone to shading aliasing. We use it both in the pre-filtering step of our technique, and for LTC lookup in the ground truth and baseline techniques. For our technique, we prefilter the specular component at a resolution of 8 elevations θ and 16 azimuths φ. The diffuse component is prefiltered separately, using a lambertian base BRDF (as specified in the method), and its tabulation resolution is 2 elevations θ and 4 azimuths φ.

For precomputations, we use a NVIDIA GeForce RTX 2070 SUPER GPU with 8GB video memory (a source of limitations in our prefiltering algorithm). We have access to 64GB RAM and a Intel(R) Xeon(R) W-2255 CPU @ 3.70GHz with 10 cores. With our tabulation settings ($n_\theta = 8$, $n_\varphi = 16$, 128x128 normal map), the filtering of the specular component, takes around 4 hours, and it takes around 15 minutes for the diffuse component.

4.1.3 The scenes

For each normal map, we setup two scenes to evaluate the results (figure 4.2).

We study the visual difference between the three techniques when the camera distance increases, for both scenes. As we get further, the plane occupies less screen space, thus we magnify the image with nearest neighbor extrapolation to compensate. This has the same effect as lowering the display resolution or increasing the number of samples per pixel.

We then study the difference between the techniques when the scale of the polygonal light increases. The energy output by a polygonal light is proportional to its surface area. To make the computation of error fair as the light scale changes, we therefore decrease the intensity of the light proportionally to its size.

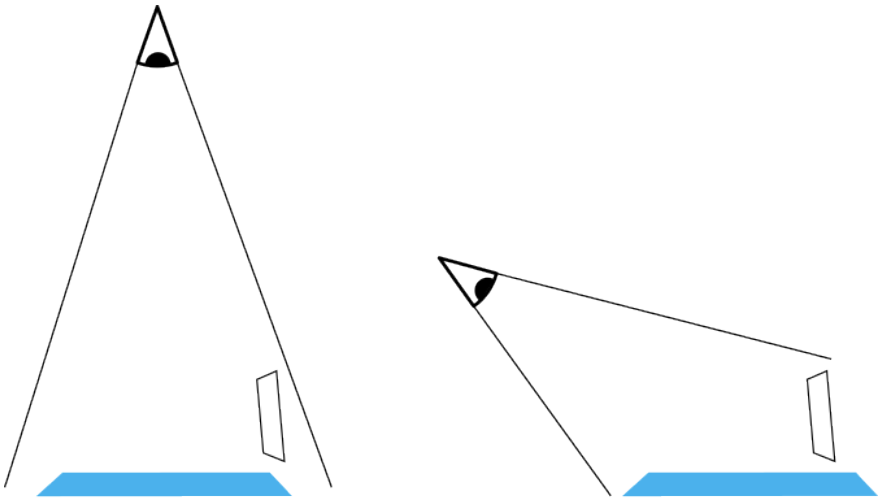


Figure 4.2: (a) SCENE 1. the view to the shading surface is vertical so we are close to $\theta = 0$ for all pixels. (b) SCENE 2. The plane is viewed at a low elevation and the light is facing the camera. In general, a grazing view angle or light angle is a more difficult case to filter, thus we expect visual quality to be generally better in SCENE 1 than in SCENE 2.

4.2 STONE

We start to test our method using the STONE normal map, created by Sergün Kuyucu at https://www.artstation.com/artwork/kbnmy.

This normal map has large bumps and ridges, which exacerbates the effects of aliasing in both the diffuse and specular components (mipmapping in figures 4.3 and 4.11). Please see specific comments on the figures.

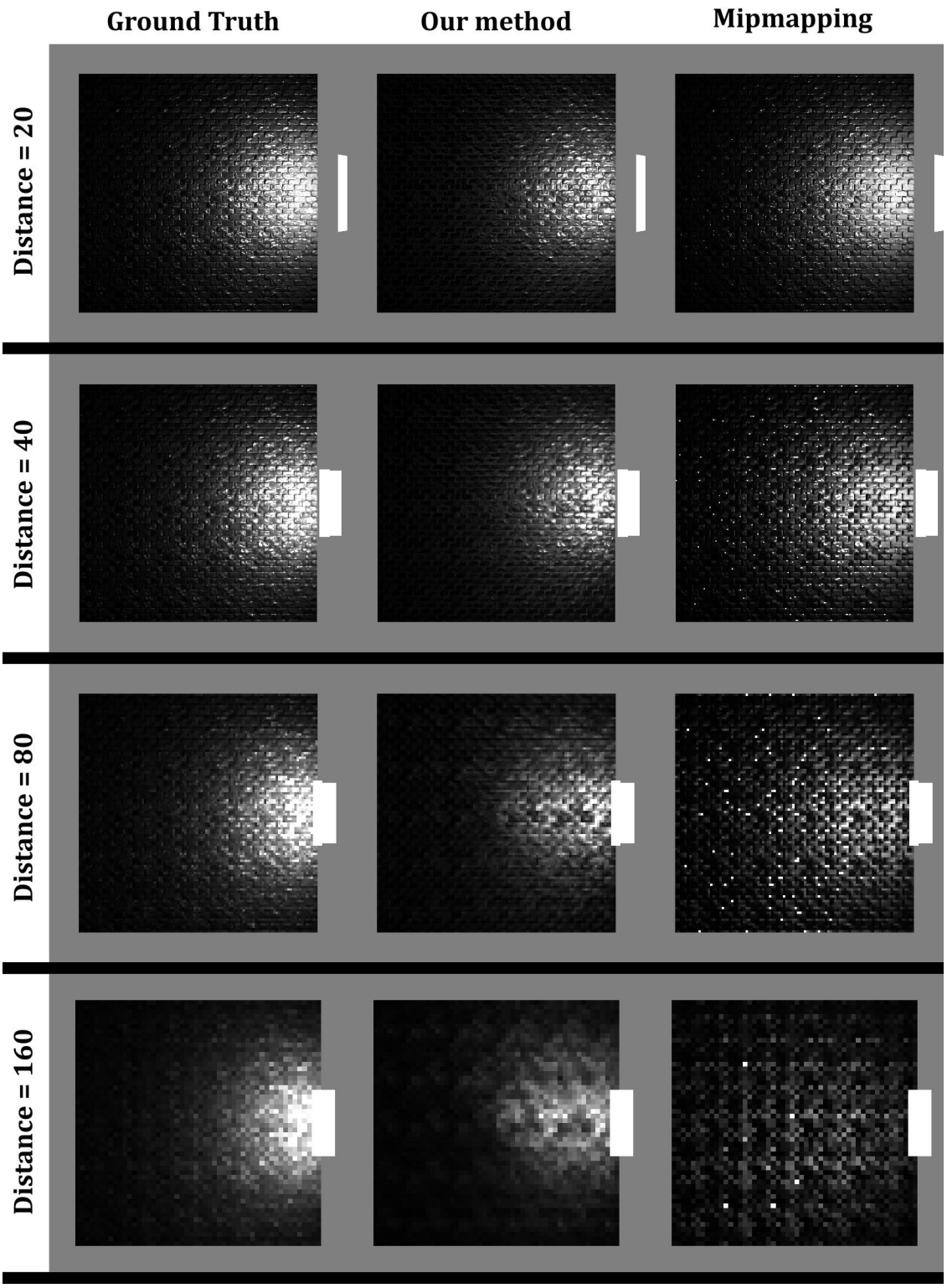


Figure 4.3: Visual comparison of the three methods for several view distances for SCENE 1. We see that our method yields a result free of aliasing at all scales, but it also displays a small energy loss. As expected, the naive mipmapping technique shows heavy aliasing. It has some fireflies at a closer scale, and even its diffuse component breaks down as the view distance increases.

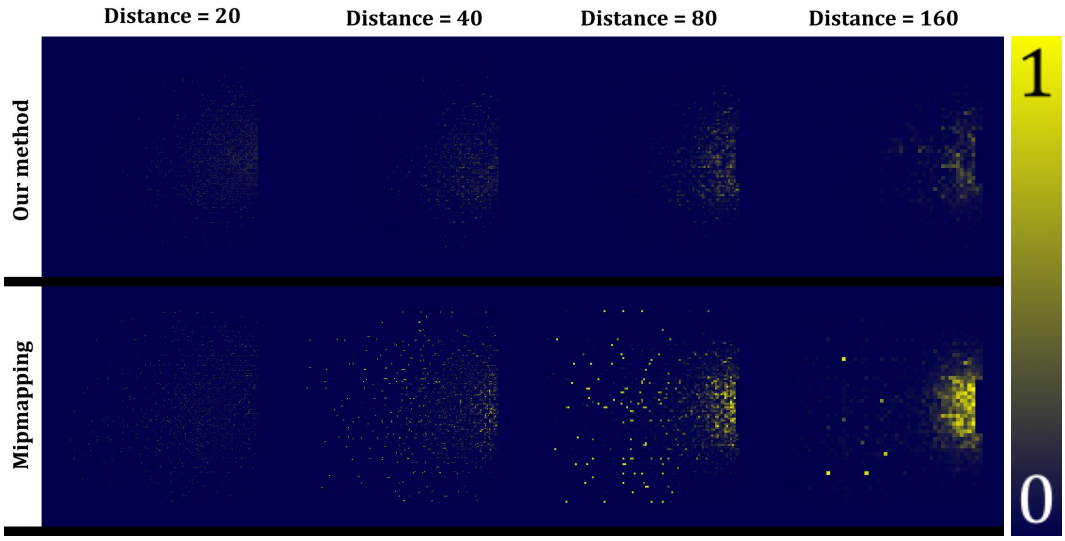


Figure 4.4: Squared error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1.

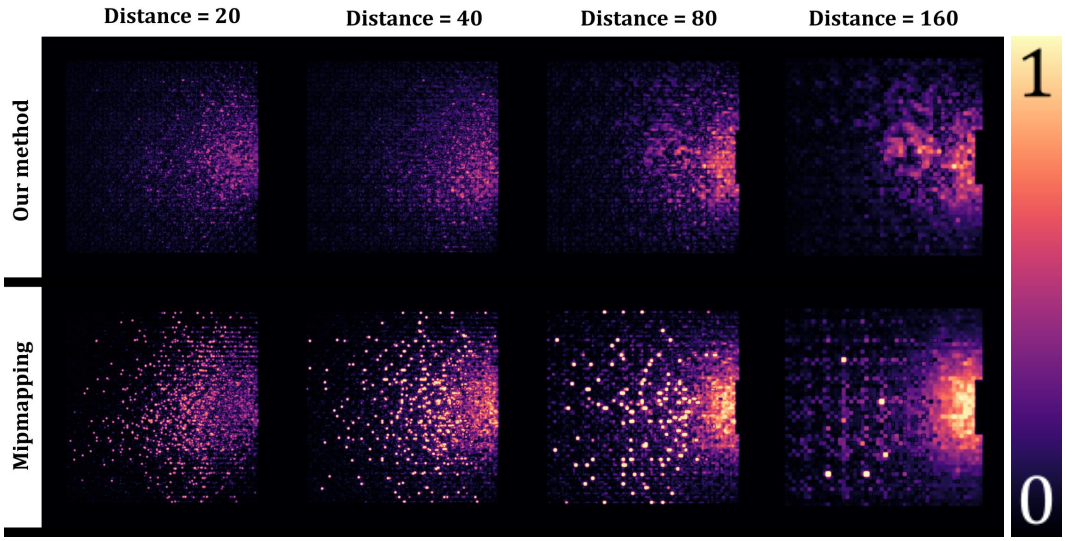


Figure 4.5: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1.

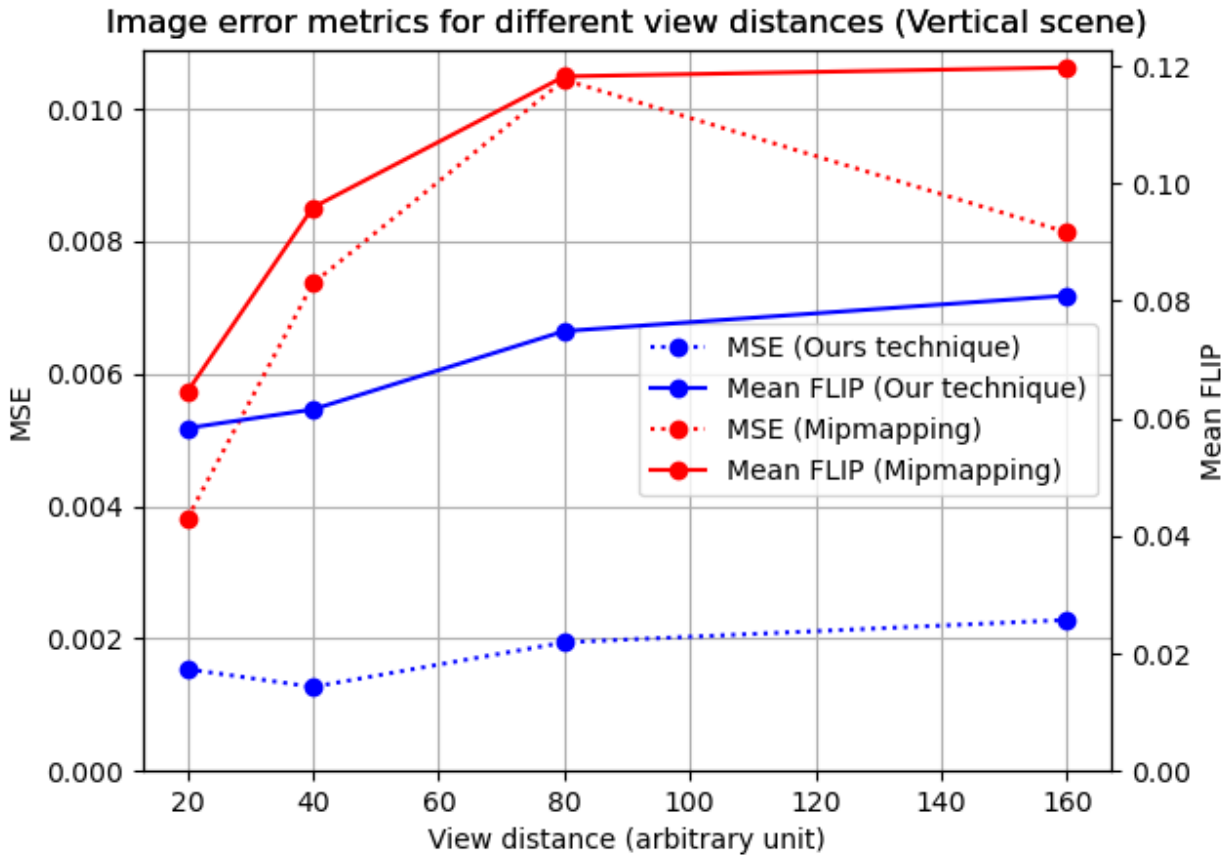


Figure 4.6: Error metrics pooled over the whole images, between our method and the ground truth (blue), and between the naive method and the ground truth (red) as a function of view distance, for SCENE 1. We can see that at a closer distance, both techniques have similar MSE and FLIP scores, as barely any filtering is needed. As the view distance increases, our technique compares better to the ground truth than naive mipmapping, for both error metrics.

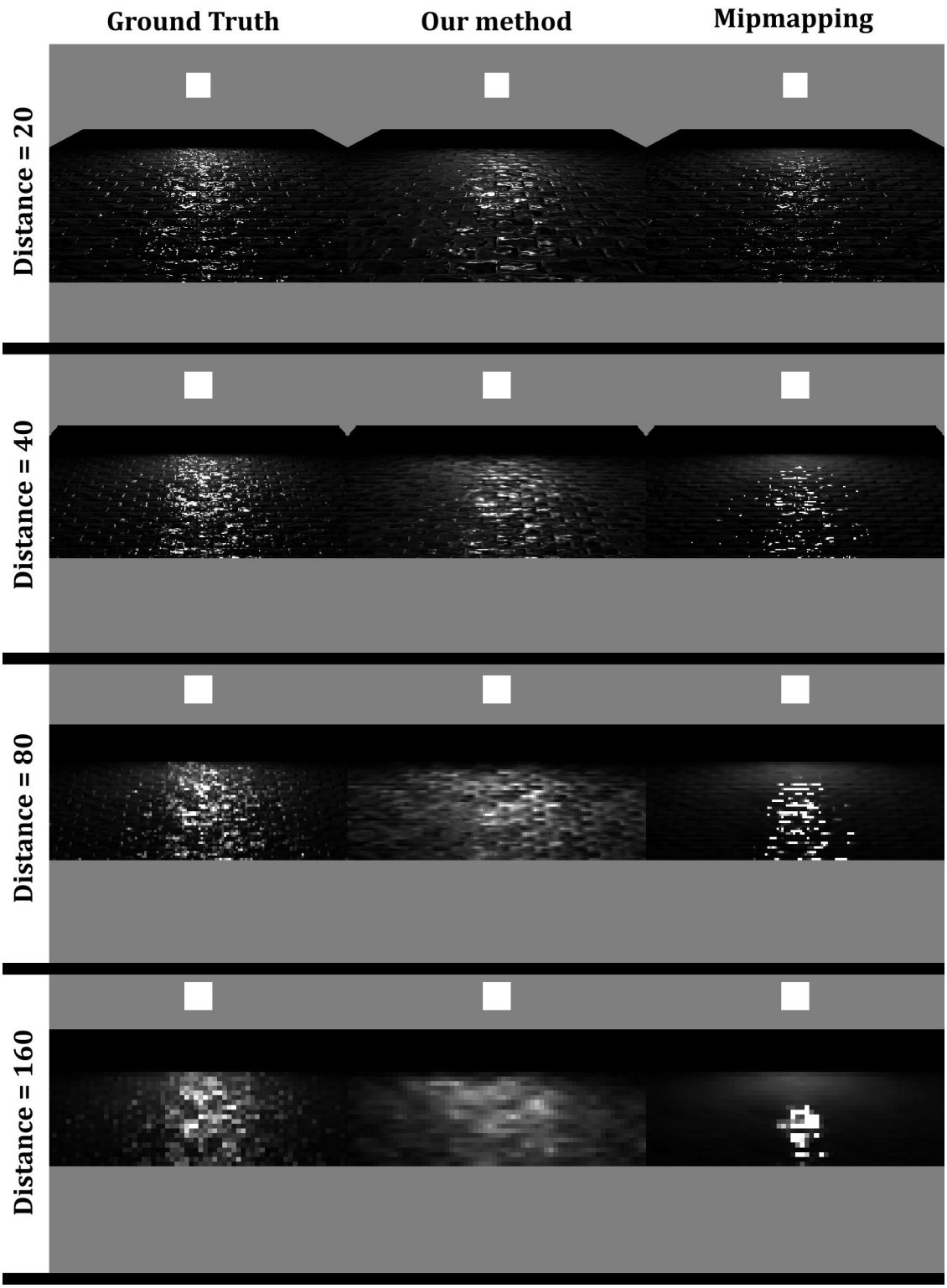


Figure 4.7: Visual comparison for several camera distances, for SCENE 2. We see that at this low view angle and at further distances, our technique yields considerable blurring whereas the naive technique averages the normals to a practically flat surface.

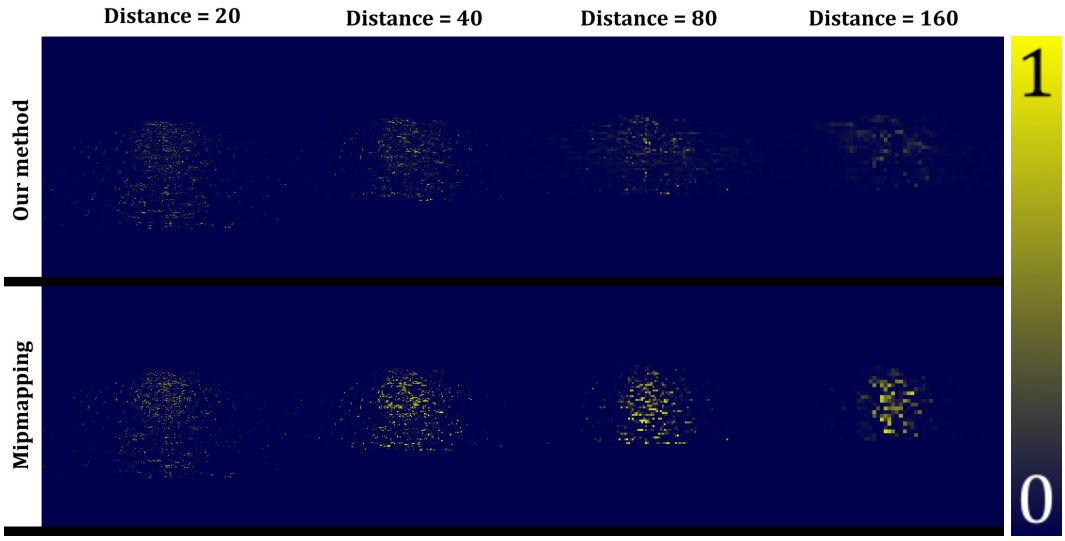


Figure 4.8: Squared error at several camera distances, for SCENE 2.

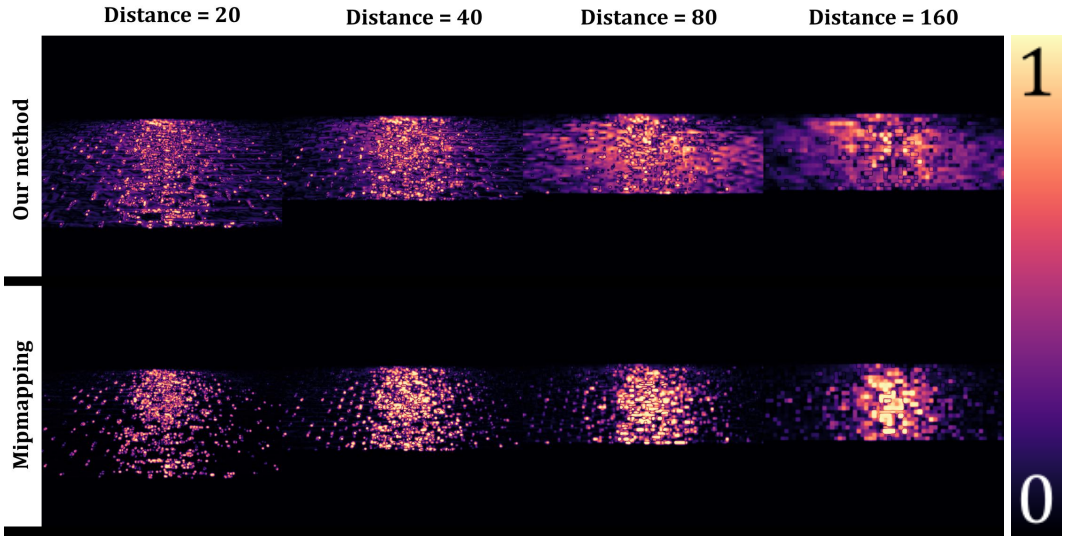


Figure 4.9: FLIP error at several camera distances, for SCENE 2.

