Department of Computer Science
Augmented Vision

On Machine Learned Predictions of SV-BSDF Parameters for Known Geometries

Master Thesis

Lauritz Feick

Supervisors:

Prof. Dr. Didier Stricker
Dr. Gerd Reis
M.Sc. Torben Fetzer

Kaiserslautern, 27th January 2021

Abstract

The generation of photo-realistic images from digital models is of particular importance in many areas of research and application, such as simulation, marketing and sales or the film industry. An important component in making digital scenes appear real, is the description of the interaction of an object's surface with incident light. BRDFs/BSDFs are commonly used to model the reflective properties of a surface for arbitrary illumination and camera configurations. BRDF estimation is about predicting a suitable model or parameters of a predefined model, that describes the material's behavior well. In particular, stable estimation for a variety of materials, from limited input is a challenging problem. In this work, we give an overview of existing methods, their applicability, advantages and disadvantages, and describe a new approach to the problem that overcomes certain limiting assumptions of existing methods. We propose an architecture that allows to estimate parameters of the widely used *Disney BSDF* based on given 3D scanning data. Specifically, the method is designed to work with the arbitrarily sized output images of a *structured light* or an *active stereo* system. Additional calibrated light sources as well as scans from multiple positions are optional, but help to improve the quality of the predicted parameters. To solve the stated task, we train five similar neural-networks in a supervised manner on a synthetic dataset, that we specifically created for this task. Each of these networks is designed to determine one of the BSDF parameters. This design allows to provide the inference with already known parameters or avoid the prediction of some – making the approach also suitable for *passive stereo* systems with additional light sources. Some of the incorporated networks are dependent on the predictions of others. Besides that, the nets work independently on various scalings of the input. Each stepwise refines its predictions, by fusing intermediate results of a variable number of different input-views and increases the resolution at the same time. Inference is performed almost in real-time and produces spatially varying parameter maps of the same size as the inputs.

Keywords: deep learning, BRDF, BSDF, SV, material estimation, appearance capture

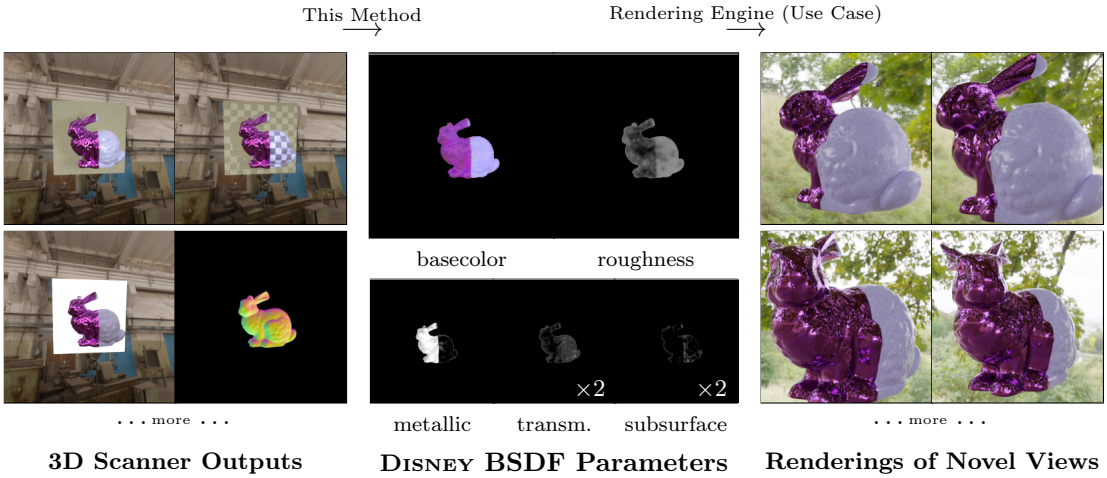


Figure 1: The purpose of the presented method is, to predict the most important material parameters of the Disney/*Principled BSDF* in terms of human perception. 3D scanning data, containing various photographs and geometry information, serves as input. Our predictions can be used, to re-render the scanned scene realistically in novel lighting and/or from novel positions.

Contents

Chapter 1

Introduction

Computer graphics are widely spread and can be found in nearly every domain. Its applications range from movies or games in the entertainment industry, over print media, to user interfaces created for the interaction with all kinds of machines. Not always, but very often graphics are desired to look as realistic as possible. The common way to obtain life-like (and other) images from a computer is, by letting an artist define a scene, that contains one or more objects and further light sources. Subsequently a computer can be used, to compute an image of this scene, taken with a simulated camera. This process is called rendering and can be implemented with different algorithms. Especially notable are such ones, that are able to produce realistic looking images from properly specified scenes. The creation of these scenes can be very laborious. Therefore various advances have been made to support artists in their workflow and to automate the overall process. One example is the utilization of 3D scanning, which became more and more common in the recent years. It allows to capture real objects' geometries. Because such objects are very often very rich in details, modeling them by hand is time consuming and rather prone to errors. In this task, 3D scanning notably supports the creation of artificial objects. There are two reasons for this. First, because the scanned objects contain all the details (up to a certain resolution), but also because the capture is much faster than an artist could model all the details manually. Still, a scene is not solely defined by the geometry of its contained objects. Also important are the material properties of them – describing how the object interacts with light. Again, they have to be defined by the artist as well. To ease this task, libraries with commonly appearing materials had been build in the past. Moreover, such libraries are usually included in modern graphic engines as well. This allows an artist to simply select an appearance (from the database) for each of the objects in the scene. Because these materials are most often defined by a multitude of parameters[1], this assignment work-flow already speeds up the overall process, of creating a renderable scene, a lot. Yet, this approach is far from perfect, because it is only possible with spatially constant[2] materials or repetitive tiles (that are small images) of materials, that seamlessly connect when placed next to each other. If a single object contains many different materials, that are also non-repetitive, this assignment task gets harder for the artist. This becomes even worse, if the material-database does not contain one, that describes the object sufficiently well. Unfortunately, especially real-world objects are best described by such non-repetitive spatially varying materials. For an artists' efficiency it would be highly beneficial to have a tool, that automatically assigns proper materials to scanned objects. This is actually the main driving idea behind this work – the prediction of a material based on the data of a 3D scanner. For comparison, the classic way to create such a material, would be the following: An artist, would manually draw each parameter individually on the object. The way, this works, is by using an image, that stores the parameter values and is mapped onto the 3D object before rendering. However, many graphic engines simplify this task as well, by allowing to directly paint on the object. For materials, that are spatially varying with a high resolution, their creation is especially time consuming. However, they are necessary