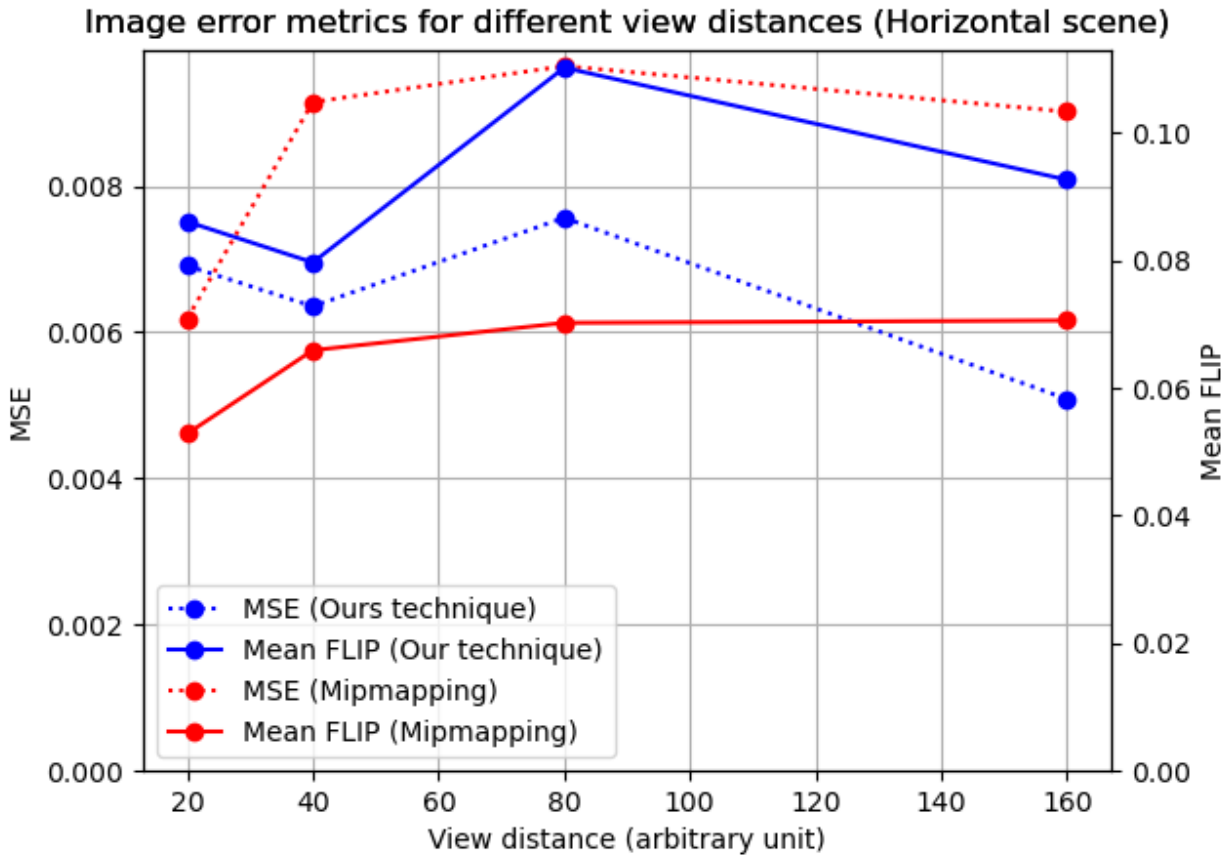


Figure 4.10: Pooled error metrics between two methods and the ground truth as a function of view distance, for SCENE 2. At small distances, again, the error metrics are similar for both techniques. Our technique performs better with the MSE metric at all scales except the closest, and the mipmapping technique performs better at all scales with the FLIP metric. Thus, overblurring affects the FLIP metric a lot more than the smoothing of the surface.

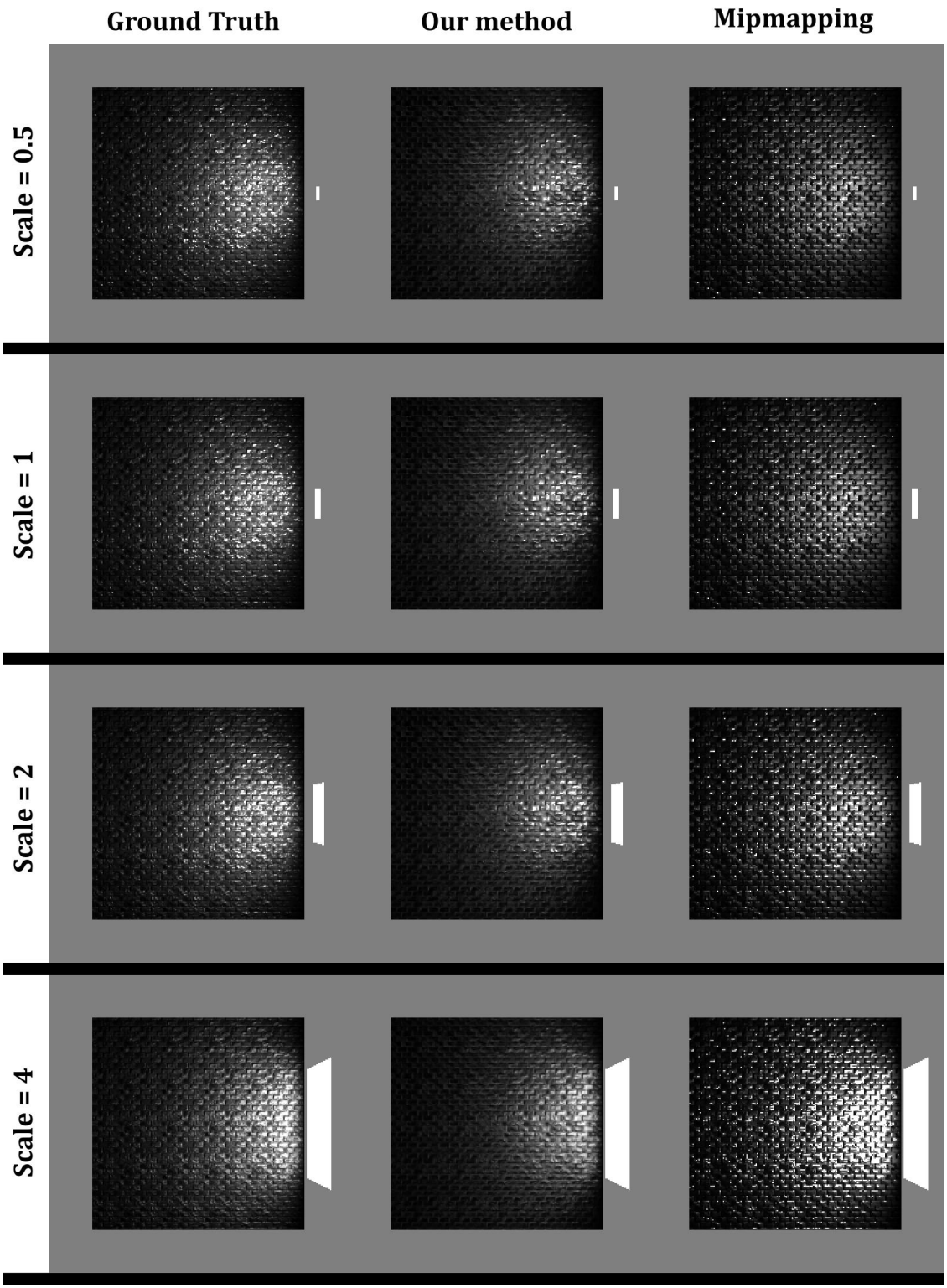


Figure 4.11: Visual comparison for several polygonal light sizes, for SCENE 1, at a fixed camera distance of 25. We see that our technique struggles to display the glints occuring further away from the light source when it is small, while the baseline shows fireflies at all light scales.

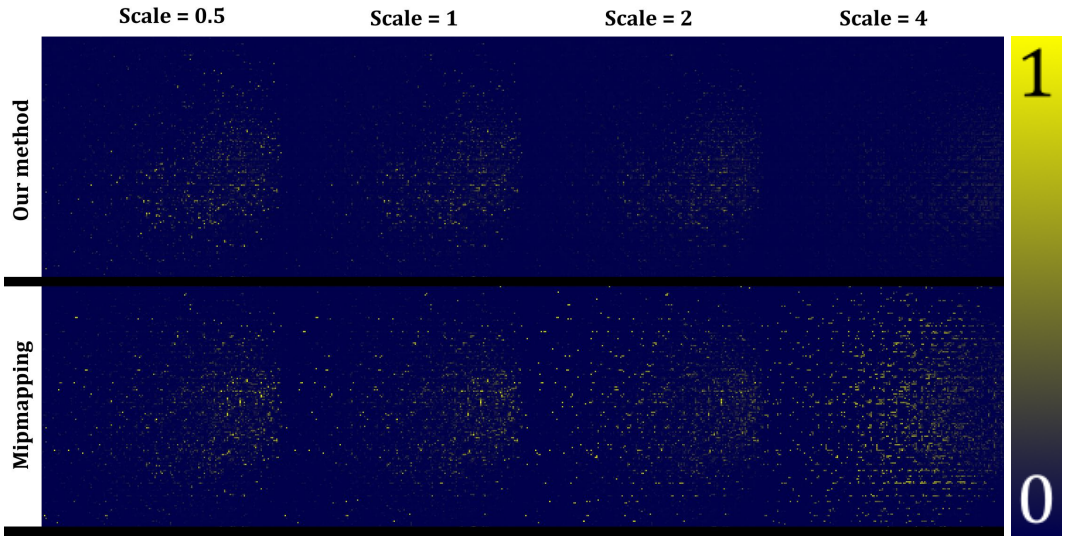


Figure 4.12: Squared error between our method and the ground truth, and between the naive method and the ground truth at several light scales, for SCENE 1.

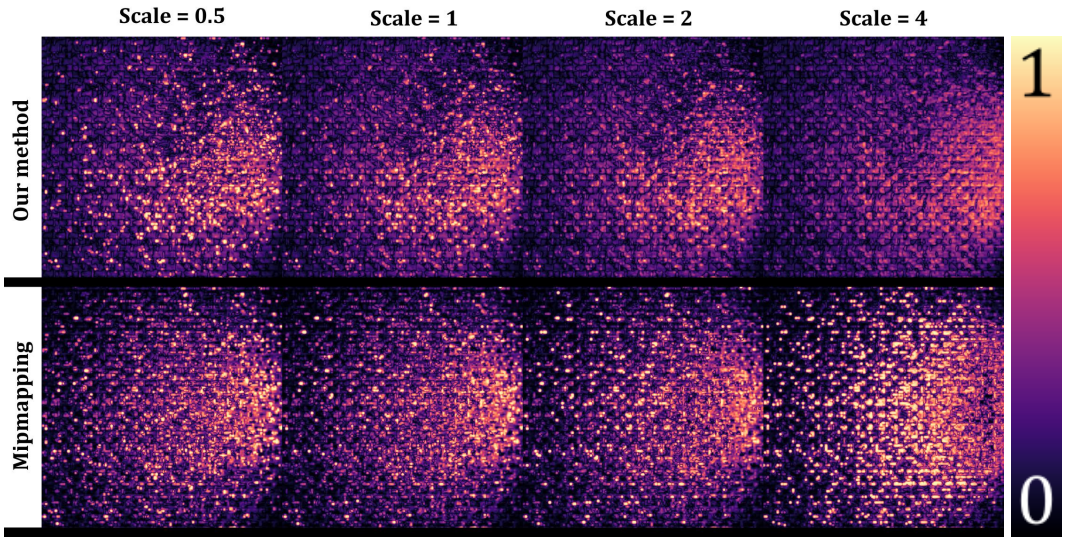


Figure 4.13: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several light scales, for SCENE 1.

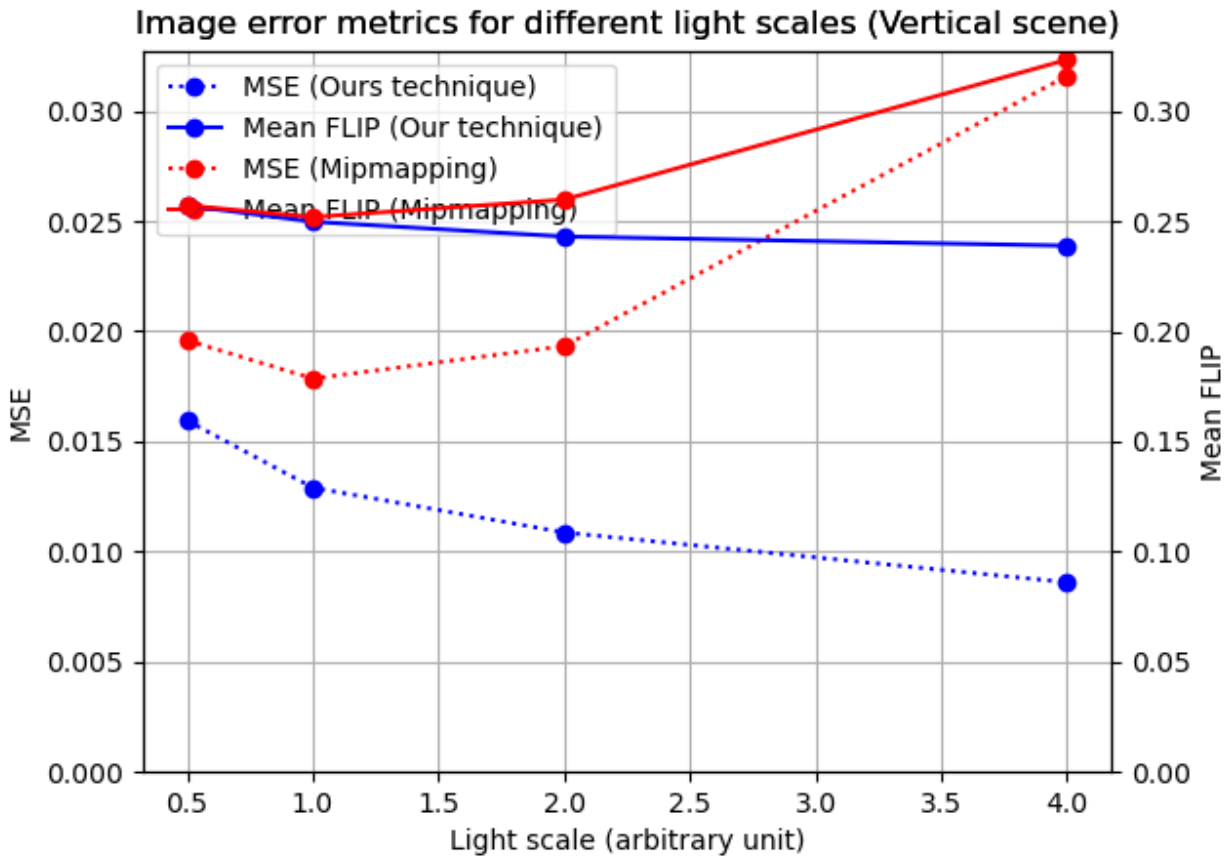


Figure 4.14: Pooled error metrics between our method and the ground truth, and between the baseline and the ground truth, as a function of light scale for SCENE 1. We can see that a larger polygonal light means lower frequency illumination, which is easier to filter. As the source gets larger, our results resemble the ground truth better. It is the opposite with the naive baseline, as its error increases with both metrics as the light gets larger.

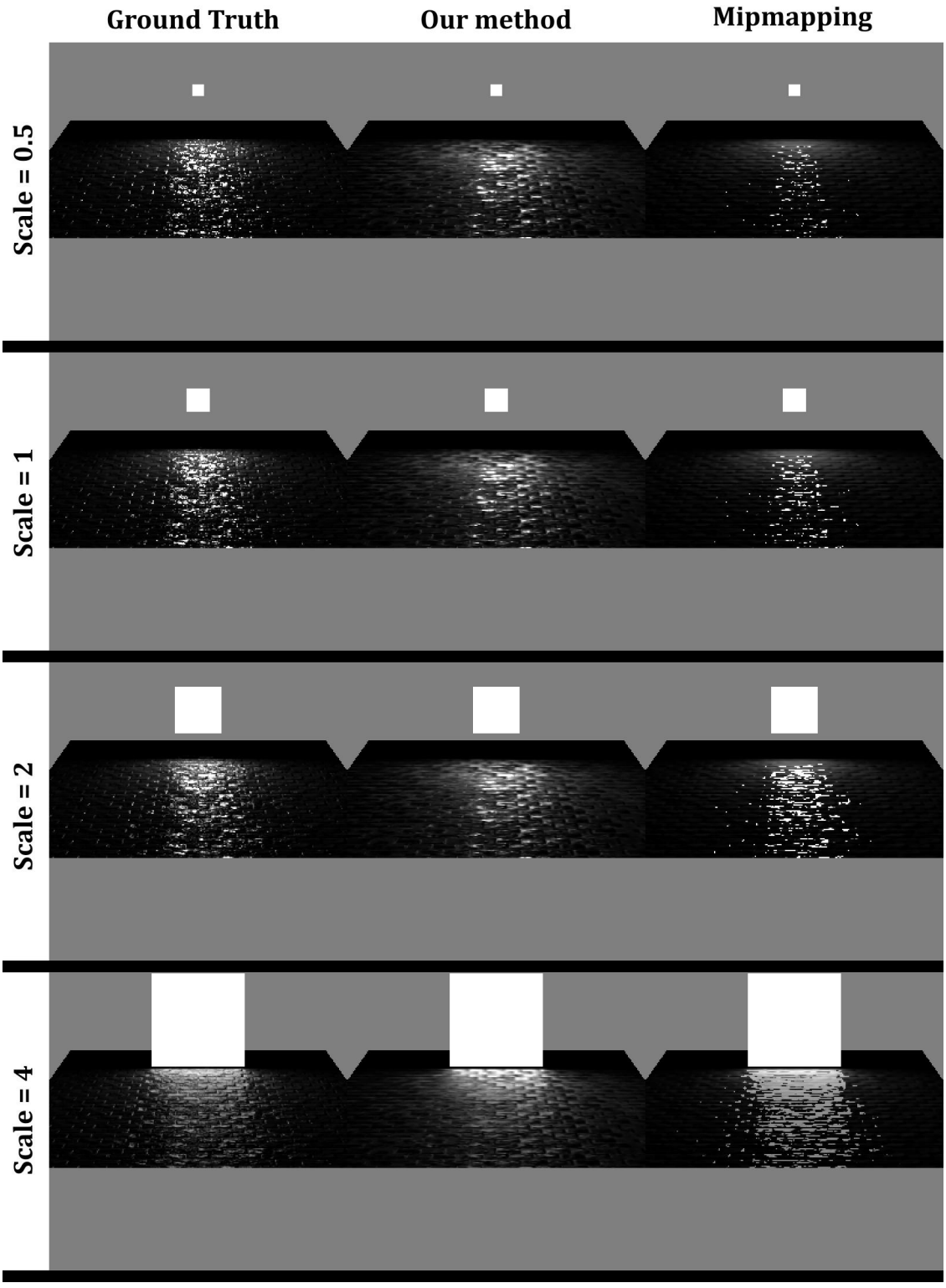


Figure 4.15: Visual comparison for several light scales, for SCENE 2. When the light source is small, SCENE 2 makes it more obvious that our technique displays some blurring of the highlights instread of the high frequency glints. In general however, our renders are free of aliasing at all light scales.

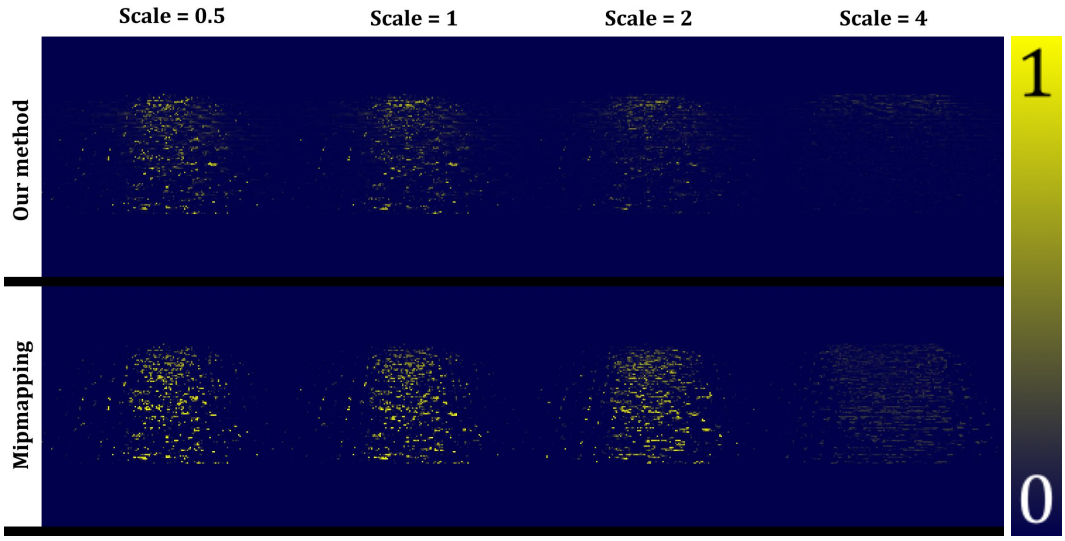


Figure 4.16: Squared error at several light scales, for SCENE 2.

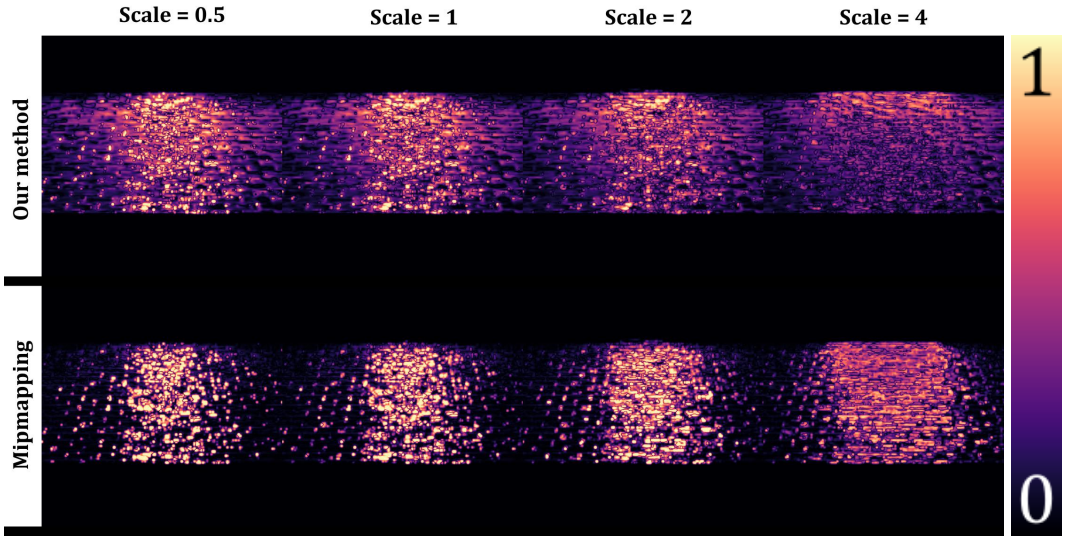


Figure 4.17: FLIP at several light scales, for SCENE 2.

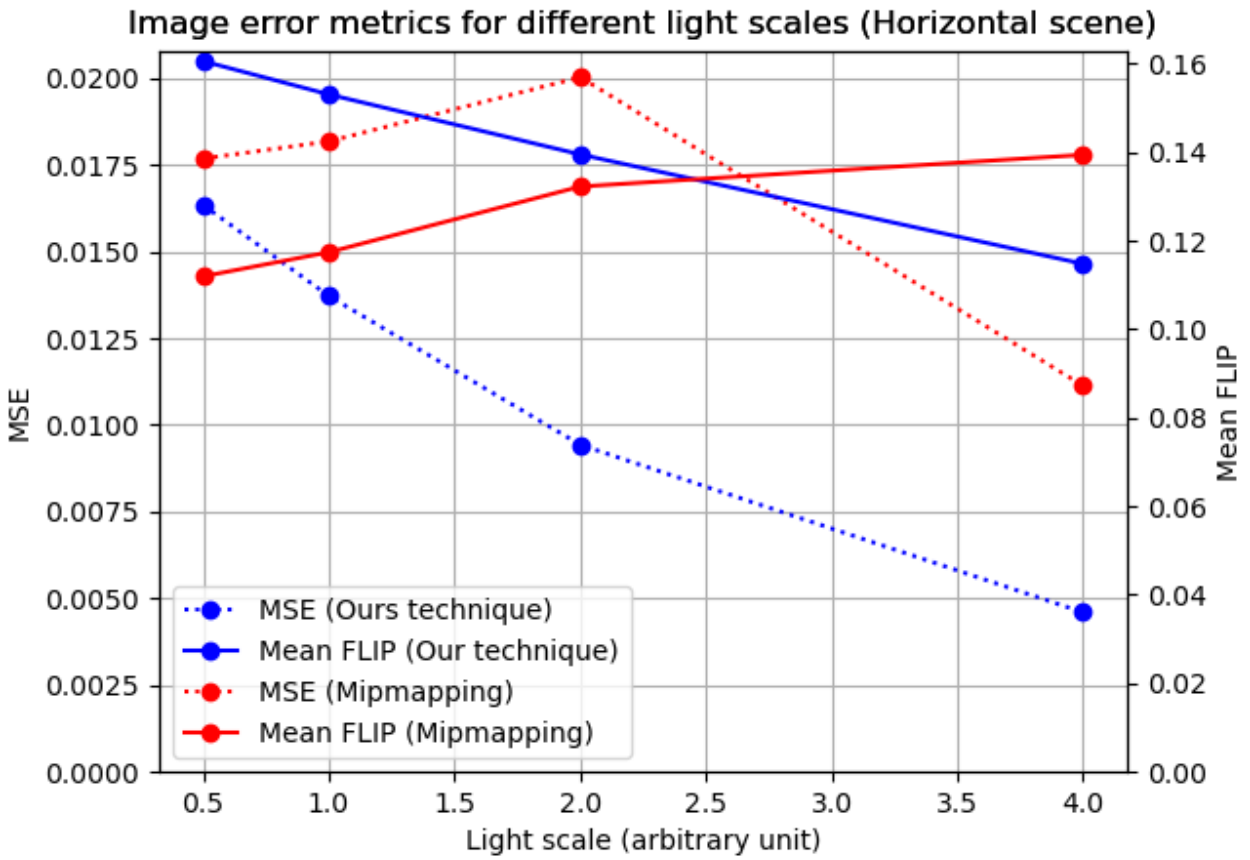


Figure 4.18: Pooled error metrics between the two methods and the ground truth as a function of light scale, for SCENE 2. Our MSE error metric is consistently better than the baseline, but out FLIP metric beats the baseline for larger light sources only. We see that it is mainly due to our technique's diffuse component which is wrong for smaller quad lights.

4.3 GLITTER

We proceed with the same studies for the GLITTER normal map. This texture was obtained from https://help.autodesk.com/view/ARENDERING/ENU/?guid=GUID-97CC0DD3-35A8-4D8F-80A0-B1C4AD54D2B0.

This normal map shows less structure and more noisy detail than the STONE normal map. Moreover, each glitter flake is the same size so it is harder to distinguish glints from aliased fireflies. Thus, we expect the naive mipmapping to perform better.

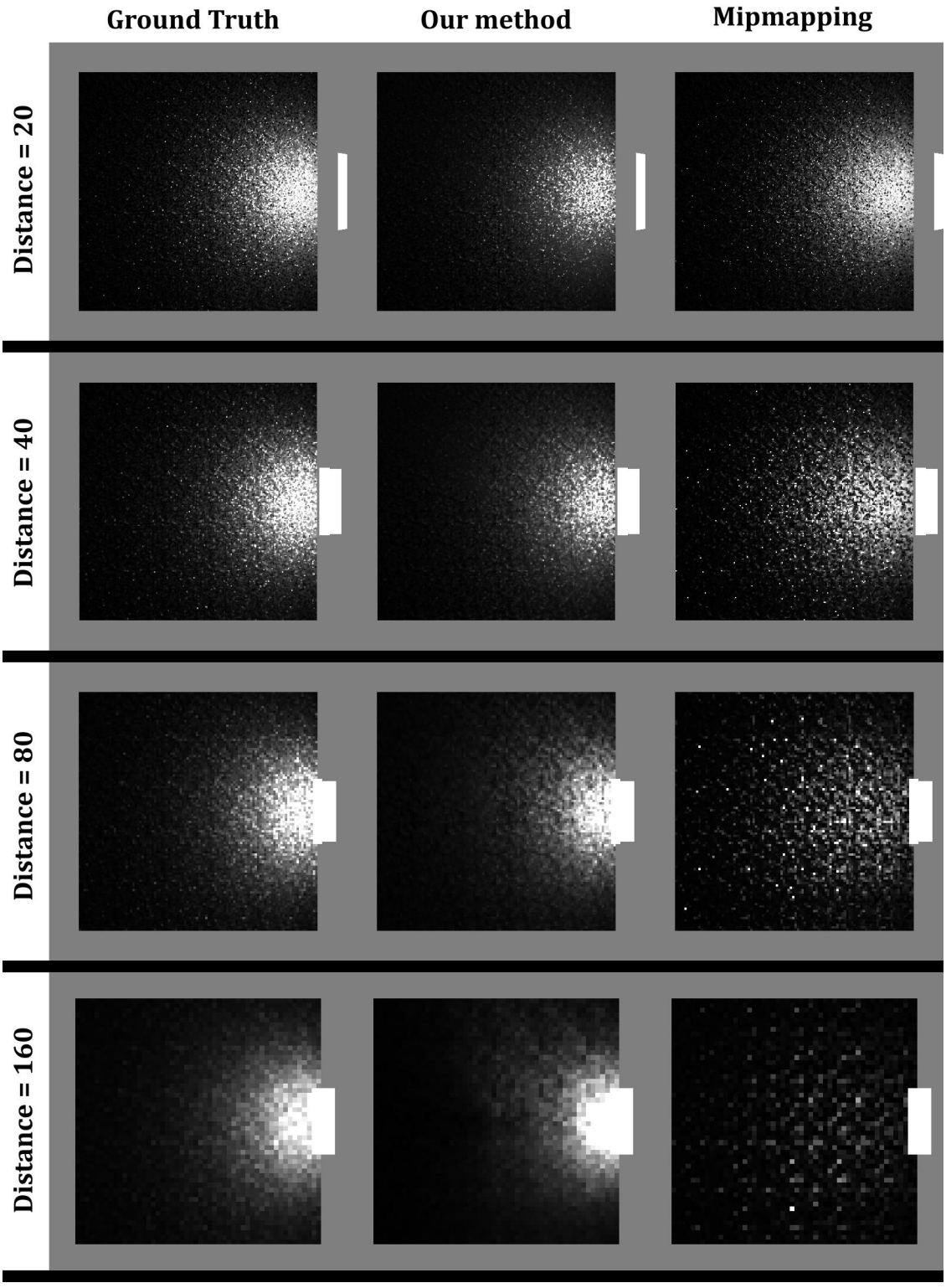


Figure 4.19: Visual comparison for several camera distances for SCENE 1. We can see that at this high amount of noisy detail, our technique is visually almost undistinguishable from the ground truth at all scales. The naive technique does not perform well for further distances and almost always misses the polygonal light at a distance of 160.

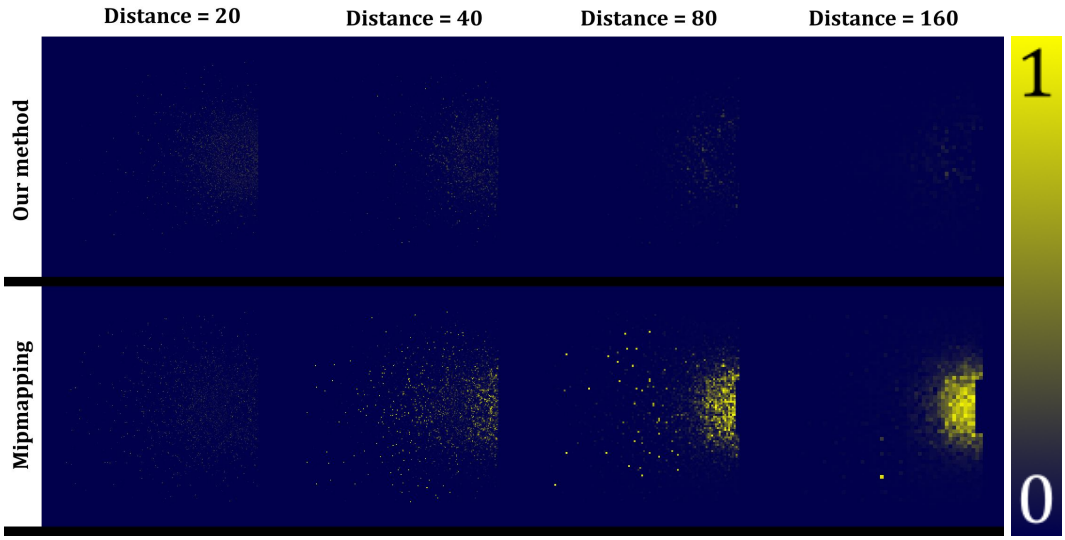


Figure 4.20: Squared error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1.

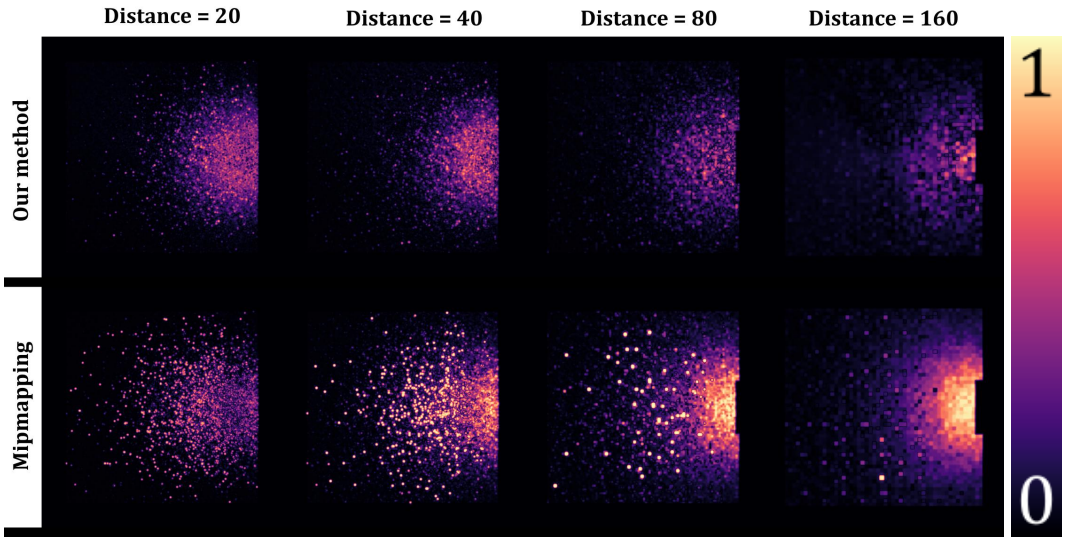


Figure 4.21: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1.

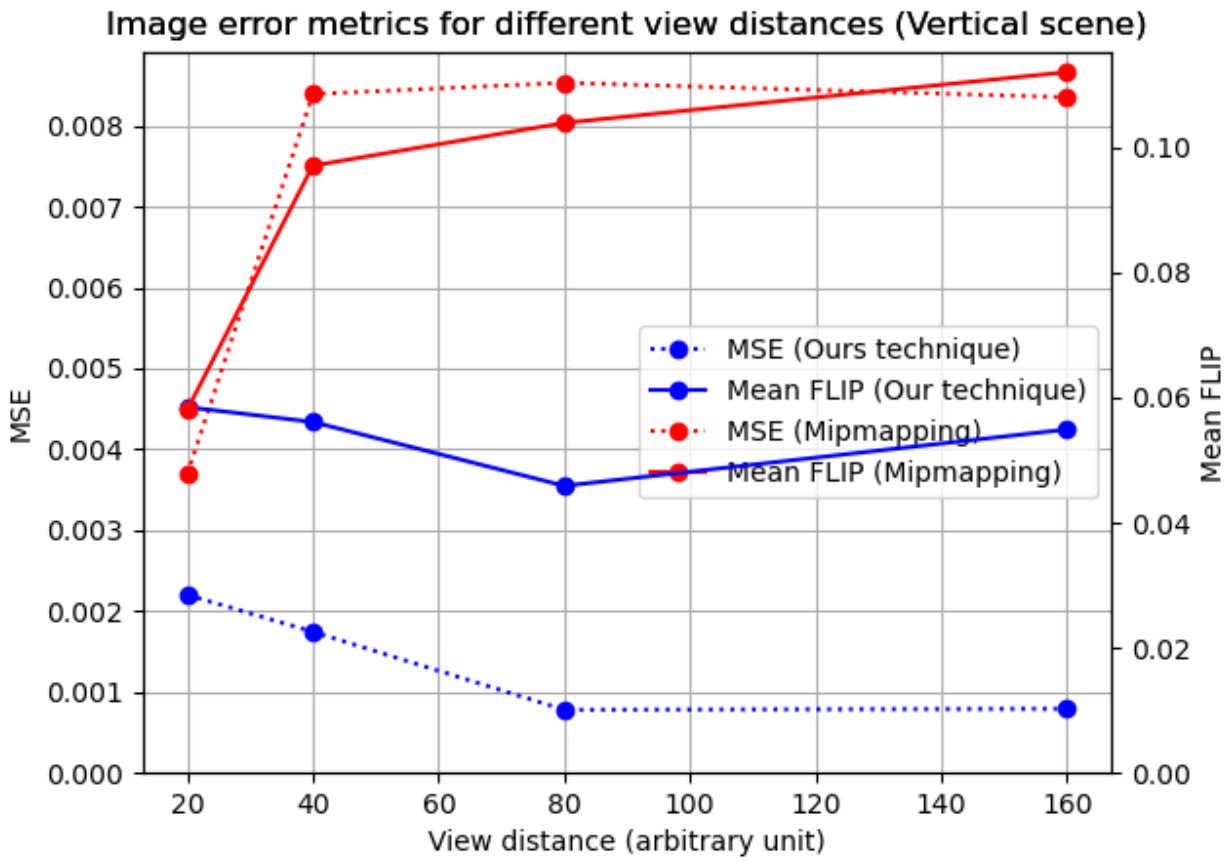


Figure 4.22: Error metrics pooled over the whole images, as a function of view distance, for SCENE 1. Our method's error metrics stays relatively constant with distance, while the baseline's errors increases with distance.

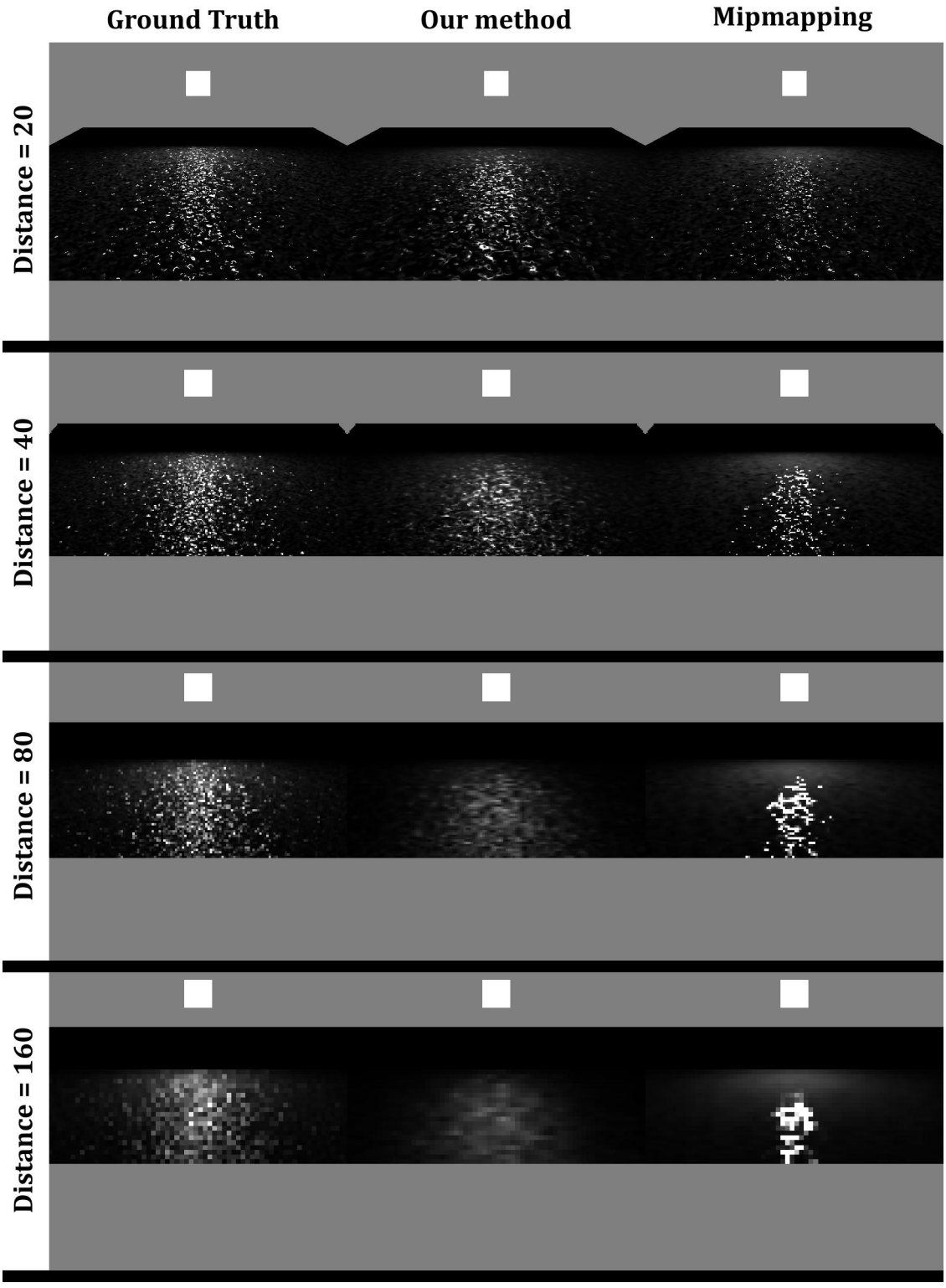


Figure 4.23: Visual comparison for several camera distances for SCENE 2. Unlike in SCENE 1, the visual quality of our method breaks down at further distances, and shows considerable overblurring. In contrast, the ground truth maintains high frequency glints that are difficult for our technique to filter. The fitted LTC lobes are too broad, which also cause energy loss.

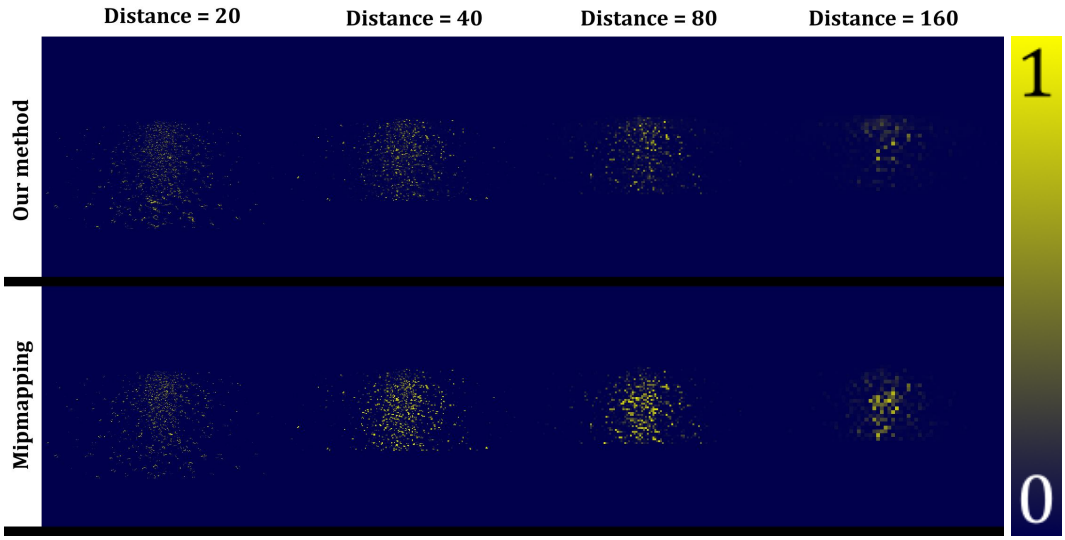


Figure 4.24: Squared error at several camera distances, for SCENE 2. The error with our method is high because of overblurring, and the error with the baseline is high because of the averaging of normals, which produces a smoother surface than desired.

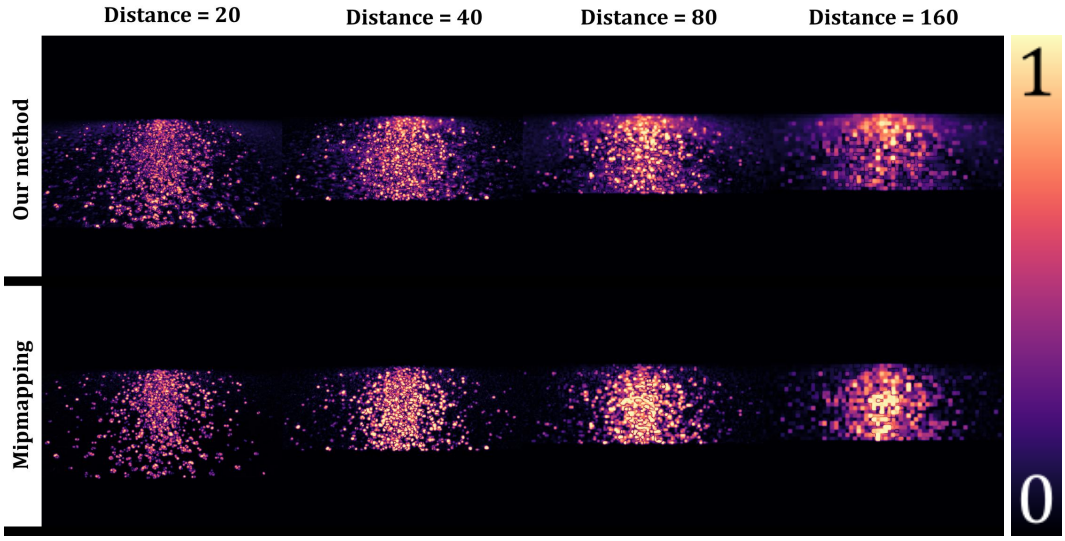


Figure 4.25: FLIP error at several camera distances, for SCENE 2.

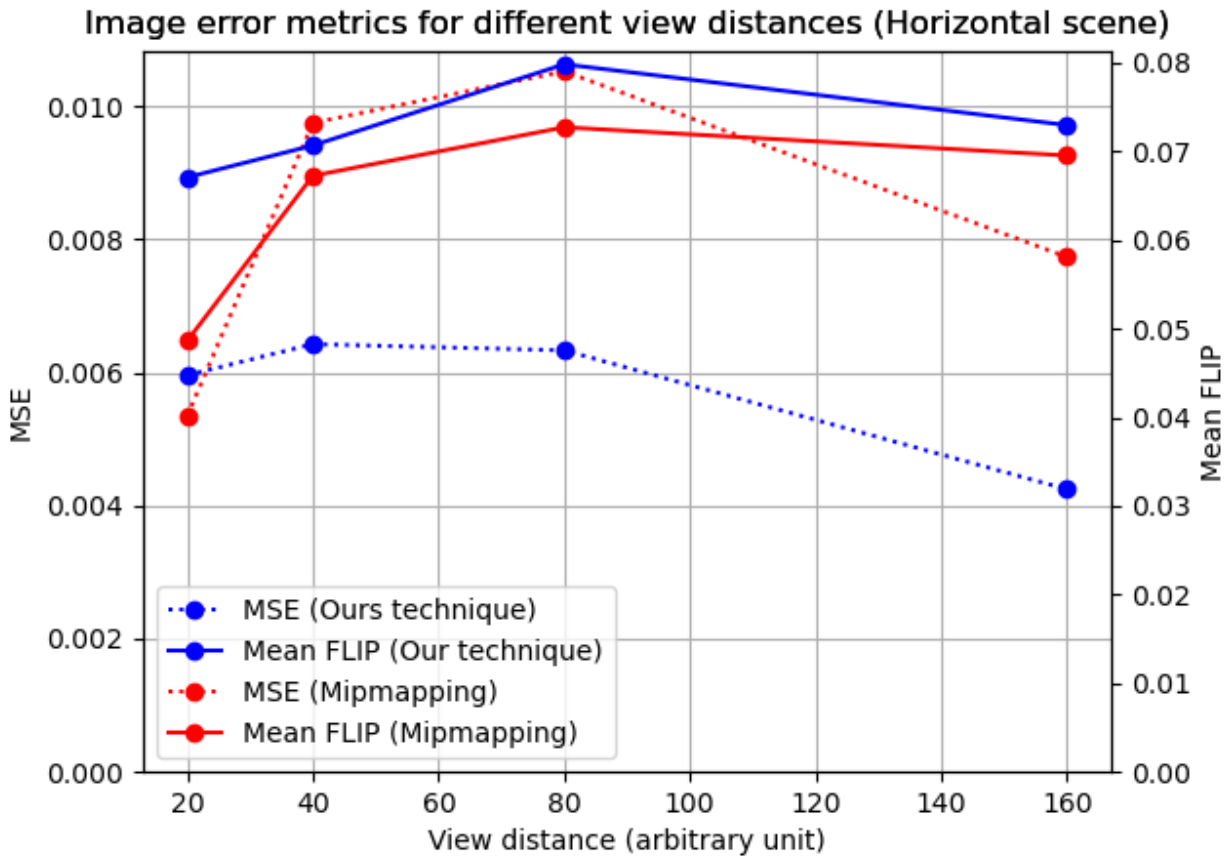


Figure 4.26: Pooled error metrics between two methods and the ground truth as a function of view distance, for SCENE 2. The mean FLIP is similar for both techniques, but the visual result is poor for two different reasons. For our technique, it is overblurring, and for the naive baseline, it is an overly specular reflection of the area light.