[1] In general, color is just one aspect of it. Reflectiveness and more are used as commonly.
[2] With regard to the surface location of an object.

if realistic graphics are sought to be generated. As mentioned already, this material-assignment can be automated with the utilization of 3D scanner information. Note, that different types of 3D scanners are available. We are only interested in image based approaches, even though other scanner types (like laser or time of flight) exist as well. Image based approaches calculate geometry from ordinary images. Besides the calculation of the object geometry, these also allow a simple query of the corresponding pixel values from the captured images. The respective values can be used as an approximation to the objects inherent color for any chosen material descriptor. The problem with this is, that the images were taken under certain lighting conditions and the camera(s) had a specific orientation to the object. This means, that the illumination of the scene at the time of capture is baked into the so approximated color of a 3D object. This is especially problematic for highlights and other reflections in the images. Approximation errors created in this way would destroy the illusion of a re-render. Especially if the object is shown from a different position than the one used for the scan or if the lighting of the scene is changed for re-rendering. Thus, it is desired to remove such illusion-breaking imperfections of the object's approximated color. Further still, it would be necessary to also determine the other material parameters (such as reflectance or transparency) of the scanned object. If this is done spatially varying with respect to the object's surface, with a sufficient resolution well enough, then realistic looking re-renderings are possible. Such properly calculated materials allow life-like renderings under arbitrary light-camera configurations and moreover with novel scene illuminations. BRDF/BSDF estimation is the umbrella term for algorithms that deal with the automated computation of such material properties. However, these techniques vary in their use cases and therefore not all work with 3D objects or even spatially varying materials. In this work we give a brief overview of different approaches, that are used for BRDF/BSDF estimation. However, as pointed out already, we are mostly interested in the estimation of non-uniform materials for arbitrary 3D objects. To achieve this, we propose a novel design of a learning-based algorithm, that does this computation. We work with the Disney/Prinicpled BSDF and determine some of its parameters on the basis of 3D scan data accordingly. This is a major part of what we call the main contribution of our work. The second part of our contribution is the creation of a dataset, for the training of the proposed architecture.

To summarize this thesis, it is divided into the following chapters: First, in Chapter 2.1, we describe in more detail what BRDFs/BSDFs are. Then, in Chapter 2.2, we concretize the problem description of BRDF/BSDF estimation and show which approaches to it already exist and how they differ (in Chapter 2.3). The main contribution of this thesis is introduced with the description of our particular problem (Chapter 2.4), before presenting our method for solving it in Chapter 3.2 – an architecture consisting of five neural networks that are partially interdependent. To enable training of these networks, we first create our own dataset (Chapter 3.1) – the second part of our contribution. Finally, in Chapter 4, we examine the obtained results and show visualizations of them. We conclude the work with Chapter 5.

Chapter 2

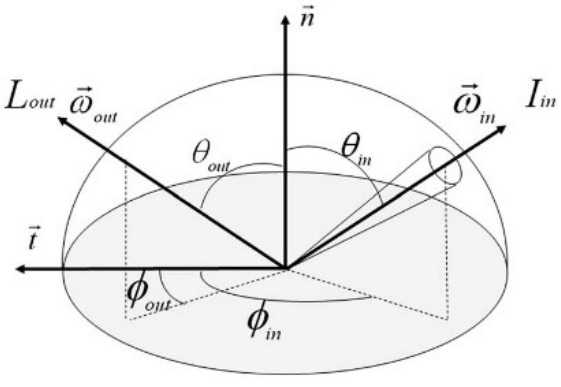
Preliminaries

2.1 Bidirectional Reflectance Distribution Functions

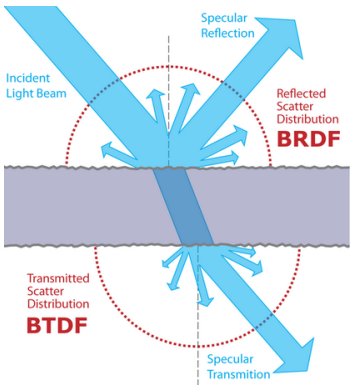
The de facto standard way in computer graphics to describe a materials visual properties is, by using a *bidirectional reflectance distribution function* (BRDF) f_r [23]. Such a function describes how strongly L_{out} incident light I_{in} from one direction ω_{in} is reflected from the surface to another outgoing one ω_{out}. These directions are defined (as shown in Figure 2.1a), by the *zenith*-angles Θ_{in}, Θ_{out} relative to the surface normal and the *azimuth*-angles Φ_{in}, Φ_{out} relative to the surface tangent (i.e. the rotation around the normal). Strictly speaking, the light intensity is not defined per ray, but physically by a square unit area $d\omega$ around it (defined by $d\Phi$ and $d\Theta$), called the *solid angle*. In the most general case a BRDF is defined as the fraction of reflected light to incoming light, as in Equation 2.1:

$$f_r(\omega_{in}, \omega_{out}, \dots) = \frac{dL_{out}(\omega_{out})}{dE_{in}(\omega_{in})} = \frac{dL_{out}(\omega_{out})}{I_{in}(\omega_{in})cos(\Theta_{in})d\omega_{in}} \tag{2.1}$$

The cosine term in this equation comes from projecting the solid angle onto the materials surface. This models the light arriving on the material surface E_{in} (*irradiance*) per unit area. The actual input-variables of f_r depend on which type of BRDF is used. However, the in- and outgoing



(a) BRDF: Surface Geometry (taken from [19])



(b) BSDF Composition (taken from [5])

Figure 2.1: (a) Shows the geometry around a surface point, that is required to define a BRDF. (b) Depicts the idea of extending a BRDF to a BSDF by adding *transmission* and *subsurface scattering*.

light directions are always arguments – usually defined with the Θ and Φ angles and sometimes implicitly specified. One common used variant are *isotropic* BRDFs, meaning that a combined rotation of both rays around the surface's normal does not change the function. For isotropic BRDFs the Φ_{in} and Φ_{out} parameters are replaced by their difference Φ. In other words, Φ is just the angle between the in- and outgoing rays, projected onto the surface. Moreover, the BRDF usually also depends on the considered location u,v on a surface. If the BRDF is not constant for the entire surface, it is called *spatially varying* (SV). Further, the BRDF depends on the wavelength. In practice this is usually simplified with a color space – for example with the three-dimensional RGB-space. In this case three BRDFs would be necessary to describe the overall light transport. To simplify the notation, the wavelength or alternatively the color channel indices are omitted. Thus, the returning values of f_r would be three-tuples, if the RGB-space is used. Moreover, f_r usually is defined with respect to some hyper-parameters, which also depend on u and v. These are usually selected by an artist and sought to be inferred by BRDF estimation. As before, to simplify notation, they are usually omitted as well. In summary a BRDF can generally be described by a function of the following arguments:

$$\text{anisotropic BRDF: } f_r(\Theta_{in}, \Phi_{in}, \Theta_{out}, \Phi_{out}) \tag{2.2}$$

$$\text{anisotropic SV-BRDF: } f_r(\Theta_{in}, \Phi_{in}, \Theta_{out}, \Phi_{out}, u, v) \tag{2.3}$$

$$\text{isotropic SV-BRDF: } f_r(\Theta_{in}, \Theta_{out}, \Phi, u, v) \tag{2.4}$$

For further details please refer to [23] or [30].

Physically Based BRDFs In the past many different BRDFs were developed. Often they are physically motivated (called *physically based*), meaning that physical laws are approximately fulfilled. Deviations from them are usually caused by simplifications, that make their computations less costly. Relevant laws, that should be meet by *physically based* BRDFs are *positivity*, the *Helmholtz reciprocity* and *energy conservation* [16, 23, 11, 22, 17]. *Positivity* (Equation 2.5) states, that it impossible for a surface to reflect a negative amount of light. Methematically this means, the BRDF of any argument must be positive or equal to zero:

$$f_r(\omega_{in}, \omega_{out}) \geq 0 \tag{2.5}$$

Helmholtz reprocity demands, that the role of incoming and outgoing rays is supposed to be symmetric (Equation 2.6) – meaning that for the response, it does not matter whether the roles of the rays are exchanged:

$$f_r(\omega_a, \omega_b) = f_r(\omega_b, \omega_a) \tag{2.6}$$

Conservation of energy (Equation 2.7) requires, that for a surface point the reflected light in all possible directions $d\omega_{out}$ on the hemisphere Ω over that point, can not be stronger than the incoming light. Simply speaking, the material does not emit any light, by itself:

$$\forall \omega_{in} : \int_{\Omega} f_r(\omega_{in}, \omega_{out}) cos\Theta_{out} d\omega_{out} \leq 1 \tag{2.7}$$

Further, the same principles can be used to extend the definition of the BRDF. A *bidirectional transmittance distribution function* (BTDF) [8] extends the BRDF with transparency and a *bidirectional scattering-surface reflectance distribution function* BSSRDF adds *subsurface scattering*[17]. Using both extensions defines the *bidirectional scattering distribution function* (BSDF)[17], meaning that the set of BRDFs is a subset of all BSDFs. What all parametric BRDF/BSDFs have in common, is that they rely on some preselected hyper-parameters. In general these are modeled in a preferably intuitive way, because they are usually selected manually by artists. Over the years many different BSDFs where developed. They vary in computational efficiency and their ability to model differently complex materials well. Some commonly used examples are given in table 2.1.

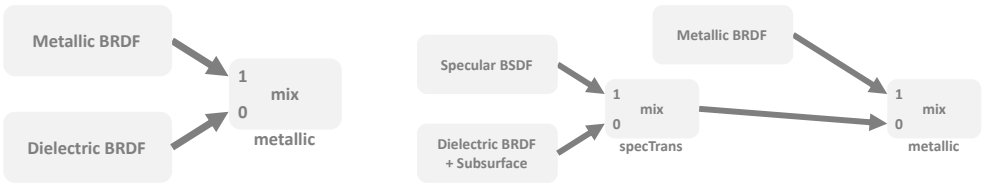
name	type	parameters	PB	well modeled materials
Blinn-Phong [9]	BRDF	specular, diffuse, ambient	no	plastics
Cook-Torrance [13]	BRDF	specular, diffuse, roughness	yes $*$	plastics + metals
Ward [29]	BRDF	specular, diffuse, anisotropy	yes $*$	anisotropic materials
Principled (/Disney) [10]	BSDF	many (refer to Section 2.1.1)	yes $*$	most materials

Table 2.1: Properties of some BRDF. *Physically based* (PB) means, that physical laws are approximately meet. $*$ stands for proper selected parameters.

Note that even tough some parameters have identical names, the BSDFs usually implement them differently. Please refer to [11] or [22] for a more complete overview of different BRDF/BSDFs. Due to the fact, that different BRDFs represent different materials differently well, it has become common practice in the past to work with *mixed* models. This means that not only one BRDF is used to portray a material, but rather a (usually linear) combination of several. The *Principled BSDF* itself is such a mixed function. The design of it as such a mixed function, gives it the ability to model a variety of natural, realistic materials and likewise artificial ones. In addition, the intuitively controllable hyper-parameters have helped the model to become an industry standard and one of the most widely used BSDFs nowadays. For said reasons, we are also interested in modeling our materials with this function. Ultimately, we seek to derive the most relevant parameters of the *Principled BSDF*, in terms of good representation of most real objects. In the next section we give a more detailed description of it. We discuss which parameters are used and how they affect the appearance of a rendered surface.