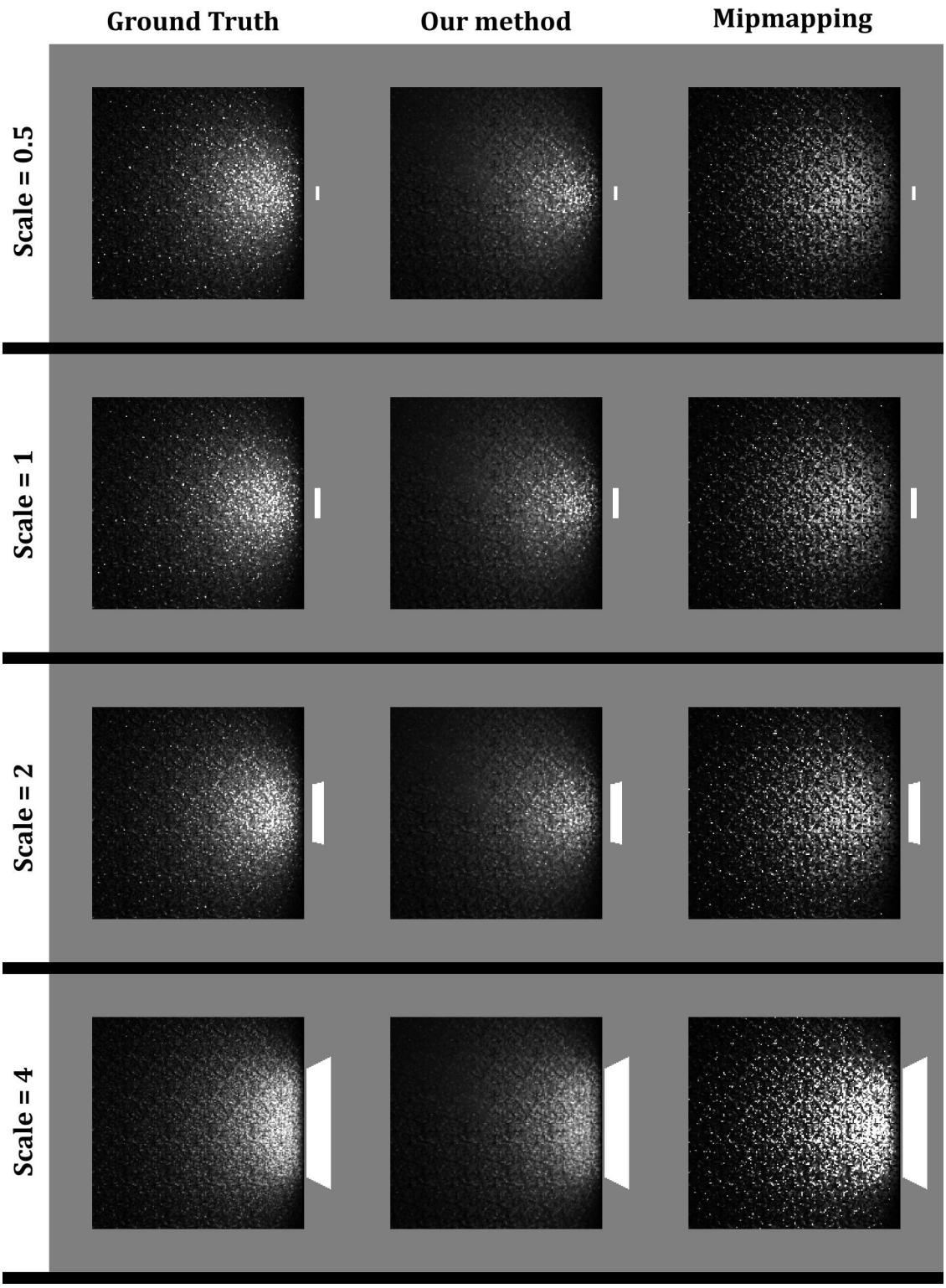


Figure 4.27: Visual comparison for several light scales, for SCENE 1. Our technique yields pleasing and convincing results, but it fails to render some small and distant glints when the area light is smaller than 4. We also see a minor energy loss at all scales. With the baseline, the noisier result is accompanied by a large excess of energy.

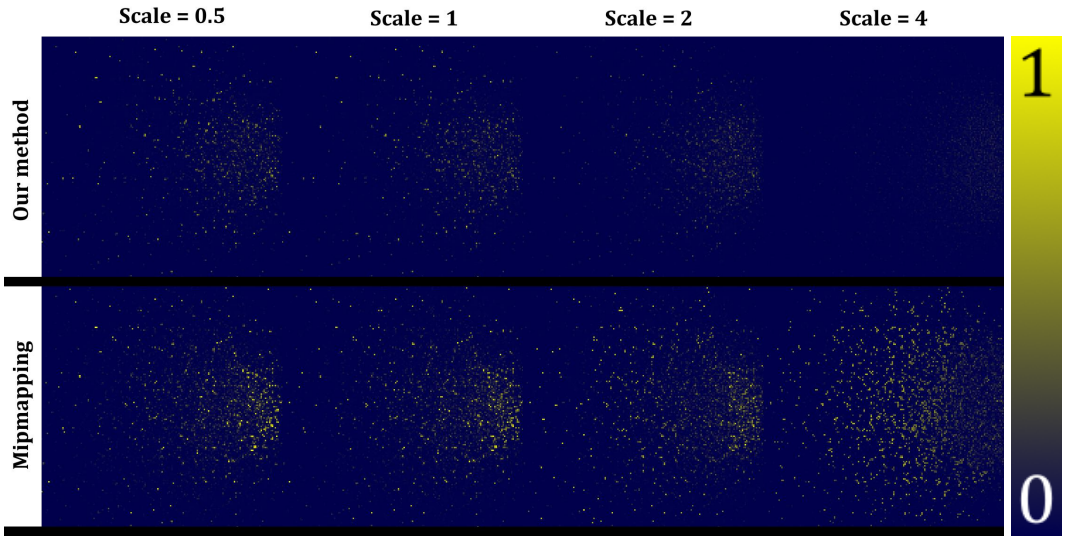


Figure 4.28: Squared error between our method and the ground truth, and between the naive method and the ground truth at several light scales, for SCENE 1.

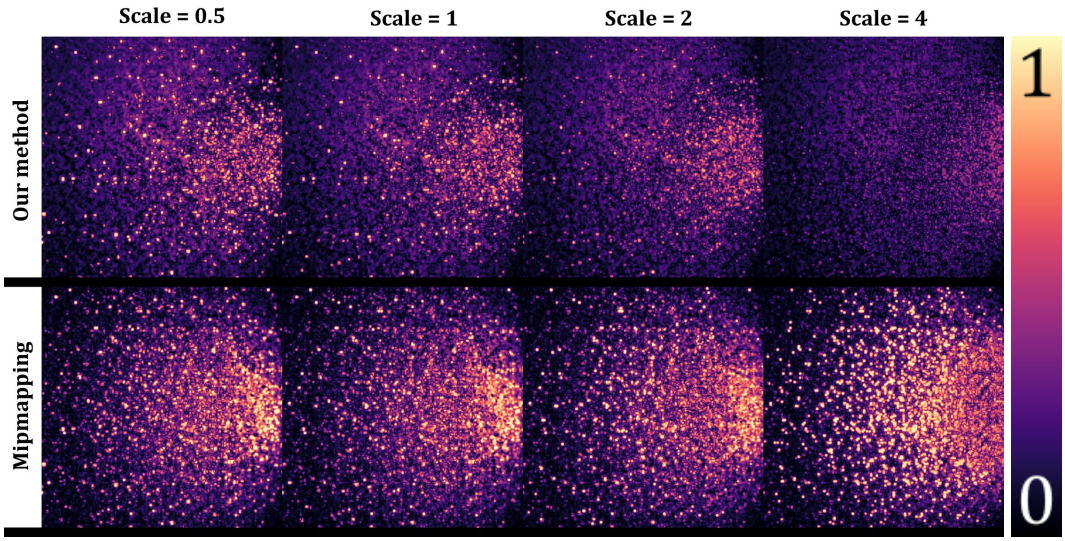


Figure 4.29: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several light scales, for SCENE 1.

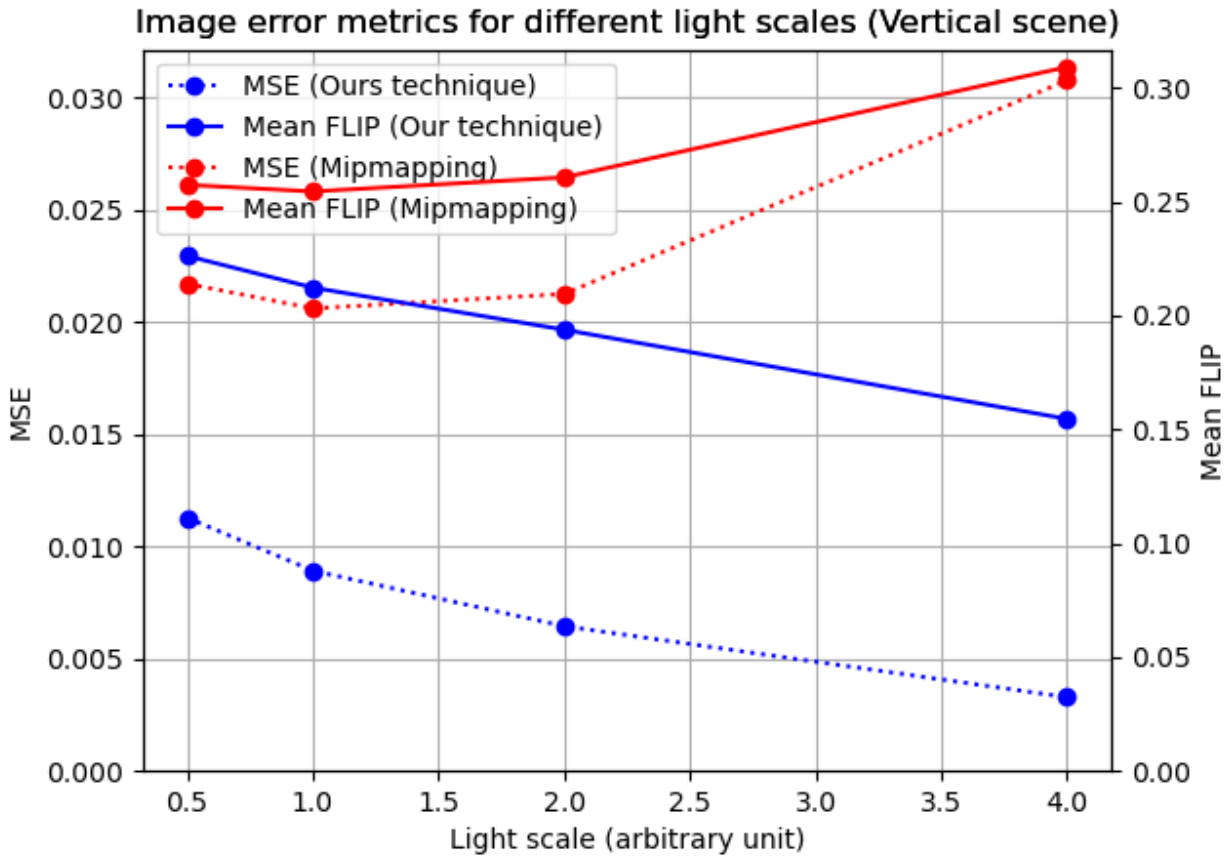


Figure 4.30: Pooled error metrics between our method and the ground truth, and between the baseline and the ground truth, as a function of light scale for SCENE 1. As observed in other SCENE 1 comparisons, our results are consistently better than the baseline.

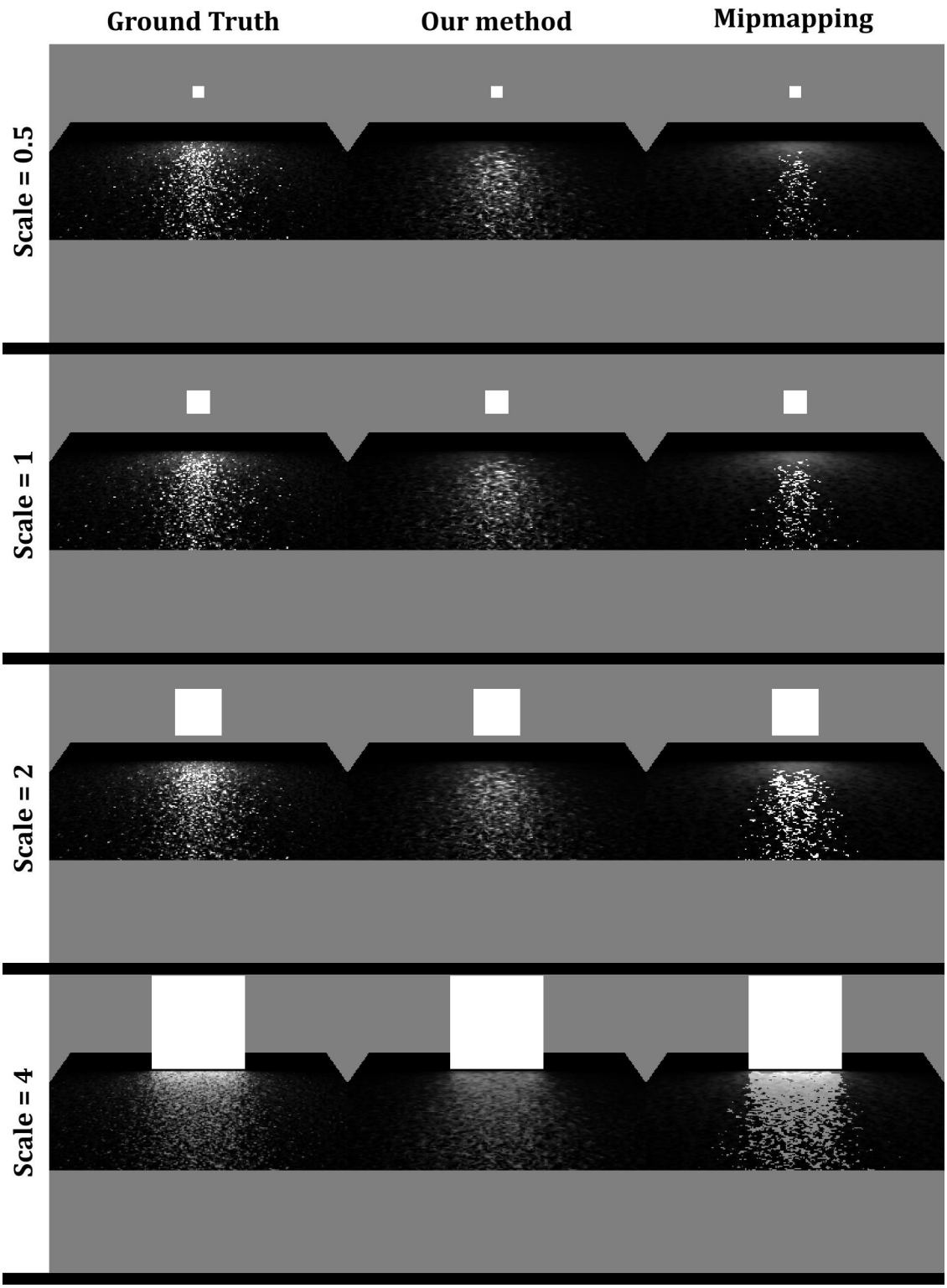


Figure 4.31: Visual comparison for several light scales, for SCENE 2. Our technique's renders contains is some minor blurring and energy loss, but visually the results are more convincing than with naive mipmapping.

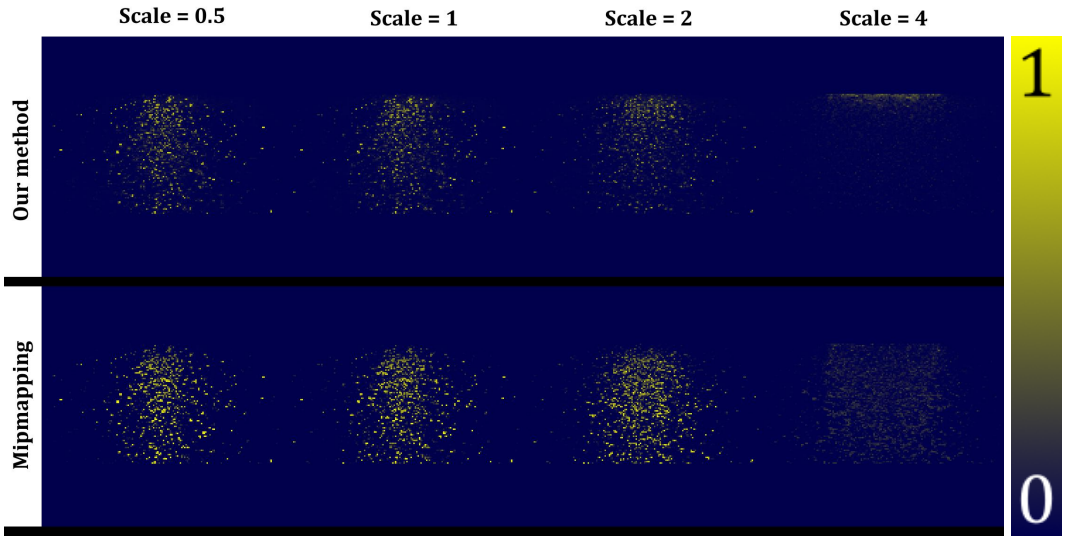


Figure 4.32: Squared error at several light scales, for SCENE 2.

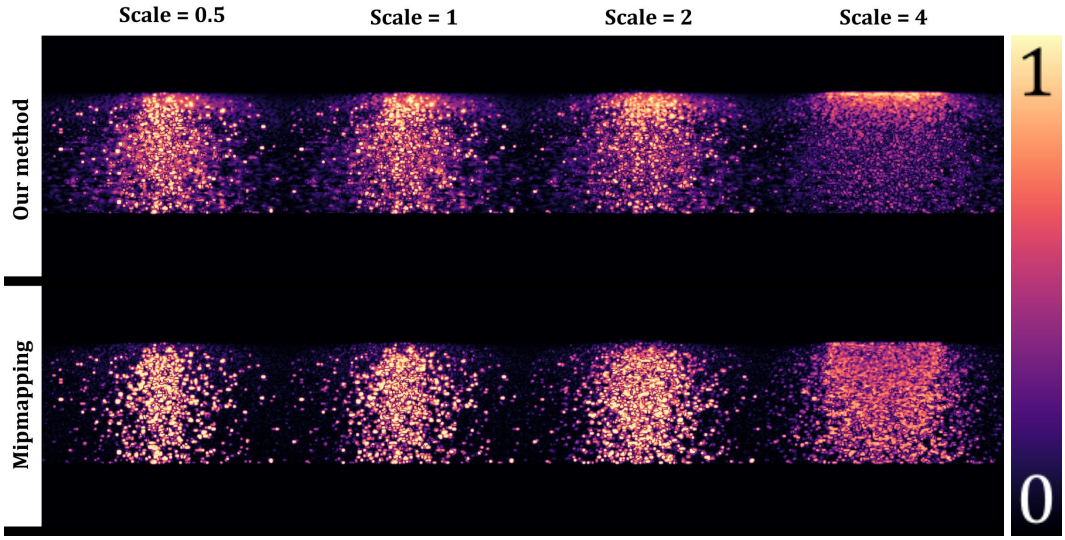


Figure 4.33: FLIP at several light scales, for SCENE 2.

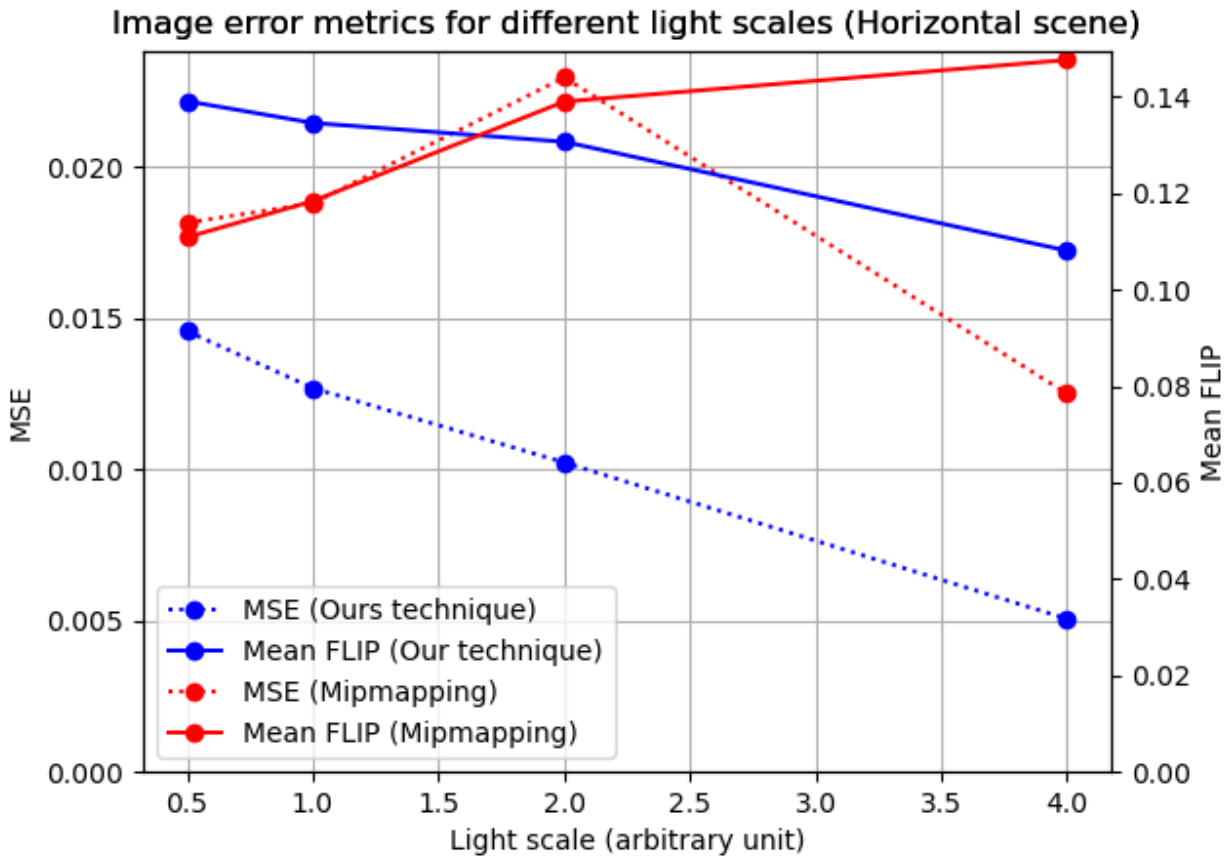


Figure 4.34: Pooled error metrics between the two methods and the ground truth as a function of light scale, for SCENE 2. For both SCENE 1 and SCENE 2, our technique's MSE and mean FLIP decreases as the light source gets larger. With the baseline method, it is more unpredictable. Our FLIP metric only beats the baseline when the light source is not too small.

4.4 SCRATCH

We finally run the same studies on the SCRATCH normal map, taken from https://www.turbosquid.com/FullPreview/Index.cfm/ID/1061804. For this normal map, the detail exclusively occurs at very high spatial frequency, thus we use another strategy to render the diffuse component.

Since there are no large spatial patterns in this map and only small scratch details, most of the normals on the texture are vertical. In this case, there is almost no aliasing of the diffuse component and our filtered diffuse component is of no use (and it does not correctly fit the lambertian reflectance of the flat surface). Therefore, we choose to use our filtered specular component, and to compute the diffuse component on the flat surface with the standard isotropic LTC technique (as if the surface had no normal map).