2.1.1 Disney/Principled BSDF (as implemented by Blender)

The *Principled BSDF*, also known as *Disney BSDF* [10] was first defined as a BRDF [11] and later extended to a BSDF. As mentioned before, it is not a single formula but a model, that combines three individual BSDFs. This is done, because the different models are responsible for different types of materials. Moreover, the models have varying computational effort. The definition as a mixture therefore also helps to reduce the rendering-time for less complex materials. In detail the following three models build the BSDF: a *metallic* BRDF, a *specular* BSDF and a *dielectric* BRDF with additional *subsurface scattering*. This is shown in Figure 2.2, that we have recreated from [10]. As stated before, the aim of the *Principled BSDF* was to create one, that is capable to



(a) Disney/Principled BRDF (b) Disney/Principled BSDF

Figure 2.2: Mixture of different BRDFs, describing the *Disney BRDF* (a) and its extension, the *Disney BSDF* (b).

describe many diverse materials and also the majority of all realistic materials well. To make the artists work easier, an attempt was made, to make the control parameters intuitive and to reduce their number as much as possible. Because of this, all parameters are in a range of $[0, .., 1]$ for realistic materials, but can be in- or decreased further. However, some would simply be clipped then. For parameters in this range the BSDF is *physically based*, conforming approximately to the previously stated laws. The parameters that we finally seek to infer are listed in the first column of table 2.2. All remaining parameters of the *Principled BSDF* are listed in the other columns. For the creation of our dataset, we set the values of them to reasonable constants. In the subsequent we provide some more information about the listed parameters [4] and discuss why we did use them or not.

Inferred Parameters	Constant Parameters		
metallic	subsurface-radius	subsurface-color	normal
transmission	specular	specular-tint	IOR
subsurface	anisotropic	anisotropic-rotation	tangent
roughness	sheen	sheen-tint	emission
basecolor	clearcoat	clearcoat-roughness	clearcoat-normal
	alpha		

Table 2.2: Parameters of the BLENDER implementation of the *Principled BSDF* [4]. The method we describe does only infer the visually most important ones (regarding to our own perception).

The first parameter is *metallic*, this one is used to control the blend between the *metallic* and the other sub-BRDFs. One thing that is special about metals (or more physically accurate conductors) is, that reflections on them are tinted in their color. This is not the case for any other materials. We seek our method to have the ability to infer both metallic and non-metallic materials. For pure metals the *metallic* value is one and for non-metallic materials the parameter is zero. We formulate the training of our network as regression problem (and therefore also allow predictions in $[0, .., 1]$), but for our datasets we stick to pure metals or non-metals. This means that the value is either set to zero or one.

Similarly *transmission* is used for the blend between the specular BRDF and the dielectric BSDF. Again we seek to have the ability to infer both types of material and therefore use the parameter as well. The effects of both (and the following) parameters are visualized in Figure 2.3.

Subsurface scattering is the effect, that a material illuminates itself from the inside. This is caused by light, that passed into the materials surface and because of internal deflections passed back out again at a surface location close to the entering point. Materials with this property usually have a soft look, for example skin or marble. The parameters *Subsurface* and *subsurface-radius* are used together, to control how much *subsurface scattering* happens in the material. The *Subsurface* variable is used to scale the three dimensional *subsurface-radius*. More, the *subsurface-radii* control the directions of the *subsurface scattering*. For simplicity we set the *subsurface-radius* to constant $(1, 1, 1)$ and only vary *subsurface*. Note, that the previous parameters (*metallic*, *transmission*, *subsurface*) need to be interpreted with caution. That is, because the precedence of the mixtures (as shown in Figure 2.2b) leads to overriding some parameters by others. For example pure metals are non-transmissive to light. This means, that $metallic = 1$ implies no *transmission* and no *subsurface* in the rendered image, even if these parameters are set to values larger than zero. Likewise a rendered transmissive surface contains less *subsurface scattering*, than the value of the *subsurface* parameter makes it seem to. We will therefore use adjusted values of said parameters, as ground-truth variables for learning.

The next parameter is *roughness*. How a material is perceived depends very much on it. It describes, how strong the reflected light is spread. A low value manifests itself in a mirror-like appearance, whereas a high value makes the material look matte. This parameter is of special interest, because it notably influences the appearance of a material, while at the same time it is difficult to guess it properly.

Basecolor, as the name suggests, describes the inherent color of an object. For mostly opaque materials, this parameter has a similarly large impact on rendering as *roughness*, so we use this parameter as well. However, unlike for *roughness*, it is much easier to estimate this parameter for most materials in a sufficiently well-lit environment. This is, because for most materials *basecolor* is directly observable in an image, while *roughness* is not.

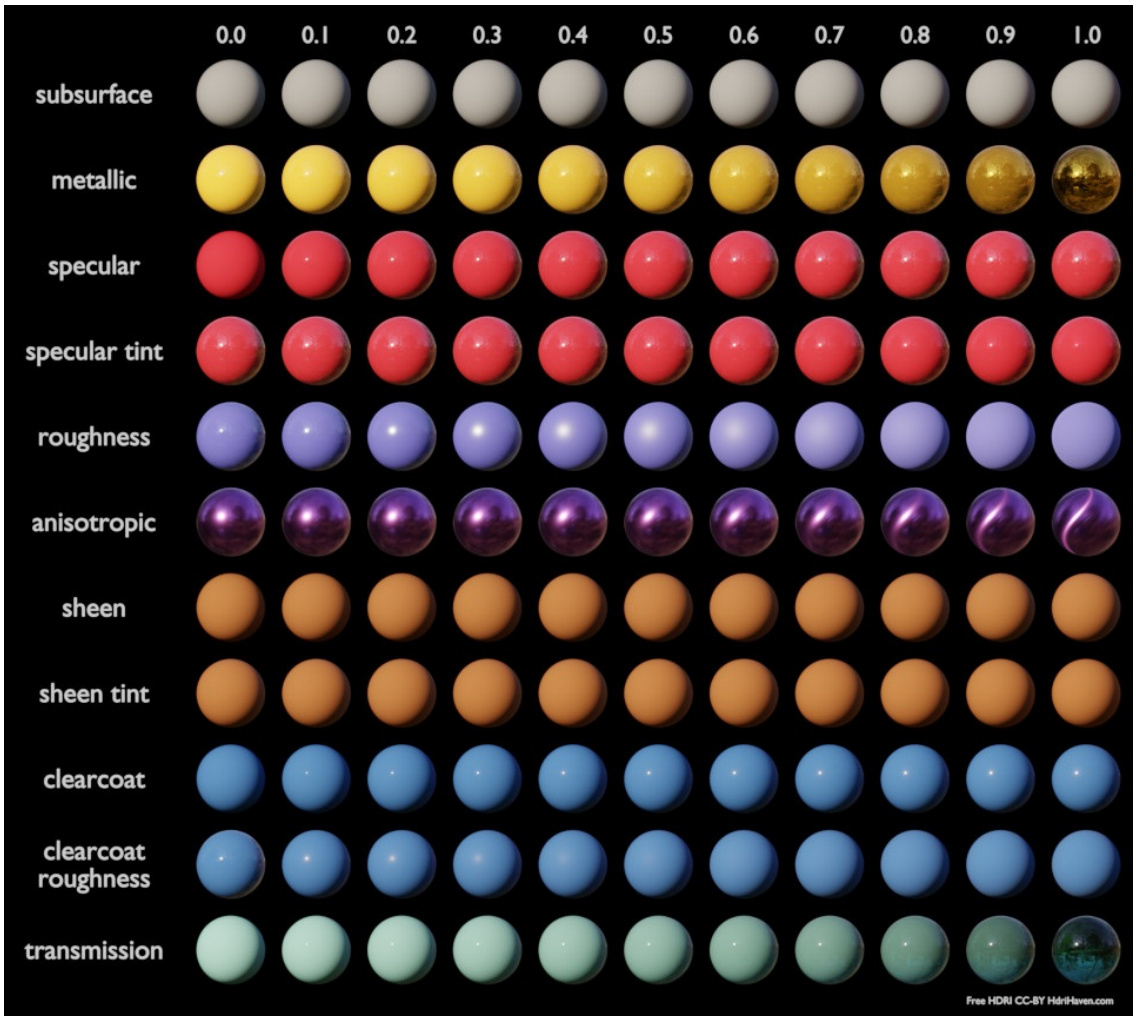


Figure 2.3: Effects of different choices for the *Principled BSDF* parameters. Taken from [4]. The image is rendered with *GGX* microfacet distribution and originally has another row *transmission roughness*, that we removed. For our purposes we always use *Multiple-scattering GGX* (more physically accurate), which does not contain this parameter. For visualization purposes the differences (of the two microfacet distribution functions) on the renderings of the other parameters are negligible.

So far these are all the parameters that we seek to predict with our architecture. However, the *Principled BSDF* is defined by some more. A short explanation of each and why we do not use them below.

Like for *subsurface-radius*, we are also not interested to infer the *subsurface color*. This parameter defines the color of the material below its surface, and changes the tint of scattered light. However, this parameter is rather meant for artistic control. Internally in the BSDF it is linearly combined with the *basecolor* (by *subsurface*) to the *mixed subsurface basecolor*. This means, that instead of the *basecolor* and the *subsurface color*, the *mixed subsurface basecolor* could be used for both parameters in rendering and the result would look the same. Moreover, in an image of an object with high *subsurface*, the *basecolor* would not be observable. The *subsurface color* alone fine the tint of the object. But, if the *subsurface* value was low, it would be the other way – *basecolor* would define the appearance of the object. For this reason we do use *subsurface color*

in rendering our dataset. Yet, we do not actually infer it and neither do we predict *basecolor* as stated before, but only the *mixed subsurface basecolor* instead.

IOR is a physical property and describes refraction in transparent objects and also the strength of the *fresnel*-effect (stronger reflections at grazing angles). For most materials *IOR* is approximately 1.45, we therefore use this value as a constant. Some few exceptions are vacuum (1.0), air (1.0003) and diamond (2.418) [1].

For dielectric materials *specular* describes the strength (weighted by 0.08) of the specular reflections (with respect to the amount of light passing trough the objects surface). It can be computed from the *IOR* as well and for the same reason as before we set it to constant 0.5. The *specular tint* defines how much the color of specular reflections is shifted towards the *basecolor*. Such a phenomenon is not physically accurate. We therefore set it to constant zero.

Anisotropy in the context of material properties describes how the uniformness of reflection in all surface directions differs. Isotropy (i.e. low anisotropy) means that the reflection is uniform, whereas high anisotropy means, that reflections happens mostly in one direction only. This effect is controlled with the BSDF parameters *anisotropy* (amount), *anisotropic-rotation* (direction) and the surface *tangent* (direction of zero rotation). This effect can be visually notable in some materials, but in favor of simplicity we do not seek to infer it. Thus, all values are set to zero.

The visual effects of *sheen* and its tint are much more subtle. They are used to add possibly colored reflections at grazing angles. The effect is somewhat similar to *fresnel*, but is not physically motivated. The original design choice for *sheen* resulted from observations of measured materials. Fabric samples showed stronger than usual and also tinted fresnel reflectance. Both parameters were introduced to mimic these observations. The authors assumed that the stronger reflections come from the fact that fabrics usually have many tiny transmissive fibers that cause this effect. In addition, the introduction of transparent materials [10], causes energy losses at the edges of an Object. *Sheen* can also be used to eliminate these errors and make the renderings seem more physically plausible. However, as already mentioned, these effects are very subtle and do not influence renderings in such extend, that we would seek to compute them. Again for simplicity we set *sheen* to zero. Therefore, the value of *sheen-tint*, that describes the tint towards the *basecolor*, does not matter and we set it to zero as well.

Since many real-world objects have a clear coat on top of its actual surface, the *Principled BSDF* allows to approximate it within each material definition as well. This is computationally cheaper, than mixing a BSDF for the actual material and an additional one to explicitly model a clear coat layer. For our purposes, however, this adds too much complexity and we set the values of *clearcoat*, *clearcoat-roughness* and *clearcoat-normal* to zero.