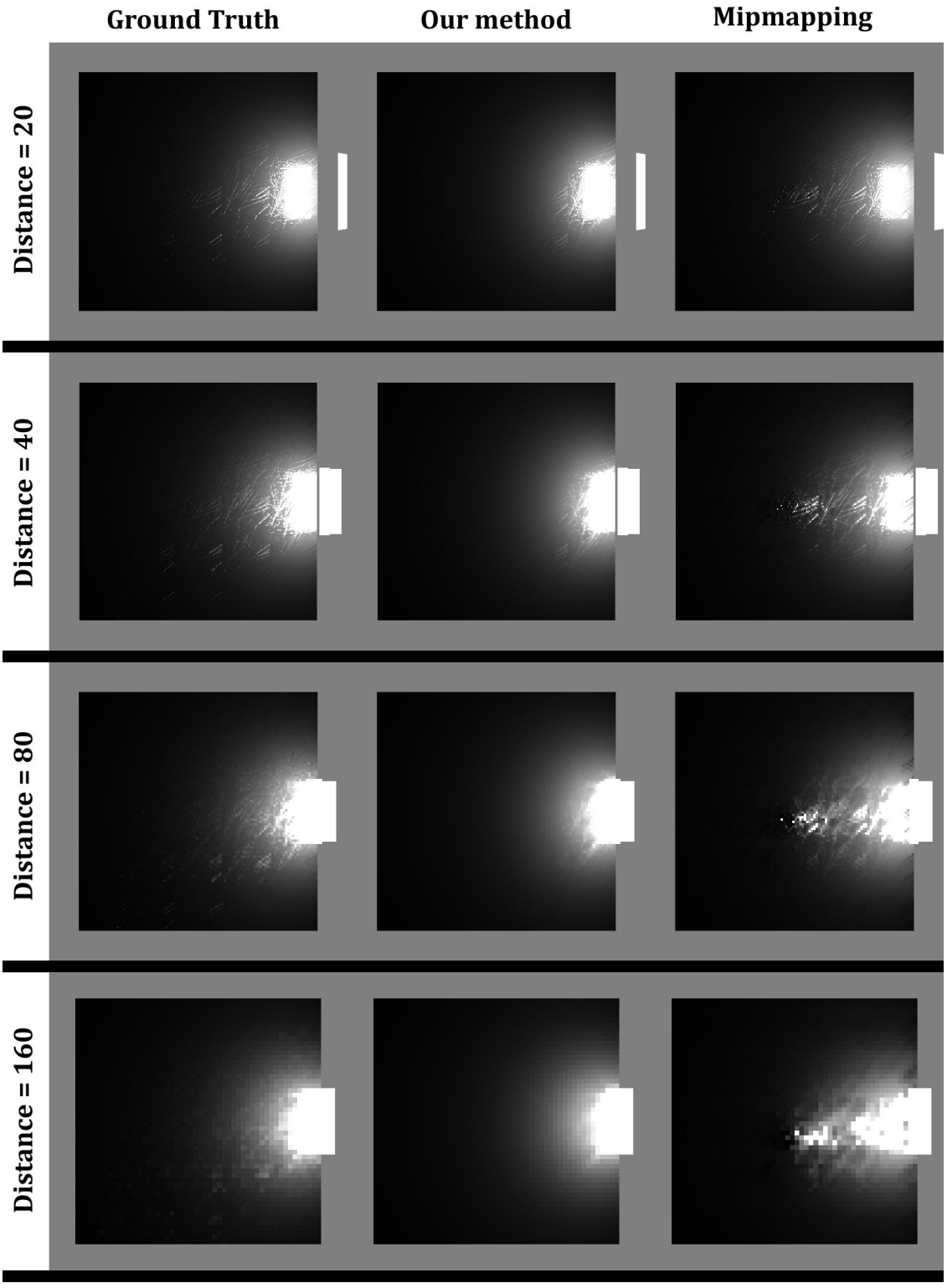


Figure 4.35: Visual comparison for several camera distances for SCENE 1. We see that some fine glints displayed by the ground truth are missing from our method, and are especially hard to represent at large distances. The naive baseline shows important aliasing as the normal directions around the scratches are averaged.

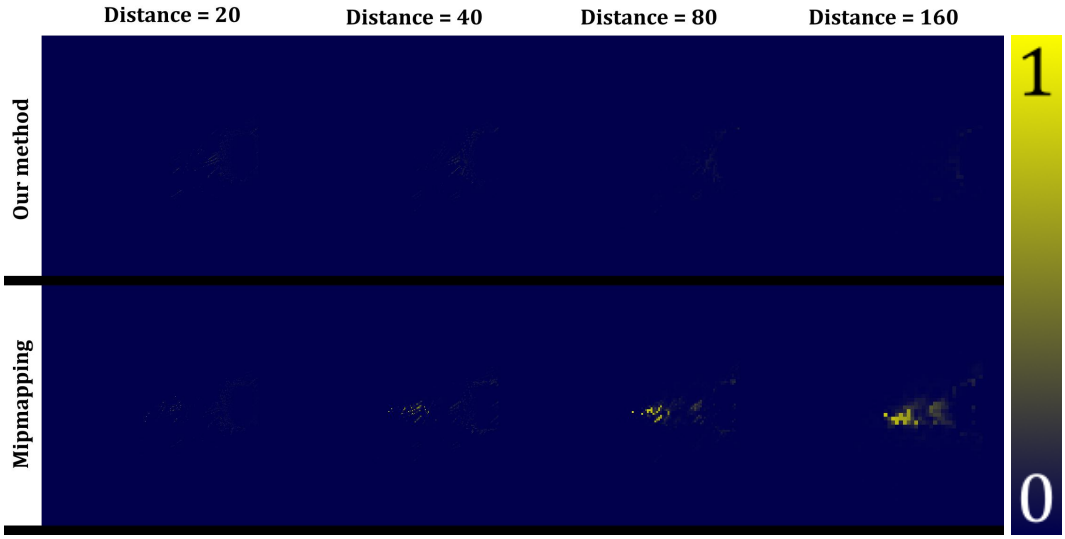


Figure 4.36: Squared error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1. The only difference between our technique and the ground truth are very fine glints.

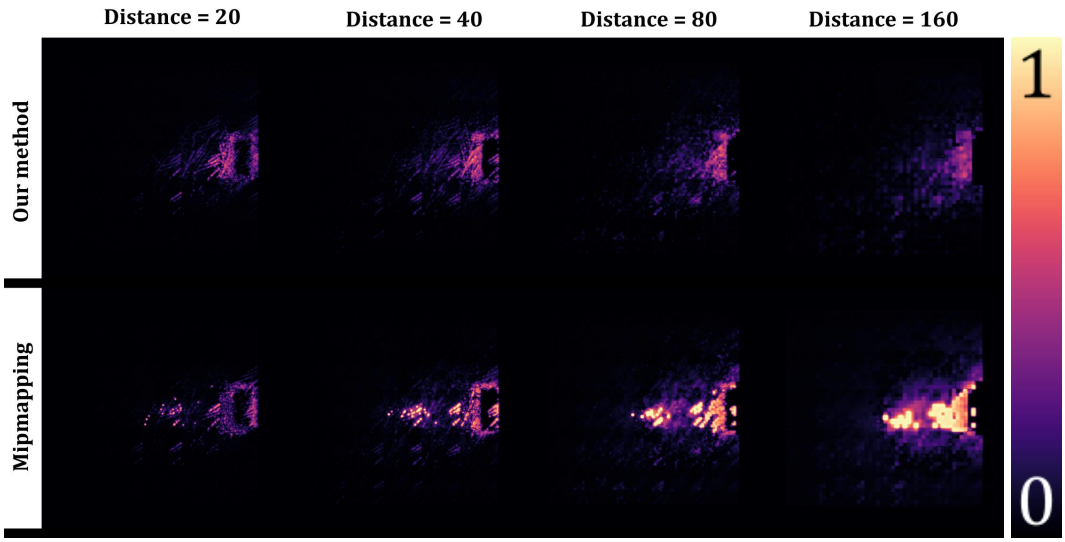


Figure 4.37: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1.

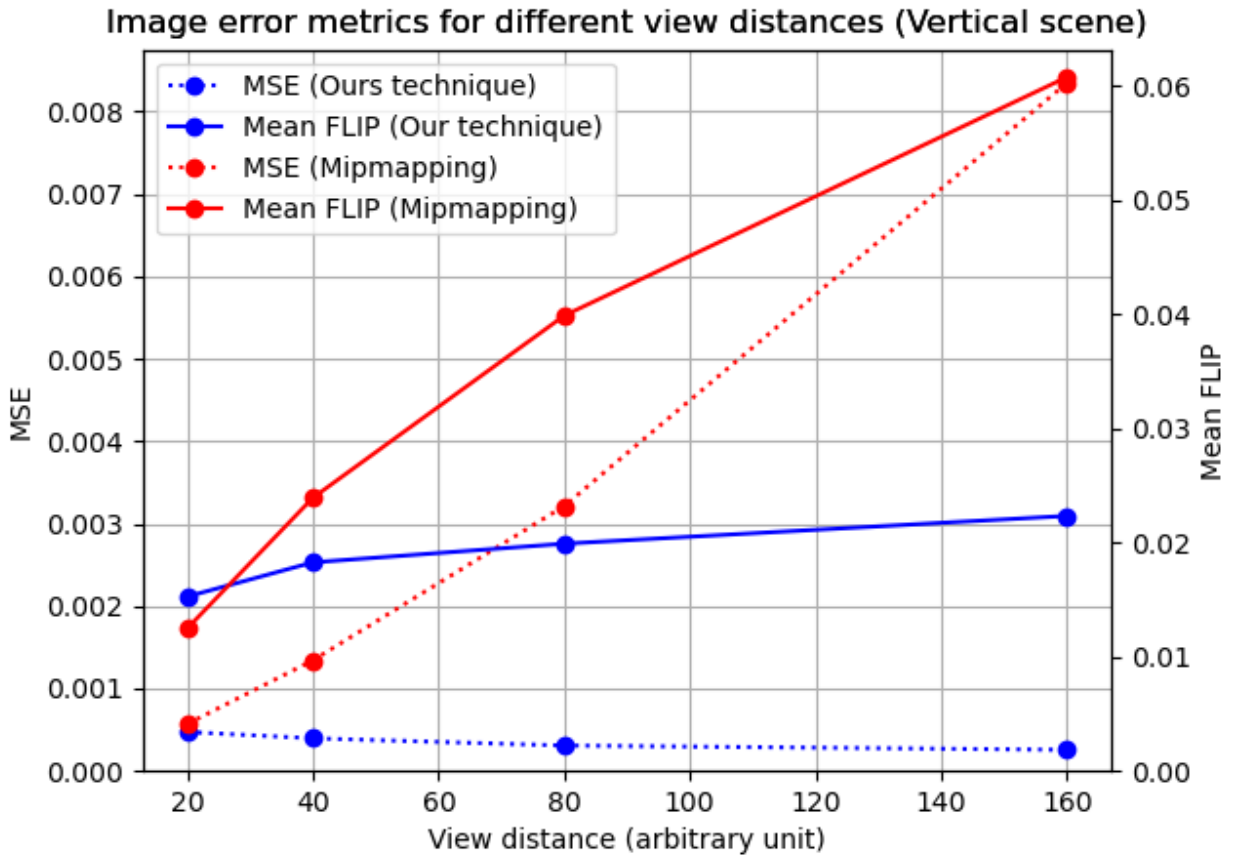


Figure 4.38: Error metrics pooled over the whole images, as a function of view distance, for SCENE 1. We see that the baseline's aliasing is more important as the distance increases, but our technique approximately maintains the same accuracy for both image metrics.

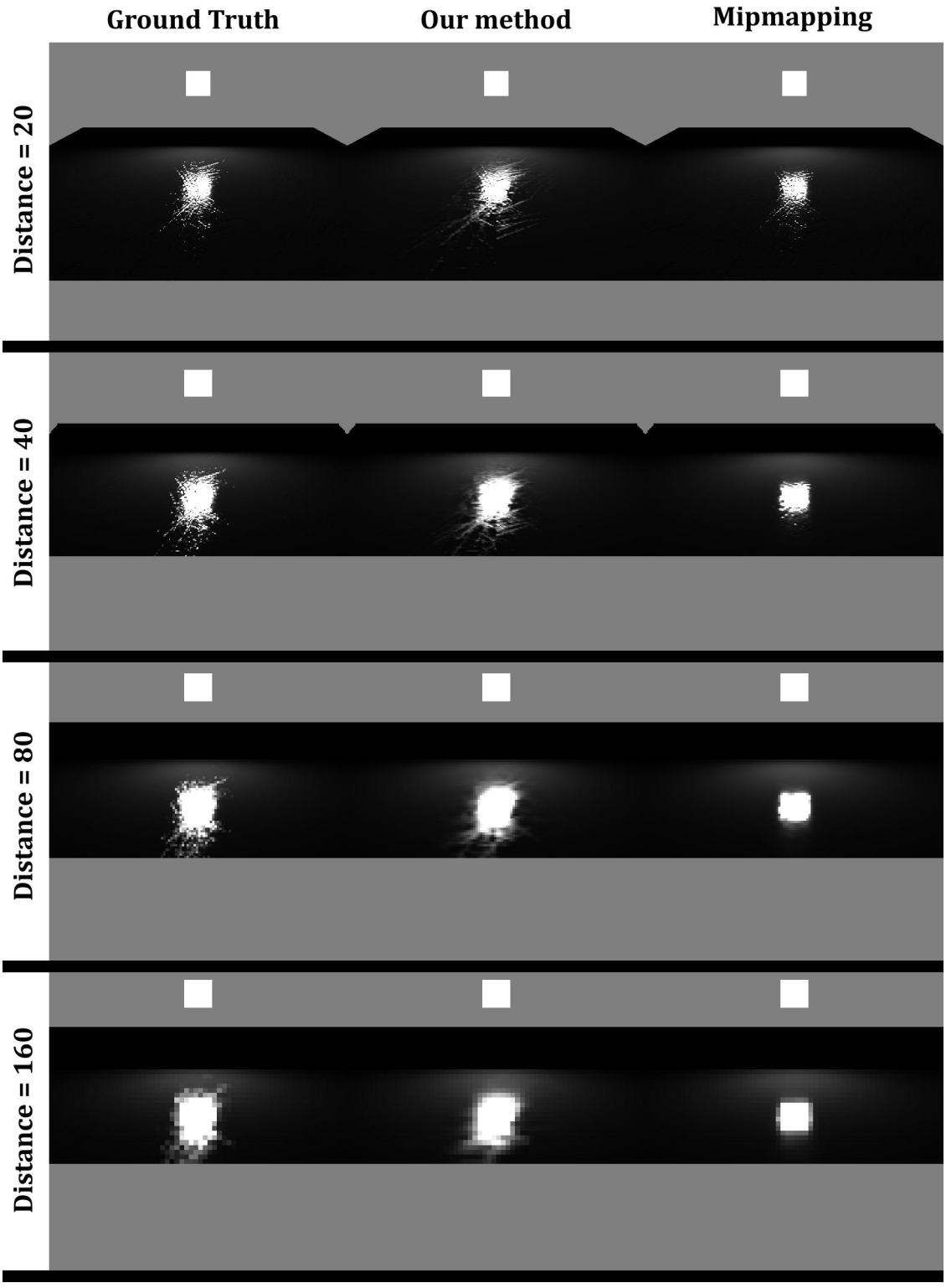


Figure 4.39: Visual comparison for several camera distances for SCENE 2. Our method shows some overblurring at distances 40 and above, and this is expected since the eBRDF lobes of such fine details are hard to filter. However, the shape of the highlights are preserved. At further distances, the naive baseline completely averages the normals to a flat surface.

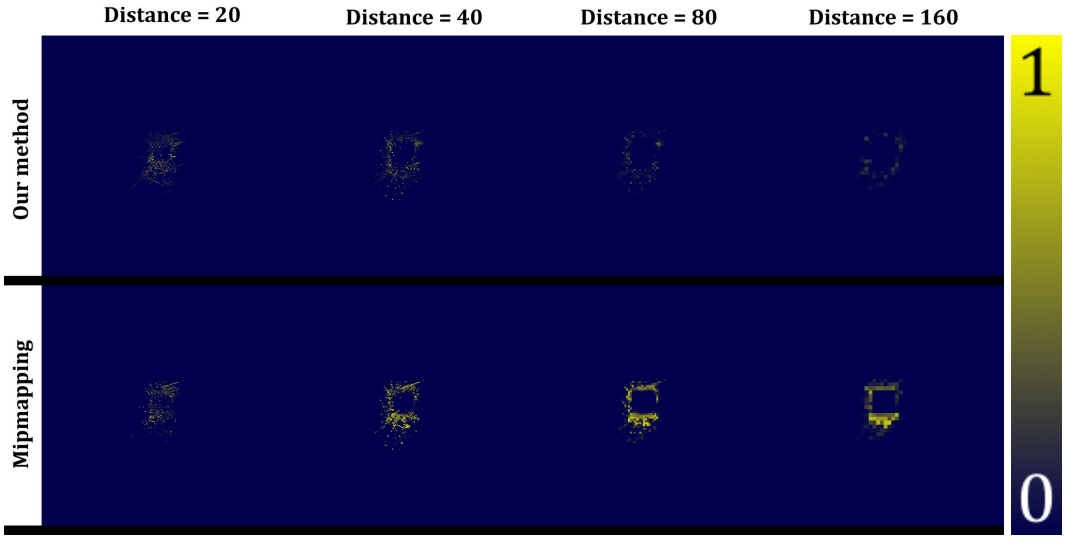


Figure 4.40: Squared error at several camera distances, for SCENE 2. The small amount of blurring by our method does not generate a high L2 error, as the highlight shape is accurate. This is not the case with the mipmapping technique, which completely misses the shape of the highlight.

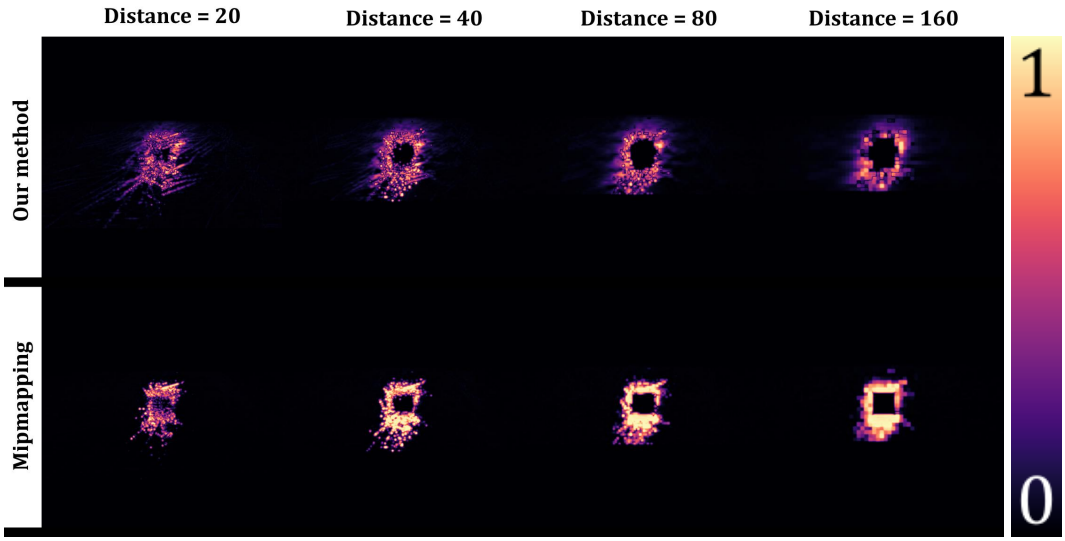


Figure 4.41: FLIP error at several camera distances, for SCENE 2.

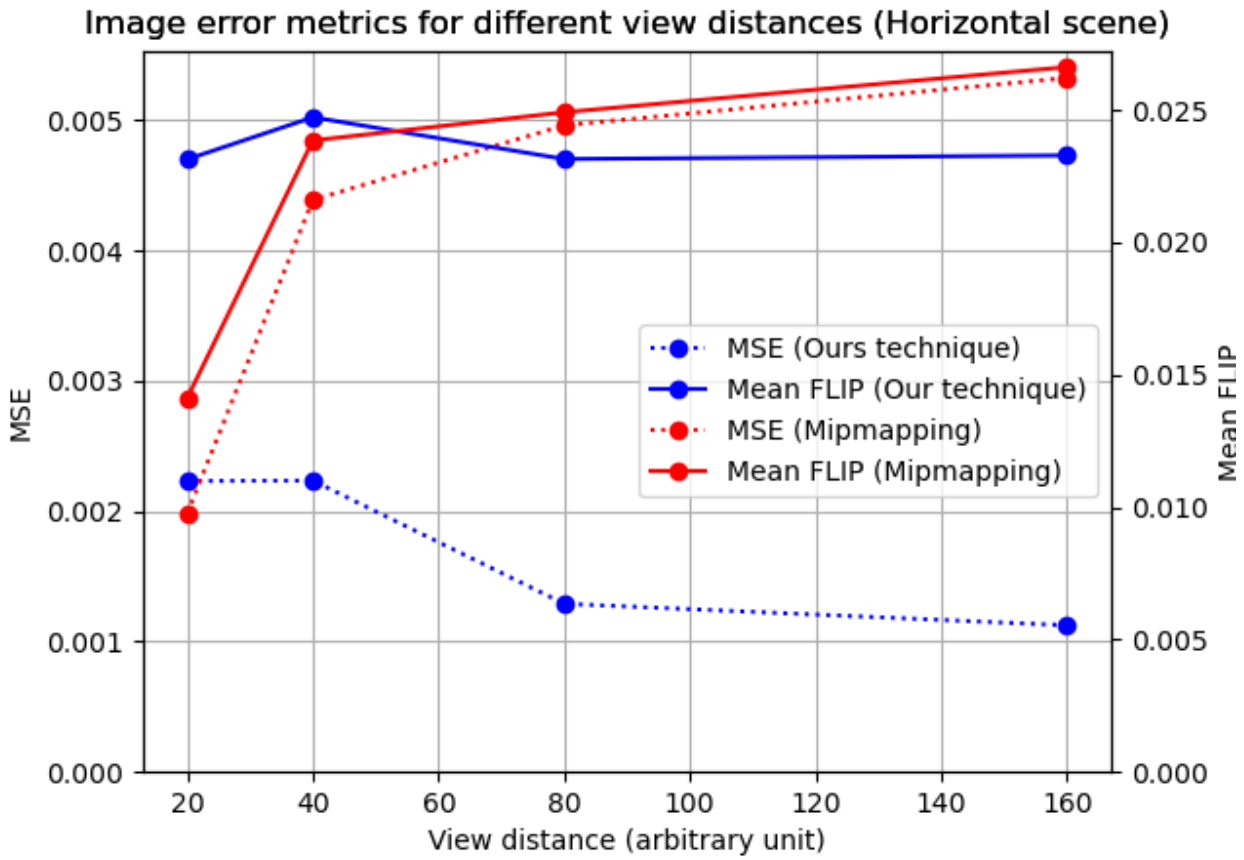


Figure 4.42: Pooled error metrics between two methods and the ground truth as a function of view distance, for SCENE 2. The FLIP metric is more forgiving to the misshapen highlight generated by the baseline method. However, MSE is higher with the baseline than with our technique, and the gap increases as the distance increases.