The *alpha* value in the BLENDER implementation is not an actual BSDF value, but rather an after the rendering applied transparency. This has nothing to do with our problem and we therefore set the value constantly to one.

Likewise *emission* is constantly set to zero, because it does not relate to our problem. It describes an additional light emitting property of the material. We are not interested in this.

The last remaining parameter is the surface-*normal*. This is a geometry information we do not seek to estimate. We rather take as input to our method as well. For our dataset this value is varied, but it is also provided to our network at inference time.

2.2 General Problem Description of BRDF Estimation

In general, the BRDF/BSDF estimation problem can be described as the task of deriving visual material properties from inputs, that are very limited in the number of sampled light angles. Usually one or more photographs under different illuminations are used for the inference of the sought parameters. Further, these material properties should enable a re-rendering of the captured scene that is visually perceived to be as close as possible to the original. This is desired not only for the original view and lighting conditions, but explicitly for novel ones as well. This means that after successful material estimation, a captured scene (with known geometry) can be rendered sufficiently correct from any viewing direction and under any lighting condition. The usual way to achieve this is to first choose a BRDF/BSDF and then develop a method to derive the appropriate BRDF/BSDF parameters. The choice in the BRDF/BSDF already constrains how good, at best, the algorithm can be for this task. This is because (as described in section 2.1) BSDFs vary in how well they describe different materials. Moreover, the BSDF that is chosen also defines how challenging the task of deriving its parameters is, due to the different complexities of these functions.

2.3 Existing Approaches

The intuitive approach to BRDF/BSDF estimation is to measure the materials of interest accurately and densely from different viewing and lighting angles. These measurements can be used to recreate the appearance of an object. Alternatively, the parameters of an arbitrary BSDF can be fitted to the measurements, to obtain a more convenient representation of the material. Roughly spoken, this is what the type of hardware-software solution called gonioreflectometer does. This approach is the gold standard for BSDF estimation in terms of measurement quality. However, this process is time consuming in both the data acquisition and the parameter fitting. For many tasks less accurate material estimations are sufficient, especially if the quality decrease could be traded for an improvement in runtime of the overall procedure.

In order to solve BRDF/BSDF estimation many different approaches exist. Variations in the methods are often caused by the intended use-cases. Predominantly due to the trade-off between quality of recreating a scene, runtime (of training and inference) and complexity of capturing the inputs. The quality of the results is directly visible by comparing re-renderings to the captures that were used for inputs. As stated before, the choice of the BSDF can have a big impact on the results. Further, the inferred (or skipped) parameters of it play an important role as well. They define how complex effects can be at most, that are recreated by the estimated materials (for example *fresnel*, *anisotropy*, *transparency*, *subsurface scattering*, ...). Another common difference, that relates to the quality of the predicted material is the decision on whether to support spatially varying materials or not. On the one hand, if uniform materials are demanded, the task is usually easier and allows for better estimates, because inputs provide more samples (of the same material). Spatially varying estimators, on the other hand, are able to infer multiple materials at the same time. This makes them suitable for re-creating the visual properties of a far broader range of real-world objects. However, a common limitation in the spatially varying case is, that especially the newer learning based methods [14, 15] only allow small output sizes. Besides the differences in quality of the results, the BSDF estimation techniques also differ in runtime. For learning based approaches a training phase is necessary and even more this usually includes the creation of a dataset. The required runtime for this task usually consumes multiple weeks, up to months. Yet, this is a preprocessing step to the actual algorithm and only has to be done once. For this reason it is to some extend neglectable. More important is the runtime required for inference. In this, the different methods vary a lot. As a rule of thumb, gonioreflectometers, that are the gold standard with respect to prediction-quality, take the longest – up to several hours. They are followed by mathematical models [24, 6], that work within the range of minutes to hours. Usually, the fastest are recent learning based approaches [14, 15], that compute the results almost instantly. However, also the intended quality plays a role in runtime. For example the learning based method [21],

that works with gonioreflectometer data as input, takes 15 minutes per pixel. Another substantial distinction in the different approaches to BSDF estimation is the complexity of the acquisition method for the input data. Furthermore, the implied acquisition time should not be neglected. It is usually defined by the number of photos used as input. Alternatively, there are also methods that work with short videos [24] (30 seconds). The last major difference in the capture complexity is the one between lightweight captures versus calibrated measurements. The first means that the BSDF estimation technique works with uncalibrated cameras and unknown light sources. The latter means that every aspect that contributes to the captured images is known precisely and moreover controllable (i.e. all camera parameters, light positions and intensities and the ambient illumination). The latter refers to the capture setup as available in gonioreflectometers. Lightweight captures are the opposing extreme. BRDF estimation techniques usually choose a capturing method in between. Further, it is very common to make assumptions about the input, so that it is possible to use more lightweight captures, while also relying on implicit knowledge of the scene. Examples are approximately defined light and camera placements [24, 6, 28]. Also constraints on the geometry of the object are very common. Often a flat tile is required [28] and in the spatially varying case only slight variations of the surface normals are estimated [14, 15]. In the following we give a short overview of a few already existing techniques. Afterwards in Chapter 2.4 we define our specific use-case and elaborate on how we solve it in Chapter 3.

2.3.1 Pocket Reflectometry

This paper [24] works with a handcrafted mathematical model to fit measurements of a material to representatives of known BRDF samples (Ashikmin-Shirley model [7]). This means the method relies on a BRDF chart that had been measured with a gonioreflectometer in advance (similar to a color checker). For data acquisition a single camera is used to capture a video of the BRDF reference chart together with the (approximately flat) object of interest. During the recording of about 30 seconds, a tube light is moved by hand over the object. The recording is then used to estimate the parameters of the microfacet BRDF model for the captured object. The estimate is done per pixel for arbitrary sized inputs. It is created as a weighted sum of the Lambertian and specular parts of the BRDFs from the previously measured exemplars. This fitting works with the observed specular highlights. Due to different spatial locations of the sampled material and the reference BRDF tiles in the scene, corresponding specular highlights (in terms of surface normals) caused by the moving light occur in different time frames. Therefore, prior to fitting, a pixel-dependent time-warping of the video is necessary, so that the corresponding highlights occur at the same time. This is computed with an adapted version of the *dynamic time warping algorithm*. Further the paper proposes an extension to compute varying normals for slightly bumpy surfaces. This requires further *spherical cap references* (material samples on a small dome, for known surface normal variation) and an additional light movement.