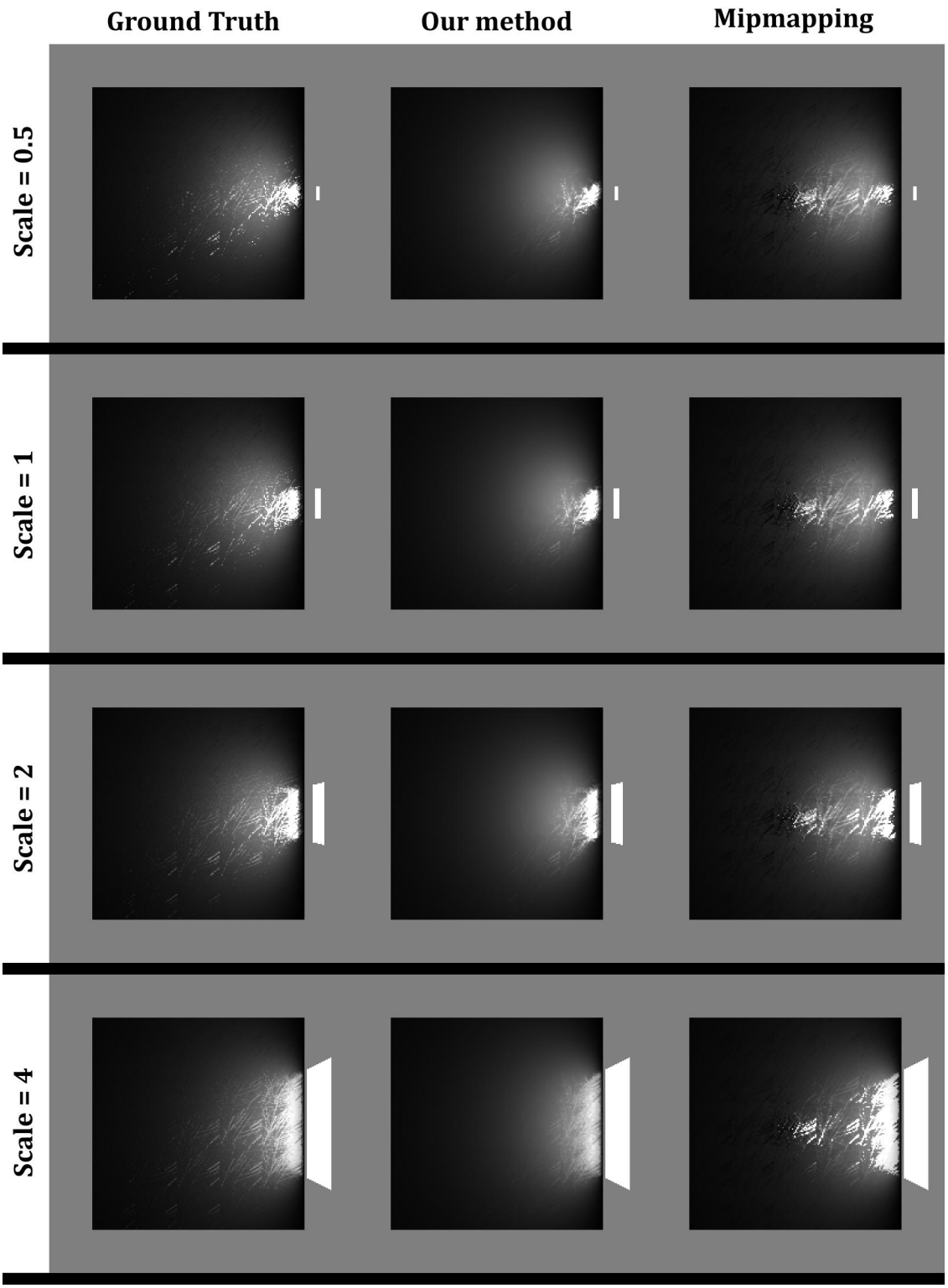


Figure 4.43: Visual comparison for several light scales, for SCENE 1. With the smaller light source, it is obvious that our technique cannot display some of the finer, high energy glints. At larger light scales, we also miss the glints that are further from the main light reflection.

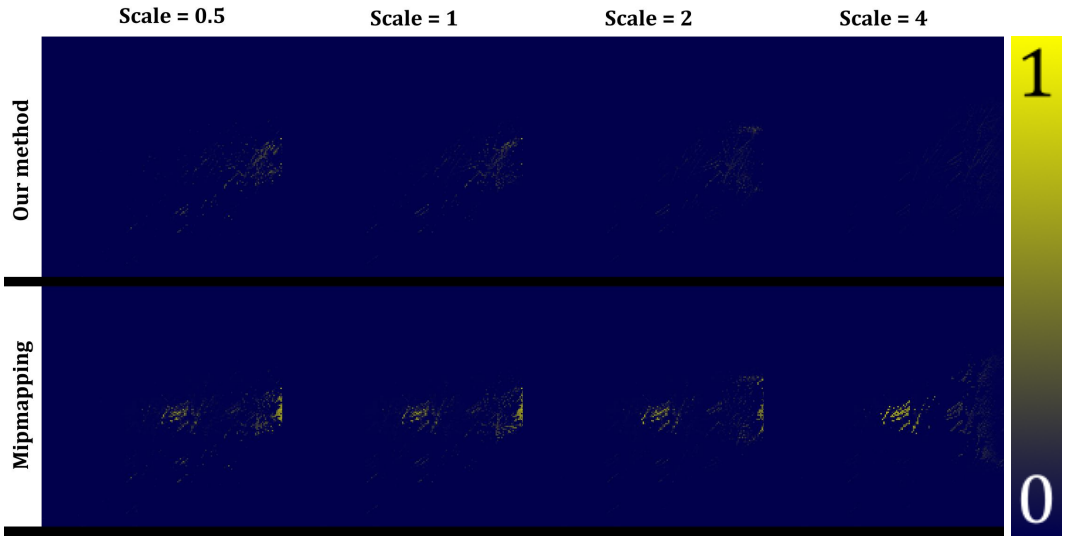


Figure 4.44: Squared error between our method and the ground truth, and between the naive method and the ground truth at several light scales, for SCENE 1.

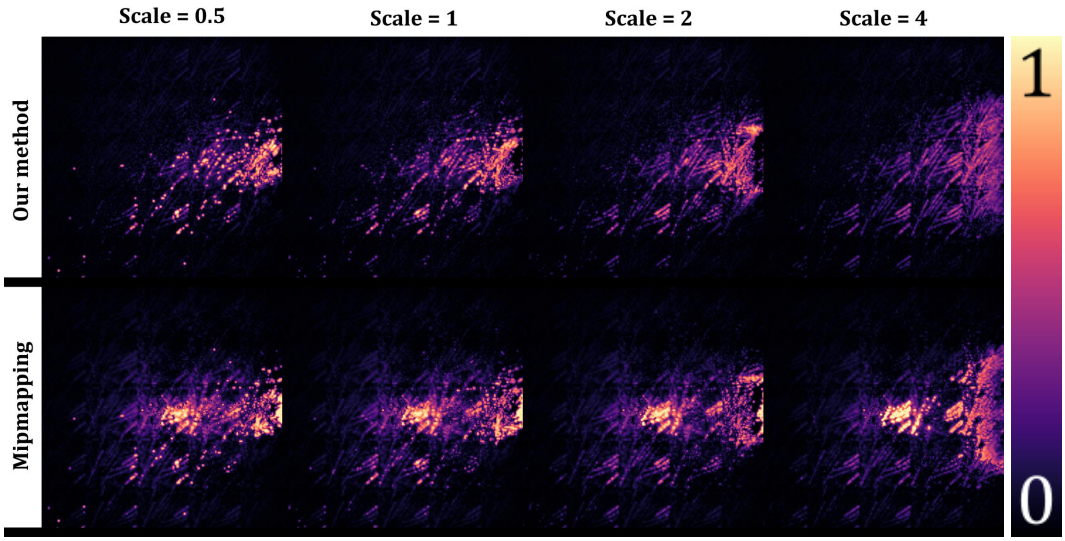


Figure 4.45: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several light scales, for SCENE 1.

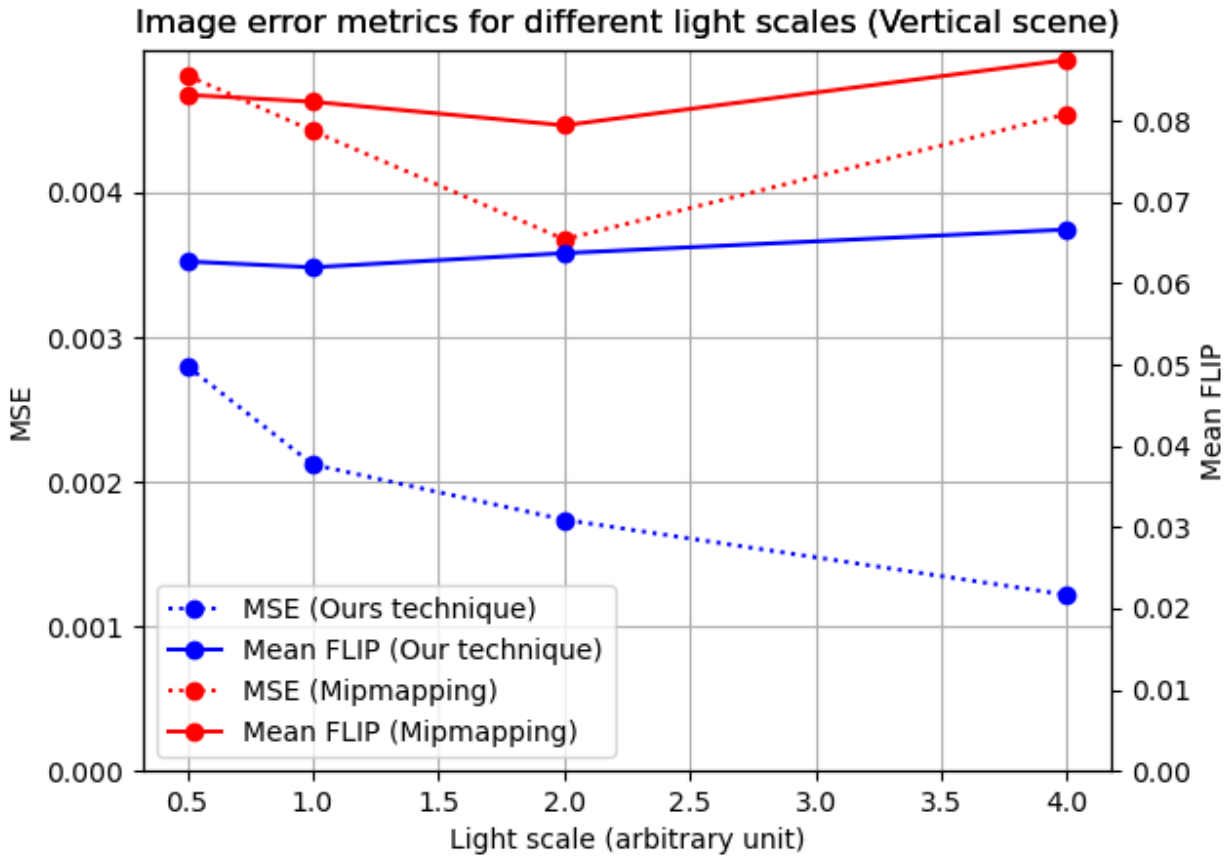


Figure 4.46: Pooled error metrics between our method and the ground truth, and between the baseline and the ground truth, as a function of light scale for SCENE 1. For both image metrics and at all light scales, our render performs better than the baseline.

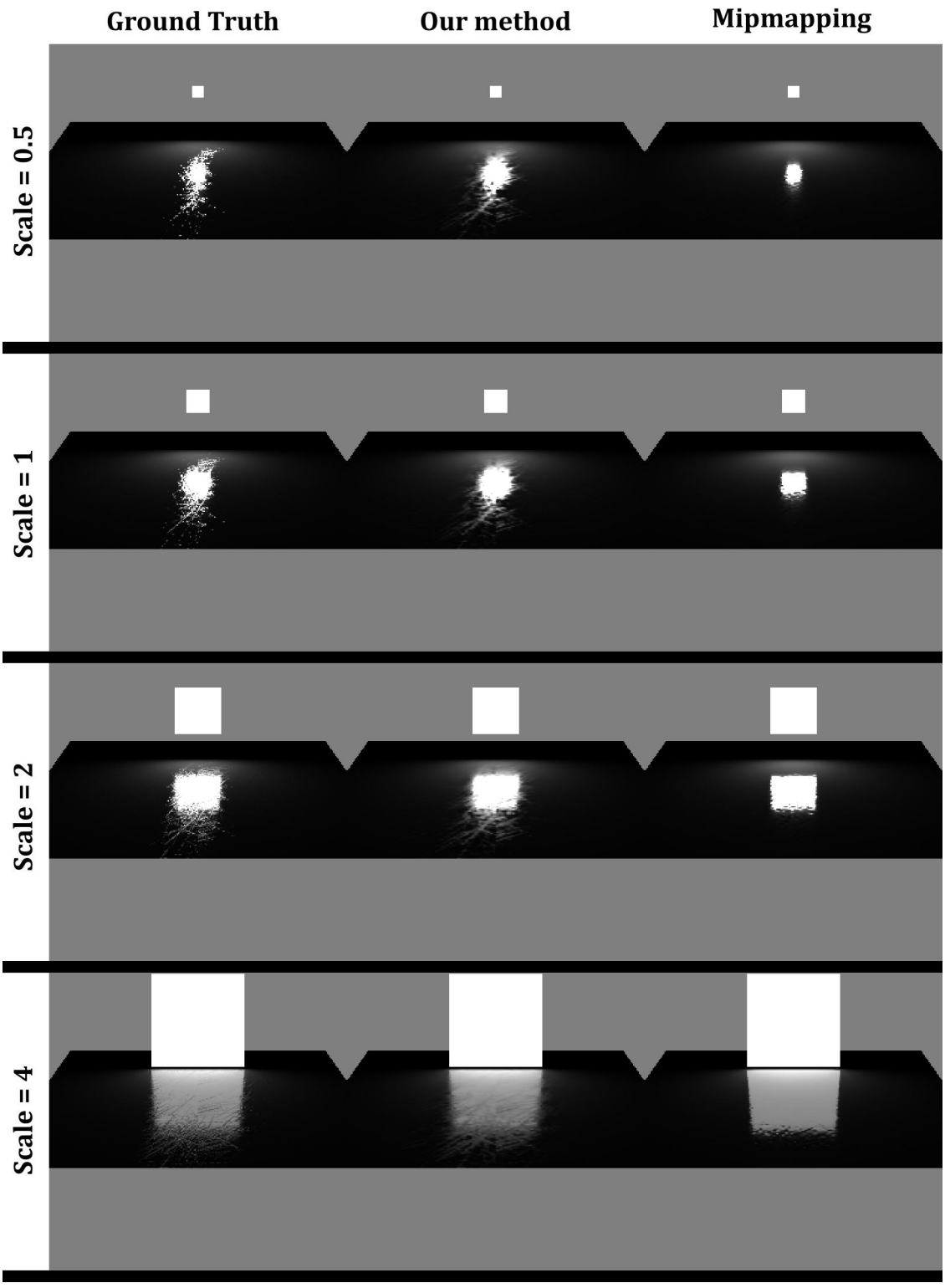


Figure 4.47: Visual comparison for several light scales, for SCENE 2. The overblurring of our method is obvious when the area light is small, and is not as perceptible when the light is large. Our techniques preserves the shape and direction of the details, unlike the baseline which shows a much smoother surface.

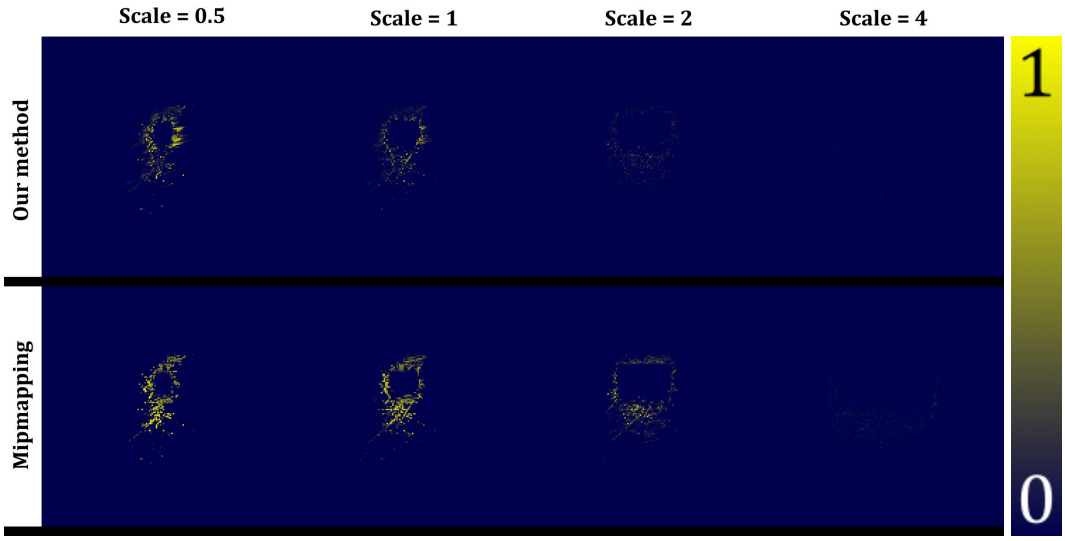


Figure 4.48: Squared error at several light scales, for SCENE 2.

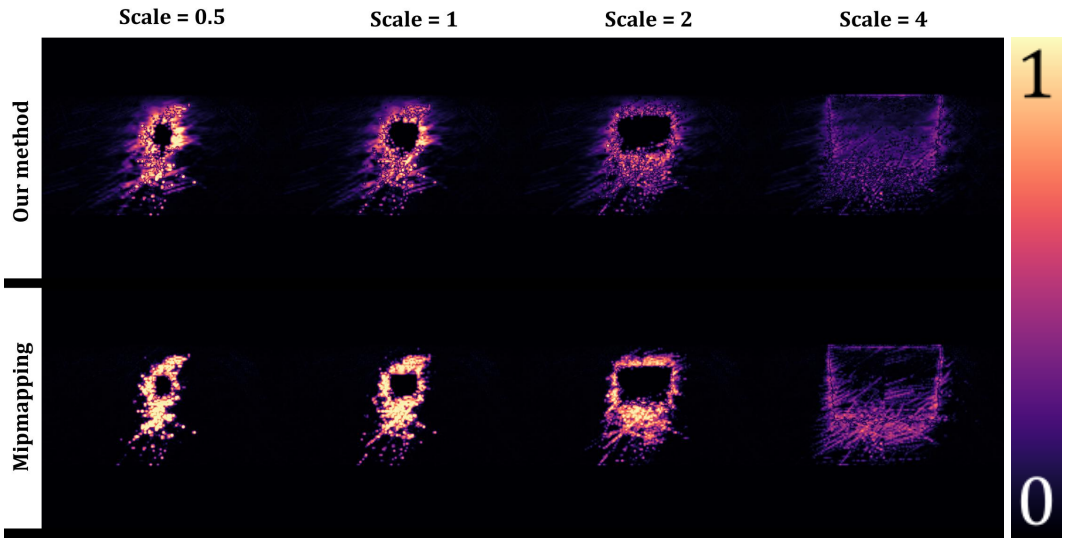


Figure 4.49: FLIP at several light scales, for SCENE 2.

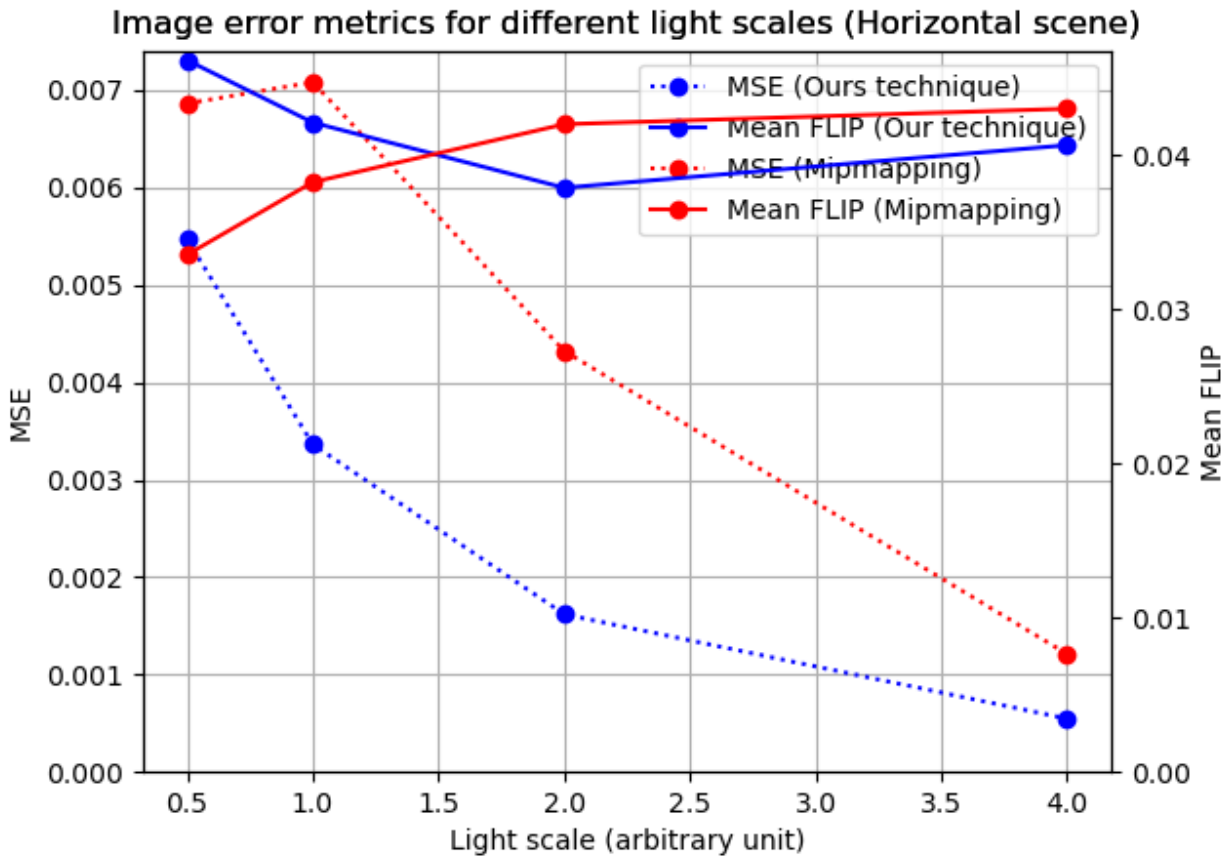


Figure 4.50: Pooled error metrics between the two methods and the ground truth as a function of light scale, for SCENE 2. While both techniques' MSE decreases as the light scale increases, our technique is more accurate as it is able to display the scratches in the middle of the main light reflection. Both methods yield the same approximate FLIP score, since the differences appear on fine details and 1 pixel wide scratches.

4.5 Specular component only

We study the error metrics for the specular component only, for the STONE normal map, as a function of view distance. In the baseline and ground truth methods, this means only using the tabulated LTC parameters, and not using the diffuse cosine distribution. For our own method, this means only using the specular pre-filtered LUT.

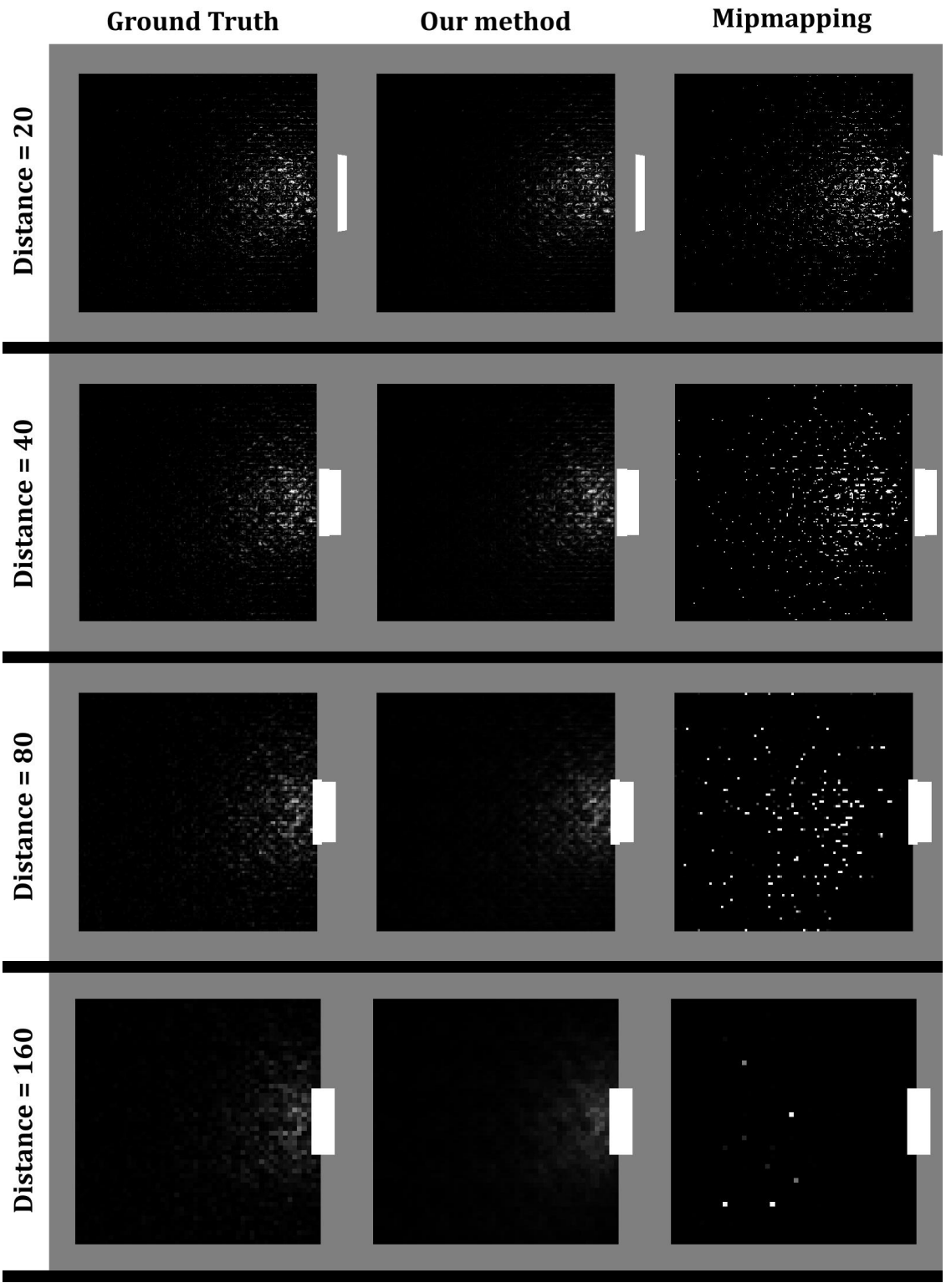


Figure 4.51: Visual comparison for several camera distances for SCENE 1, for the specular component only.