2.3.2 Reflectance Modeling by Neural Texture Synthesis

Inspired by texture synthesis, this approach [6] aims to compute a 256×256 seamlessly tileable SVBRDF patch from an arbitrary sized headlight illuminated image. The idea is to learn the SVBRDF parameters, by describing the problem as a computational graph and optimizing the difference of the input image to a render (form the choosen parameters) at inference time. For rendering the *Blinn* reflectance model is used. Finding a good minimum for the render/ground-truth difference in the BRDF parameter-space is a rather difficult problem and only works due to a clever way of representing the data, some priors and little constraints. These priors are for promoting reflectance parameters, that are stationary. This means, that statistical features of any region should also occur in others. Next to the priors, the architecture relies on an good internal parameterization, to ease the BRDF estimation. One aspect is the representation of the surface normals as height map convolved with finite difference kernels, followed by normalization. Another aspect is representing specular albedo in the YUV color space (by defining the RGB value as $\rho_s YUV2RGB(1, \rho_s U, \rho_s V)$) and constraining the chromaticity $(\rho_s U, \rho_s V)$ to be constant

(still learnable) over the whole image. $YUV2RGB$ is meant to be the transformation between the color spaces. For the overall specular albedo only one scalar ρ_s is learned per pixel and multiplied after converting back to RGB space. Moreover, the authors state that transforming the optimization variables in the Fourier domain improves fitting them notably. To overcome the problem of missing information about the exact light conditions, the authors extract multiple patches around the specular center and treat them as differently lit samples of the same material, to compute a spatially constant preconditioning. Since a single BRDF precondition is computed from N-many different patches, it is not possible to learn using a pixel based loss. Instead the comparison is driven by textural features. To do so, a texture descriptor TG is employed (obtained from the weighted Gram matrices of the first 17 layers of a pretrained VGG network without zero padding, to avoid border artifacts). Using circular shifting as an augmentation strategy helps in addition to produce seamlessly tilable SVBRDF maps.

2.3.3 Flexible SVBRDF Capture with a Multi-Image Deep Network

This publication [15] extends the paper **Single-image SVBRDF Capture with a Rendering-aware Deep Network** [14]. The approach [14] aims to predict the Cook-Torrance SVBRDF parameters from a flash lit photograph of an even surface. Due to constraints in memory and convergence of training, the output and input size is limited to 256×256 pixels. The architecture idea is, to use a standard U-Net to learn the translation from a flash-lit image to the SVBRDF maps. However, the authors argue that a U-Net alone is not perfectly suited for the task, because it generally struggles to fuse distant visual information. Therefore, the authors propose an additional fully connected (FC) network, solely for learning the global features. This global feature network has one FC layer for each conv-layer in the U-Net. It constantly interacts with the U-Net. To train the network, the authors rely on artist-picked BRDFs that cover a wide range of materials. From these a generated dataset of 200000 rendered BRDFs was created by applying additional augmentations. For training they refrained from using the common L1 loss and proposed a new rendering loss. It works as follows: During training the loss is computed as a function of the average error[1] between the ground truth image and the renderings of the SVBRDF under all lighting and camera angles. This loss is computed stochastically by randomly selecting some light and camera configurations. The idea of the extension [15] is to harness the information of a variable number of images of different uncalibrated camera light configurations and with it to improve the quality of the BRDF estimation. Yet, also the loss computation and the dataset creation were adjusted. To utilize multiple images, the architecture had to be changed. The core of the new architecture is the same as before (U-Net + global-feature-track). To cope with a varying number of inputs, this core-network is dynamically spawned for each of the input images with the same weights. This means, that for each image feature-maps are computed and for each the global track is evaluated individually. To combine the individual information, the features of the U-Net and the global-track are max-pooled. These two individually pooled paths are continued in the same manner as the original architecture. To let this network learn more stable than the original one, the authors additionally adjusted the loss. In detail they added explicit losses for each of the estimated BRDF parameters to the rendering-loss. Another change to the original training procedure is the creation of the training data. These are still renderings of linear combinations of artist designed BRDFs. However, contrary to before, the rendering and various data augmentations are done while training. This approach adds computational cost to the training procedure, but is nullified by the time saved from the avoided loading and transfering of a training-set. Moreover, precomputational efforts and hard-drive space are saved due to this method. For inference the capturing is done as before, but the camera and light locations are not assumed to be overhead anymore. Rather they are assumed to vary individually. This way the net can actually gain new material information with each additional input image.

[1] *log* of L1, to cope with specular highlights.

2.3.4 BRDF Estimation of Complex Materials with Nested Learning

This approach [28] estimates the BRDF parameters (9/25 of *Maxwell Renderer BSDF*) for a tile of a uniform material. To do so it relies on a light source placed above the tile and a color checker orthogonally placed next to it. The material is captured twice. Once with an angle of 45 degrees and a second time from overhead. The tiles in each image are then mapped to 32×32 images. Concatenated they serve as the input for the proposed neural network. The original idea for the network were two identical two-layer CNNs run in parallel followed by some FC layers, which predict the nine BRDF parameters. One of the CNNs uses the input as described above, the other works with a whitened version of the same image. This was done to remove color correlations and in turn ease the learning of roughness and IOR. However, the FC-layers were replaced by a handcrafted nested FC network. This network explicitly models the dependencies of the BRDF parameters amongst each other. This was motivated by the observation that the initial net had to learn these dependencies and struggled with the task. To ease the training even further, the authors reformulated the regression problem as classification, by allowing only 100 different values for each of the BRDF parameters. The cross-entropy-loss was adjusted accordingly by preceeding a Gaussian smoothing of the one-hot-encoded outputs. In that way, predictions that are similar to the ground truth lead to smaller losses than vastly different ones. This means that the loss maintains the spirit of a regression task. Another distinctiveness of this publication is, that it works with the CIELAB color space instead of RGB. Contrary to RGB, CIELAB was designed to linearly relate parameter adjustments to the color-difference perceived by a human. Therefore, the network optimizes a loss more closely related to human vision. For training a new synthetically generated dataset was used. Novel views (camera, light angles and material-tile rotations) where used for testing and to show the generalization of the model.