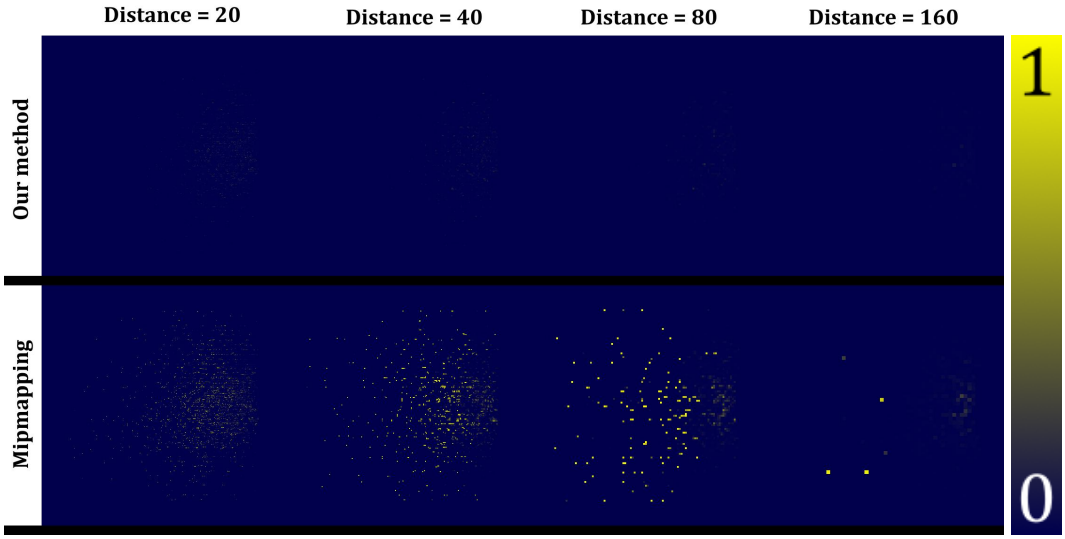


Figure 4.52: Squared error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1, for the specular component only.

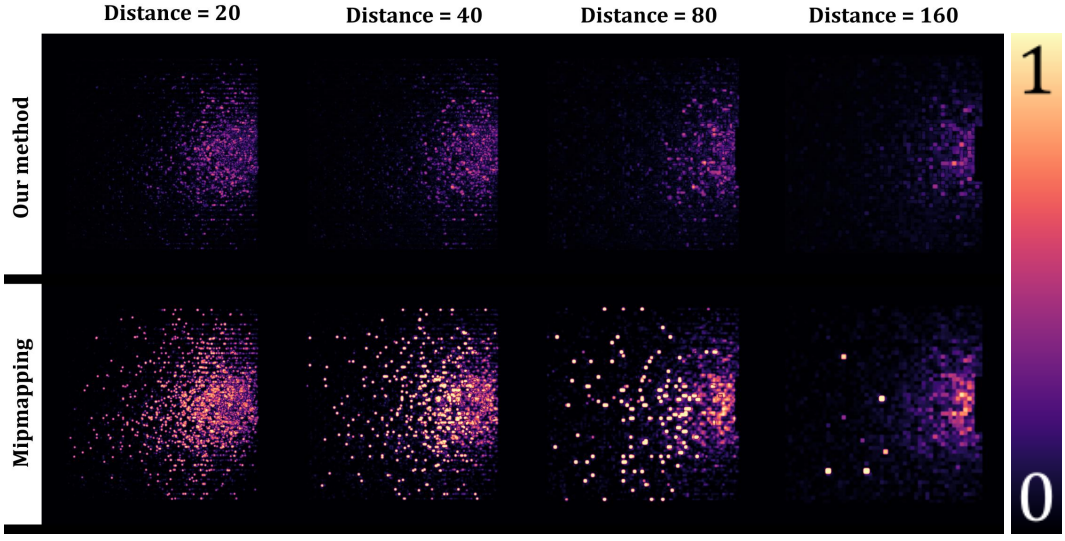


Figure 4.53: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 1, for the specular component only.

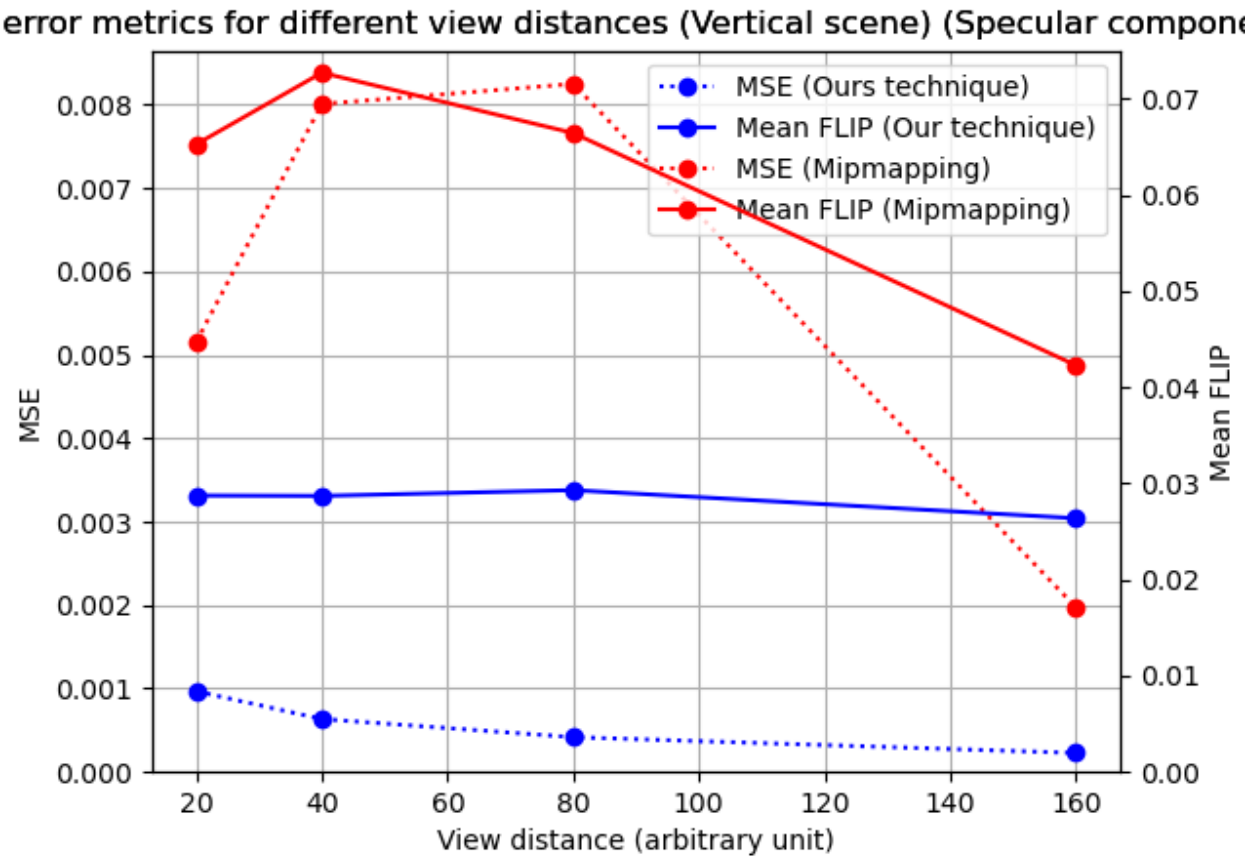


Figure 4.54: Error metrics pooled over the whole images, as a function of view distance, for SCENE 1, for the specular component only. Without the diffuse component (compared to figure 4.6), both of our method's error metrics are a lot more stable throughout view distances. This might indicate that our diffuse component is not filtered correctly.

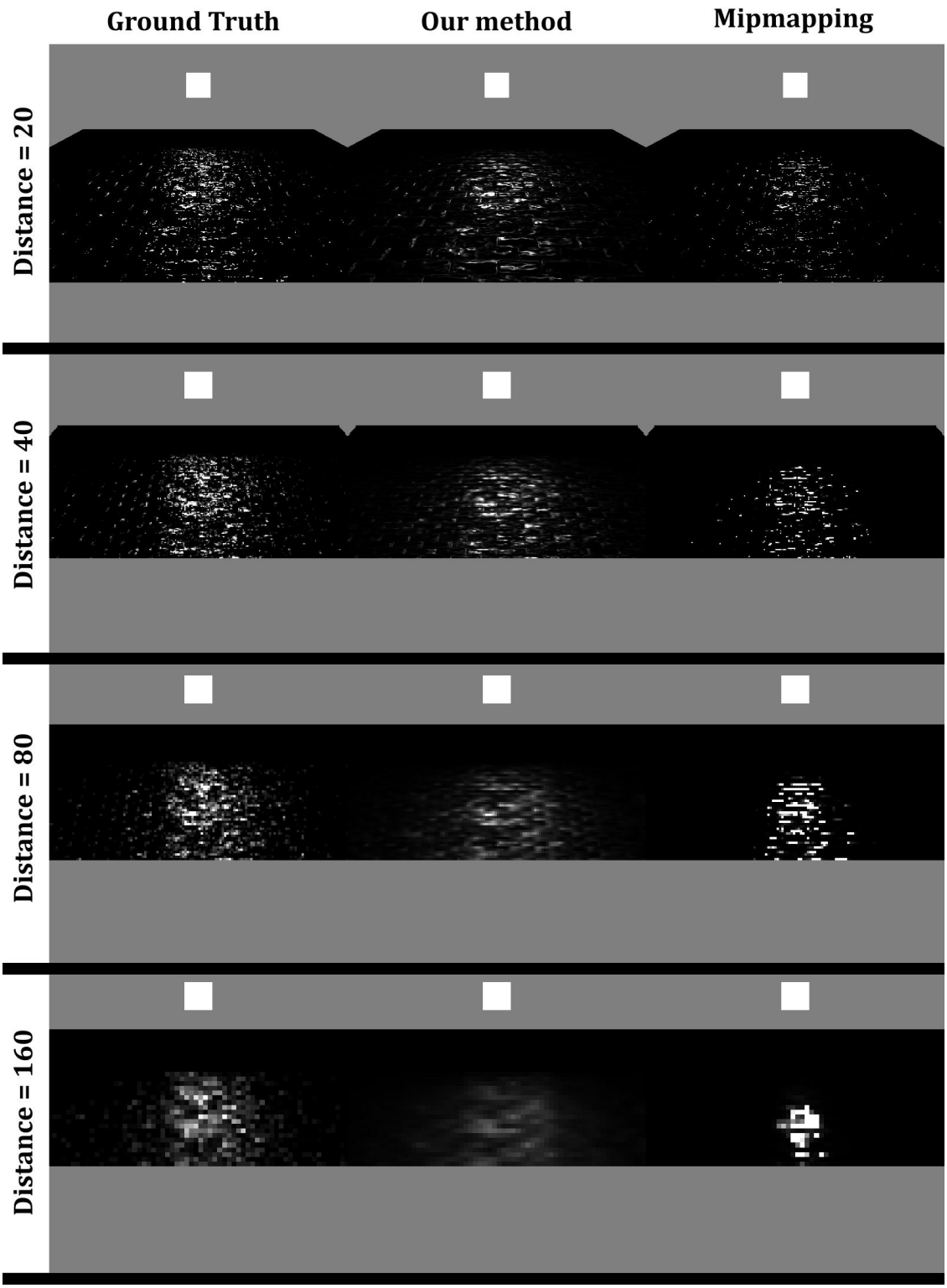


Figure 4.55: Visual comparison for several camera distances for SCENE 2, for the specular component only. We clearly see the overblurring of our method's specular component at grazing angles.

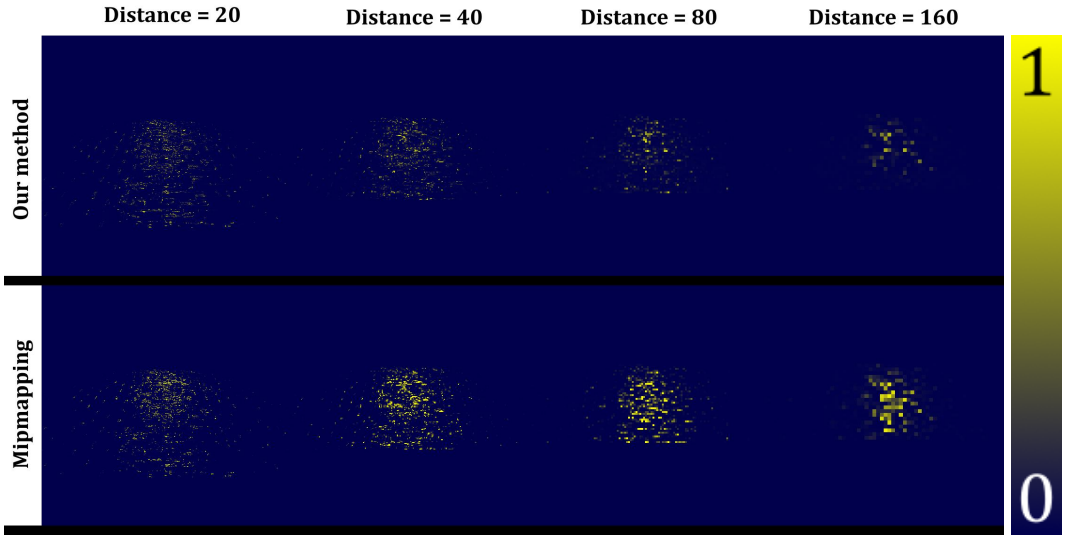


Figure 4.56: Squared error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 2, for the specular component only.

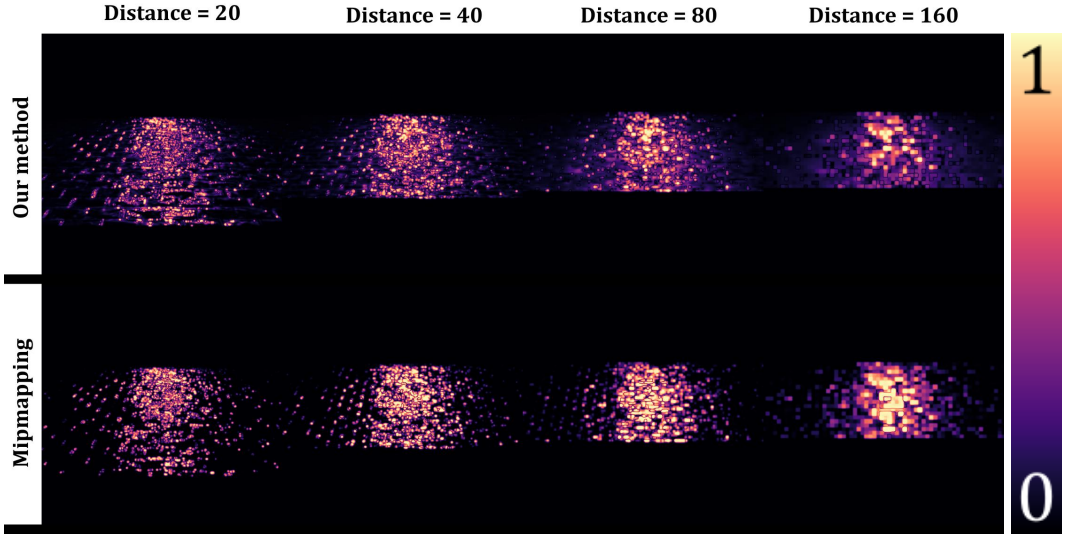


Figure 4.57: FLIP error between our method and the ground truth, and between the naive method and the ground truth at several view distances, for SCENE 2, for the specular component only.

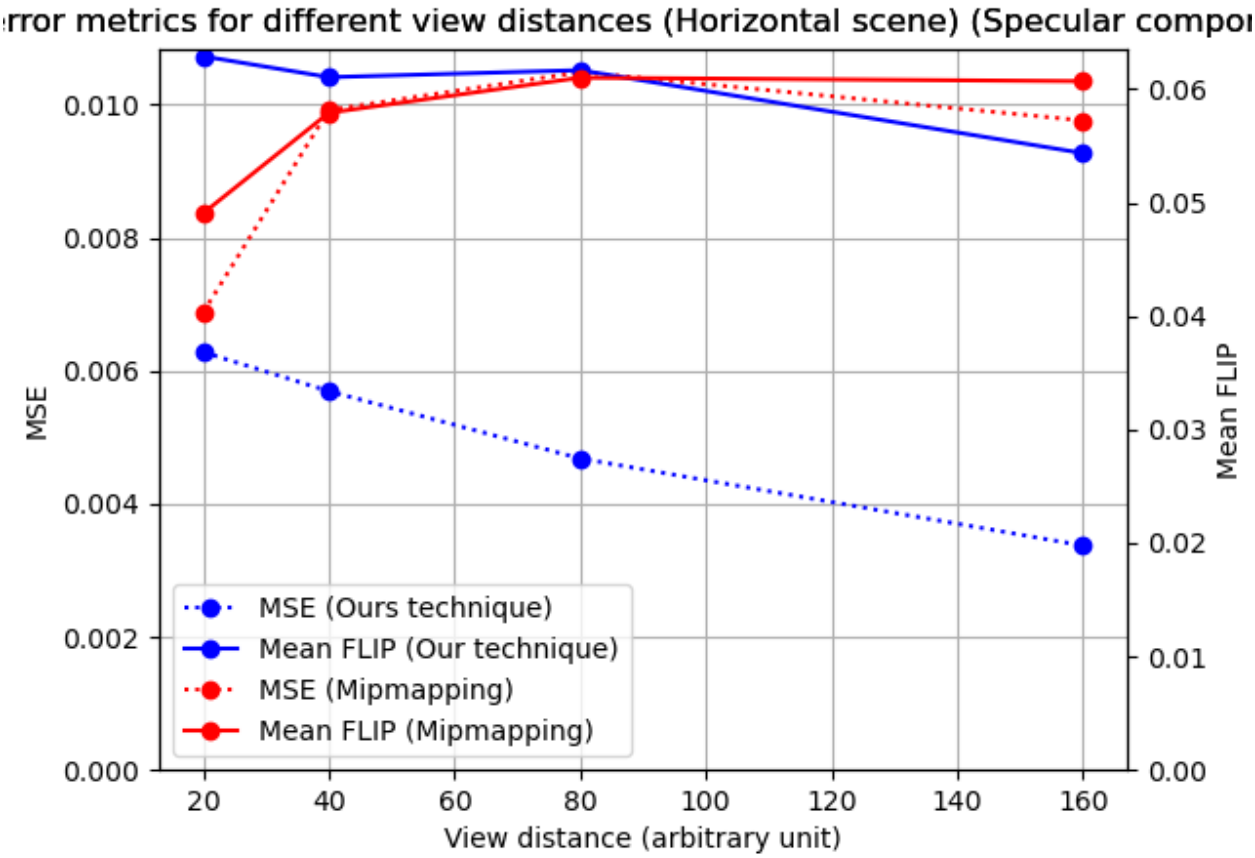


Figure 4.58: Error metrics pooled over the whole images, as a function of view distance, for SCENE 2, for the specular component only. As with SCENE 1, when there is no diffuse component, both our error metrics are more stable and decrease gradually as the view distance increases.

4.6 Storage and time measures

We now review the time and space performance of the three techniques we compared.

Gnd	Ours	Naive
29 KB	24 MB	29 KB

Table 4.1: Storage ocuppied by the textures/LUTs to run each technique, with a normal map of size 128×128. This is the data that needs to be loaded to video memory at render time.

We first show performance for the STONE normal map in SCENE 1, at a resolution of 512×512. The measurements here were made on a ground truth scene on both CPU and GPU, with 49 samples per pixel. They were measured in the real-time rendering portion, at the moment of LTC integration. We are mindful that the naive technique and ours run in constant time on the GPU, but the ground truth technique's cost increases with the number of samples per pixel. We note that at a certain sampling density (strictly more than 49 spp), the ground truth supersampling technique does not fit in GPU memory, and we must switch to a much slower CPU rendering.

Gnd (GPU)	Gnd (CPU)	Ours (GPU)	Naive (GPU)
1512	>40000	163	46

Table 4.2: Render time per frame (ms). Average over 20 frames.

	Gnd (GPU)	Gnd (CPU)	Ours (GPU)	Naive (GPU)
CUDA mem reserved	5964	1184	1285	208
CUDA mem allocated	1424	80	149	88
RAM usage	1380	1531	1507	1370

Table 4.3: Memory usage (MB) to render the scene at resolution 512×512. Average over 20 frames.

Due to the environment in which the method was implemented, there is overhead in the python application such that it is difficult to completely isolate the cost of the rendering part of the algorithm. We derive the memory overhead of the python framework (and of our technique, by extension), by rendering the same scene at different resolutions. We obtain these numbers with a NVIDIA GeForce GTX 1050 Ti with 8GB shared video memory.

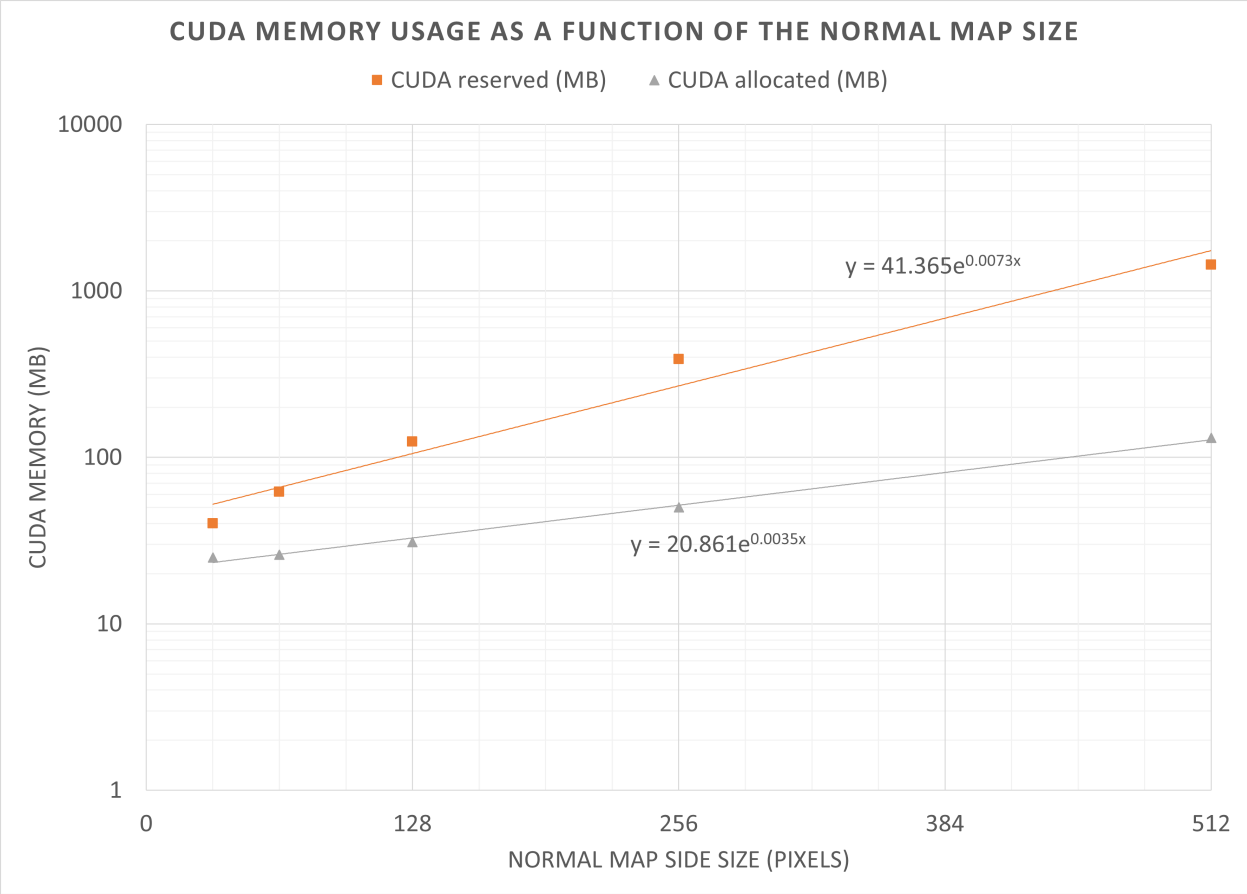


Figure 4.59: CUDA memory used by our technique as a function of the normal map size we filter. We deduce an overhead of 21 MB for the memory allocated. This is coherent with the fact that our filtered LUT is 24 MB in size.

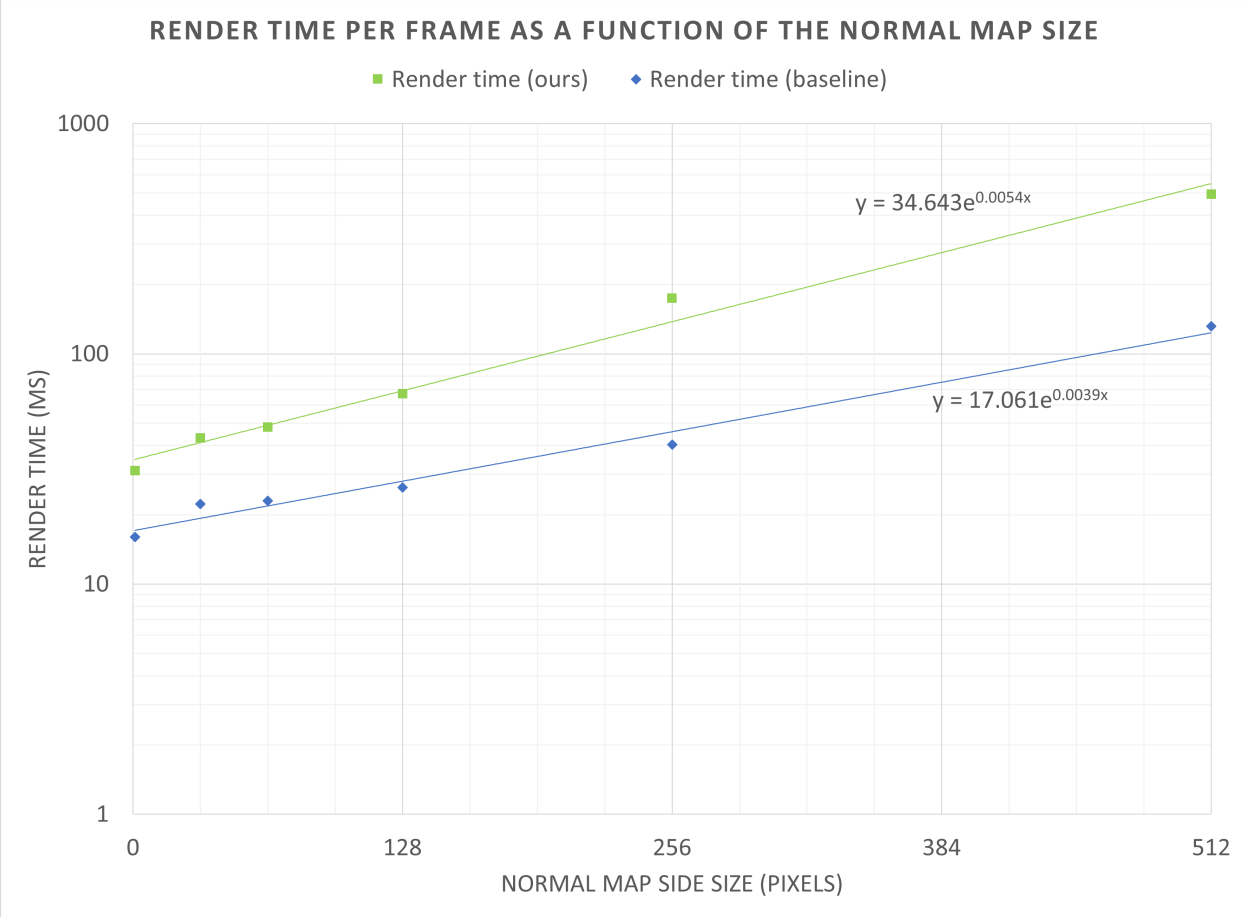


Figure 4.60: Time to render our technique and the baseline as a function of the normal map size we filter (averaged over 20 frames). We see that even at a resolution of 1×1, the baseline takes 17 ms to render, which is the overhead of the python application. At the lowest resolution, our technique takes twice the time to render the scene.

Chapter 5

Discussion and conclusion

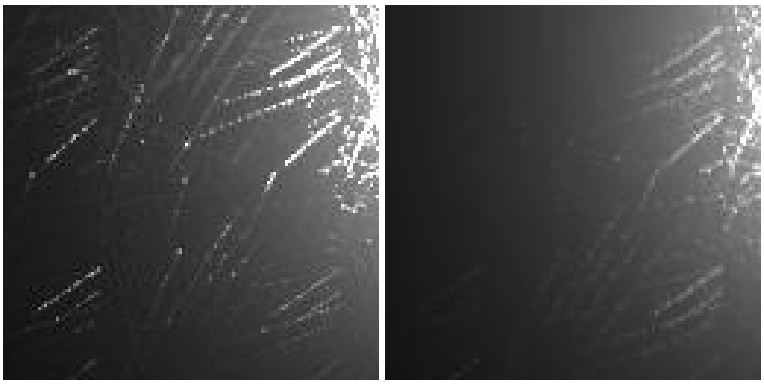
We first discuss the quality of the results, we then suggest directions for future work and finally we conclude.

5.1 Discussion

In view of the results, we can say that we succeeded in filtering the normal map, which involved preserving the detail at the bandlimit of our display resolution, and preventing aliasing. Our technique yields a detailed surface that is free of aliasing artifacts such as reflection smoothing and fireflies, and it accurately shades over the area light.

Our results are also stable throughout several frames, as we do not see flickering. The animated results can be seen on the github repository.

The method we presented is a memory tradeoff, to keep rendering real time at all scales for all light sources. In general, as the light gets smaller, the accuracy of our technique decreases, but the result are still visually pleasing and free of aliasing. On the other hand, when the camera gets further or the display resolution decreases, our methods holds well to the ground truth. We see an accurate translation of many directional highlights into an anisotropic lobe, as discussed in [OB10].



(a) Ground truth (b) Our technique

Figure 5.1: Comparison of a glinty region of the SCRATCH normal map.

5.1.1 Limitations

There are some cases in which we did not capture the glinty appearance of surfaces (figure 5.1). This is especially significant when the detail is very small, and when the area light is small, as our method was limited to fitting a single lobe to a highly multi-modal function. We think that in such cases where the glints are high frequency, the point light approximation can be used in conjunction with a standard filtering method such as the one in [OB10].

The energy loss we observed in many scenes could be fixed with a single factor, but it is a limitation that needs to be investigated. This is especially bothersome in normal maps with small details such as GLITTER and SCRATCH, where a lot of the energy from the light is captured by small glints that are further from the main reflection.

We could observe overblurring in several configurations for SCENE 2 (at grazing view angles), which shows the limitation of fitting a single lobe to the comlpex eBRDF. It also reflects the inaccuracies inherent to LTCs when integrating over an area light at grazing view angles. It was therefore unsurprising that performance was generally better in SCENE 1 than in SCENE 2.

Finally, a main drawback of our technique is the lack of control over the material roughness. When fitting a base BRDF, its roughness dictates the size/width of the LTC

lobe. As we fit our distribution to the eBRDF, we have little control over the roughness, especially if we are aiming for a more specular material. We have no control over the small amount of blurring that occurs with our technique; and it is both positive as it ensures an aliasing-free render, but it is also a key reason why it cannot exhibit high energy glints at a distance.

5.1.2 Disk and memory usage

To our knowledge, real time filtering techniques rarely fit a distribution to the effective BRDF as it is view dependant, and tabulating at a sufficient resolution involves significant storage space. Indeed, our technique requires more space, as we prefilter a view-evaluated reflectance function similar to [KHDN22]. A challenge was to ensure that it still fits in GPU memory for fast lookup and interpolation. We therefore saved a lot of memory usage by starting the filtering at MIP1. The size of our LUT increases $O(m^2)$ with the normal map resolution, so adding a mip level would double the size of our LUT. As a reminder, for a 128×128 normal map occupying 13 KB on the disk, the size of our filtered LUT is 24 MB.

Furthermore, increasing the tabulation resolution did not improve the visual accuracy but it significantly increased the size of the LUT, so the compromise we found was 8 elevations θ and 16 azimuths φ. This is because the limit of our technique's visual accuracy is probably not caused by the low angular sampling of view directions, but by the nature of the spherical function we are trying to fit.

Previous antialiasing methods prefer to fit a function to the NDF, and add the reflectance component at runtime. The effective BRDF is a convolution of the NDF with the base BRDF, which often manifests as a shifted blur filter applied to the NDF [HSRG07]. Therefore, it is lower frequency than the patch NDF, and easier to fit using a smooth spherical distribution like the LTC. We observed, however, that some normal maps had details and patterns that are difficult to filter using a single 9-parameter LTC lobe. These functions, while they are

practical for tabulating and interpolating between close view directions, cannot represent spikes or multi-modal distributions.

5.1.3 Real-time performance

We inherited some performance limitations of [KHDN22], notably the cost of inverting the LTC matrix at runtime; but this step is necessary if we want to keep our coarse tabulation resolution. In theory, the end of the algorithm is as fast as [KHDN22] and [HDHN16], since there is one LTC distribution to integrate per pixel for the specular component, and one for the filtered diffuse component. However, due to inefficiencies in our implementation, it is possible that the performance results we obtained are not representative of the potential of our technique.

The costliest part, specific to our algorithm, is the 2 dimensional software interpolation of the LTC parameters, across azimuths and elevation. Indeed, the mip interpolation can only be trilinear: between adjacent mip levels, and between adjacent texels in the u and v directions. Thus, we had to add software interpolation of the view directions, and that meant a much higher GPU memory usage at runtime. The hardware interpolation across mip texels and levels is fast, but has a heavy memory footprint due the GPU copy of all views at each pixel.

5.2 Future work

Many potential future works could be explored, both in the pre-filtering algorithm and in the real-time part. First, there are performance improvements to be made in pre-processing, for example by dividing the normal map into chunks, and fitting them on different machines concurrently. This would yield time improvements for the same size normal map, but it could also allow for larger normal maps to be filtered efficiently on the GPU. Indeed, with our current pytorch approach, the base GGX samples for normal maps larger than 128×128 do not fit in GPU memory.

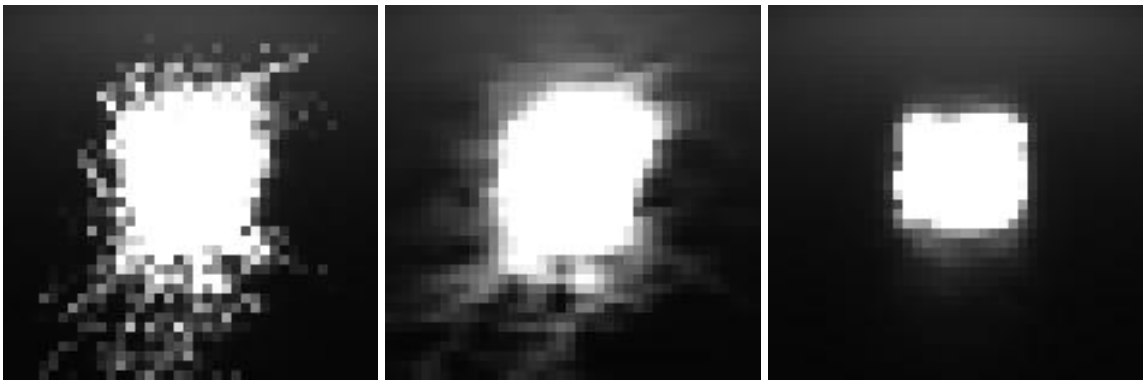
Then, we could explore the benefits of fitting more than one LTC lobe to the specular effective BRDF. By clustering the eBRDF samples, and fitting one lobe to each cluster, we suspect that it would yield a higher quality render with brighter glints, still free of aliasing. This would of course translate into higher memory usage, so the goal would be to find the right compromise between space and visual quality.

Another aspect to explore is the use of different base BRDFs. We already tabulate the base GGX at each microfacet for many azimuthal angles. Thus, using an anisotropic base BRDF would not be more expensive in terms of pre-computation or storage and we could see the effect on the appearance of surfaces.

An important future work will be to adapt our technique to curved meshes. Indeed, while our system supports arbitrary mesh inputs, we noticed that when the view angle is low (grazing the surface), the visual quality was inconsistent and we often ended up overblurring the specular reflection.

Furthermore, there are perspectives to improve memory usage. For a relatively easy gain of space, we could store float16 matrix parameters in the LUT (instead of float32). We were unable to do this due to the lack of float16 support in some pytorch functions. We could also explore optimizations where we only fit some parameters of the LTC matrix, such as [HDHN16] who only fit 4 parameters for isotropic LTCs. Thus, we could try fitting any number of parameters between 4 and 8. This is especially promising to antialias the diffuse component, which is lower frequency. There might be other ways to improve the fitting of the diffuse component since it showed true difference with the ground truth.

There is a paradox we are trying to solve here, since we want to reap all the benefits of high resolution data (high frequency glints) while drastically minimizing the time or the storage cost. Therefore, a key avenue of research for pre-computation based methods such as ours, is the use of data compression. This would allow us to keep as more information about the effective BRDF distributions, while using less space. Since each effective BRDF (or NDF) uses information that can be found in lower levels of the mip hierarchy, there is some redundancy that can be exploited with techniques like compressed sensing.



(a) Ground truth (b) Our technique (c) Naive baseline

Figure 5.2: Comparison of the shape of the polygonal light reflection in the SCRATCH normal map, at a view distance of 80.

Finally, we believe some improvements can be considered in the real time rendering portion. The mip hardware enables us to trilinearly interpolate between adjacent mip tiles and mip levels. Since our table is 5 dimensional, the method would benefit from fast quintilinear interpolation, but we ignore if it is possible using our hardware. Additionally, since elevation and azimuthal samples are distributed spherically, we could explore the use of spherical interpolation (SLERP) between adjacent samples to estimate missing values. This could be especially beneficial in the azimuthal direction, which has a lower angular resolution.

5.3 Conclusion

In this work, we investigated the use of linearly transformed cosine distributions to tackle the problem of normal map appearance filtering, under area light illumination.

We first reviewed the fundamentals of light transport and the reflectance of surfaces. We looked at the bidirectional reflectance function, and the angular integration required when the incident radiance's domain is an arbitrary solid angle on the hemisphere. We then studied existing real time polygonal illumination methods, especially LTCs [HDHN16], and its extension to anisotropy [KHDN22]. We examined techniques that model surface

micro-geometry, and the difficult problem of shading aliasing when using an explicit normal map. This is especially striking with surfaces that present glints and small highlights at the same time as macro geometry visible to the viewer. Using the definition of the NDF and the effective BRDF, we reviewed techniques tackling normal map filtering, offline and in real time.

We then presented our contribution, based on anisotropic mip-mapping and LTC fitting. We pre-filter the normal map by computing the shape of the effective BRDF at each mip texel, level, view azimuth and elevation; and fit it with a single LTC lobe. At render time, we look up the correct LTC lobe in a large mip hierarchy using hardware interpolation, and software interpolation for the view angles. We finally integrate each LTC over the polygonal light using an analytic technique [HDHN16].

Our technique renders satisfactory antialiased results in real time, that can accurately shade over an area light. Also, our technique optimizes a directional patch of microfacets into a directional anisotropic lobe, which preserves the shape of the ground truth reflection. This is not possible when simply mipmapping the normal map (figure 5.2). According to the MSE metric, we perform consistently better than the ground truth (due to the absence of high energy fireflies). With the FLIP error metric, in general, we perform better when the view angle is straight on. Our method presents limitations, notably overblurring in some configurations, and it misses fine, high frequency glints when the light is small. Finally, its high memory requirements is a limitation that needs to be addressed.

In conclusion, we explored how the shading aliasing of detailed surfaces is compounded with the difficulty of integrating radiance over an area light. We postulated that shading a normal mapped surface under a polygonal light is an expensive double integration, involving the spatial integral of texels over the pixel footprint, and the integral of BRDFs over the incident radiance. We have seen that solving this integration in real time is a major rendering problem that remains partly unsolved, especially when trying to render high energy glints. However, carefully fitted spherical functions offer a strategy to tackle both.

Chapter 6

Bibliography

[ANAM+20] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D Fairchild. Flip: A difference evaluator for alternating images. *Proc. ACM Comput. Graph. Interact. Tech.*, 3(2):15–1, 2020.

[Arv95] James Arvo. Applications of irradiance tensors to the simulation of non-lambertian phenomena. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 335–342, 1995.

[Bli78] James F Blinn. Simulation of wrinkled surfaces. *ACM SIGGRAPH computer graphics*, 12(3):286–292, 1978.

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19:542–547, 1976.

[BRPP15] Nicolas Bonneel, Julien Rabin, Gabriel Peyré, and Hanspeter Pfister. Sliced and radon wasserstein barycenters of measures. *Journal of Mathematical Imaging and Vision*, 51(1):22–45, 2015.

[BRW89] Daniel R Baum, Holly E Rushmeier, and James M Winget. Improving radiosity solutions through the use of analytically determined form-factors. *ACM Siggraph Computer Graphics*, 23(3):325–334, 1989.

[BWP+20] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 39(4), jul 2020.

[BYRN17] Laurent Belcour, Ling-Qi Yan, Ravi Ramamoorthi, and Derek Nowrouzezahrai. Antialiasing complex global illumination effects in path-space. *ACM Trans. Graph.*, 36(1), jan 2017.

[CLS+21] Xavier Chermain, Simon Lucas, Basile Sauvage, Jean-Michel Dischler, and Carsten Dachsbacher. Real-time geometric glint anti-aliasing with normal map filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 4:1–16, 04 2021.

[CSDD20] Xavier Chermain, Basile Sauvage, J-M Dischler, and Carsten Dachsbacher. Procedural physically based brdf for real-time rendering of glints. In *Computer Graphics Forum*, volume 39, pages 243–253. Wiley Online Library, 2020.

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, jan 1982.

[Deb03] Paul Debevec. Hdri and image-based lighting. *SIGGRAPH2003 Course*, 2003.

[DHI+13] Jonathan Dupuy, Eric Heitz, Jean-Claude Iehl, Pierre Poulin, Fabrice Neyret, and Victor Ostromoukhov. Linear efficient antialiased displacement and reflectance mapping. *ACM Trans. Graph.*, 32(6), nov 2013.

[EWWL98] J.P. Ewins, M.D. Waller, M. White, and P.F. Lister. Mip-map level selection for texture mapping. *IEEE Transactions on Visualization and Computer Graphics*, 4(4):317–329, 1998.

[Fer04] R. Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004.

[Fis53] Ronald Aylmer Fisher. Dispersion on a sphere. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 217(1130):295–305, 1953.

[GGN18] Luis E. Gamboa, Jean-Philippe Guertin, and Derek Nowrouzezahrai. Scalable appearance filtering for complex lighting effects. *ACM Trans. Graph.*, 37(6):277:1–277:13, December 2018.

[HDHN16] Eric Heitz, Jonathan Dupuy, Stephen Hill, and David Neubelt. Real-time polygonal-light shading with linearly transformed cosines. *ACM Trans. Graph.*, 35(4), jul 2016.

[HH17] Eric Heitz and Stephen Hill. Linear-light shading with linearly transformed cosines, 2017.

[HMB+15] Stephen Hill, Stephen McAuley, Brent Burley, Danny Chan, Luca Fascione, Michał Iwanicki, Naty Hoffman, Wenzel Jakob, David Neubelt, Angelo Pesce, and Matt Pettineo. Physically based shading in theory and practice. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. Association for Computing Machinery.

[HSRG07] Charles Han, Bo Sun, Ravi Ramamoorthi, and Eitan Grinspun. Frequency domain normal map filtering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3):28:1–28:12, 2007.

[JHY+14] Wenzel Jakob, Miloš Hašan, Ling-Qi Yan, Jason Lawrence, Ravi Ramamoorthi, and Steve Marschner. Discrete stochastic microfacet models. *ACM Transactions on Graphics (TOG)*, 33(4):1–10, 2014.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986.

[Kar13] Brian Karis. Real shading in unreal engine 4 by. 2013.

[KHDN22] Aakash KT, Eric Heitz, Jonathan Dupuy, and PJ Narayanan. Bringing linearly transformed cosines to anisotropic ggx. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 5(1):1–18, 2022.

[Kir07] David Kirk. Nvidia cuda software and gpu parallel computing architecture. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, page 103–104, New York, NY, USA, 2007. Association for Computing Machinery.

[Knu84] Donald Ervin Knuth. Literate programming. *The computer journal*, 27(2):97–111, 1984.

[Lam60] Johann Heinrich Lambert. *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. sumptibus vidvae E. Klett, typis CP Detleffsen, 1760.

[LDR14] Sebastien Lagarde and Charles De Rousiers. Moving frostbite to pbr. *Proc. Physically Based Shading Theory Practice*, 2014.

[LDSM16] Pascal Lecocq, Arthur Dufay, Gaël Sourimant, and Jean-Eudes Marvie. Accurate analytic approximations for real-time specular area lighting. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 113–120, 2016.

[LHK+20] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics (TOG)*, 39(6):1–14, 2020.

[Noe99] Nicolas Noe. *Étude de fonctions de distribution de la réflectance bidirectionnelle*. Theses, Ecole Nationale Supérieure des Mines de Saint-Etienne ; Université Jean Monnet - Saint-Etienne, September 1999.

[OB10] Marc Olano and Dan Baker. Lean mapping. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, page 181–188, New York, NY, USA, 2010. Association for Computing Machinery.

[ON97] Marc Olano and Michael J. North. Normal distribution mapping. 1997.

[RH02] Ravi Ramamoorthi and Pat Hanrahan. Frequency space environment map rendering. *ACM Trans. Graph.*, 21(3):517–526, jul 2002.

[Sch97] Andreas Schilling. Antialiasing of bump maps. 12 1997.

[TLQ+05] Ping Tan, Stephen Lin, Long Quan, Baining Guo, and Heung-Yeung Shum. Multiresolution Reflectance Filtering. In Kavita Bala and Philip Dutre, editors, *Eurographics Symposium on Rendering (2005)*. The Eurographics Association, 2005.

[Tok05] Michael Toksvig. Mipmapping normal maps. *Journal of Graphics Tools*, 10:65 – 71, 2005.

[TR75] TS Trowbridge and Karl P Reitz. Average irregularity representation of a rough surface for ray reflection. *JOSA*, 65(5):531–536, 1975.

[Vea98] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph.*, 26(2):265–272, jul 1992.

[Wil83] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '83, page 1–11, New York, NY, USA, 1983. Association for Computing Machinery.

[WLWF08] Lifeng Wang, Zhouchen Lin, Wenle Wang, and Kai Fu. One-shot approximate local shading. 2008.

[WMLT07] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, page 195–206, Goslar, DEU, 2007. Eurographics Association.

[XCM+14] Kun Xu, Yan-Pei Cao, Li-Qian Ma, Zhao Dong, Rui Wang, and Shi-Min Hu. A practical algorithm for rendering interreflections with all-frequency brdfs. *ACM Transactions on Graphics (TOG)*, 33(1):1–16, 2014.

[YHJ+14] Ling-Qi Yan, Miloš Hašan, Wenzel Jakob, Jason Lawrence, Steve Marschner, and Ravi Ramamoorthi. Rendering glints on high-resolution normal-mapped specular surfaces. *ACM Trans. Graph.*, 33(4), jul 2014.

[YHMR16] Ling-Qi Yan, Miloš Hašan, Steve Marschner, and Ravi Ramamoorthi. Position-normal distributions for efficient rendering of specular microstructure. *ACM Trans. Graph.*, 35(4), jul 2016.