2.3.5 Learned Fitting of Spatially Varying BRDFs

The main goal of this publication [21] was the computation of high quality BRDF estimates, with the acceptance of involved data-acquisition. The method is a compromise between the very costly but accurate fitting of a gonioreflectometer and the few-shot BRDF estimation techniques, which are less accurate but much faster inference and acquisition. For inference the architecture relies on scan-data and photographs of various light camera conditions, taken from the captured data of a gonioreflectometer. For training purposes only, the material estimates of the gonioreflectometer were used as well. The architecture is a four-layer CNN followed by a single FC layer that computes the 13 parameters (encoded in 14) of a modified Ward BRDF model (that can handle anisotropy and fresnel). For input the data is simply stacked into 934 layers. The first two conv-layers are 1×1 and are used to subsequently reduce the huge number of feature channels into 256 new ones. The next two convlayers are 3×3 filters and are employed for a spatial reduction. The resulting features serve as input for the final FC layer, which then computes the 14 BRDF parameters. What was described so far is used to infer the BRDF parameters of a single pixel from a 15×15 patch of the input. To compute a whole SVBRDF map, a sliding window approach is used. This allows arbitrary height and width of the input. Similar to [2] the authors noted, that batch normalization hinders learning (as it removes mean intensity). For this reason they leave it out. Evaluation showed better BRDF fits than the single-shot technique from [2], at the cost of an increase in acquisition duration and runtime. At the same time the quality is comparable to the incorporated gonioreflectometer, but with a notable decrease in fitting time.

2.4 Problem Description of the Proposed Method

In the previous chapter, different approaches to material estimation where shown. Most of them worked with implicitly known, flat objects. The overall goal of our work is, to create realistically looking re-renderings for arbitrary 3D scanning data. For achieving this, we reduce our problem to the simpler one of deriving BSDF parameters from the scan data. These parameters can then be used in conjunction with the particular BSDF and geometry (known from the scan) to re-render the captured scene. We explicitly require the method to not be restricted to flat objects. This is a difference to most other approaches, that usually work with approximately two-dimensional surfaces. In our case this means that the observed object can shadow itself and that some surface points are not observable from certain views. In detail we seek to learn the prediction of the most

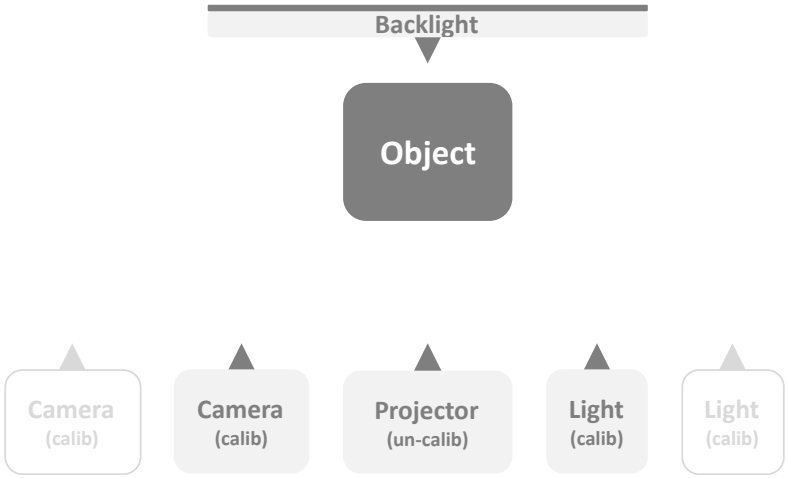


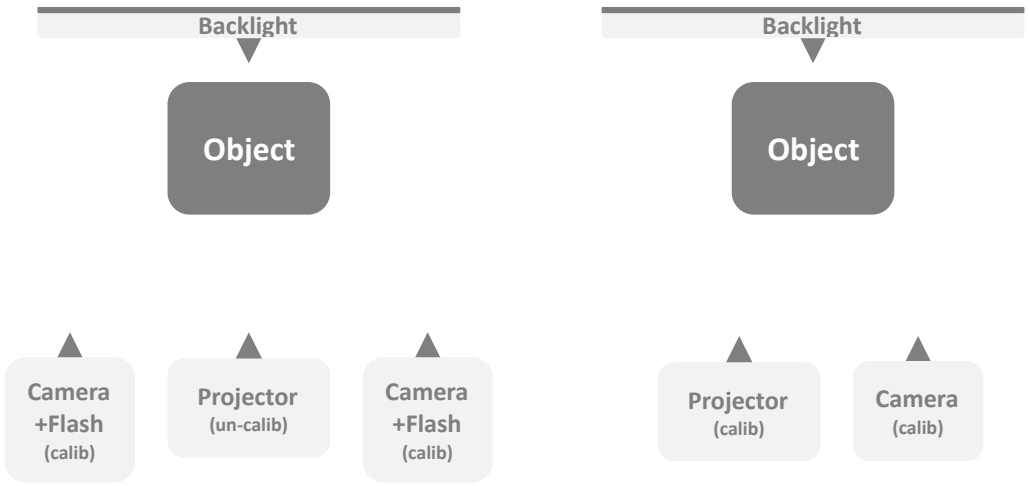
Figure 2.4: The general scanner setup for our method. Calibrated cameras and lights are placed freely in front of the scanned object. The backlight is behind. At least one camera is required, but more can be added without the need of any further changes. Likewise we require at least one light, but more can be added as well, to improve the results. In addition we also need one projector placed in a way, that its image overlaps in a preferably large area with the scan-object and with all cameras' receptive fields.

relevant parameters for the *Principled BSDF* function (also known as *Disney BSDF*), for the reasons listed in Chapter 2.1. We are interested in *metallic*, *transmission*, *subsurface*, *roughness*, *basecolor* and *subsurfacecolor*, as discussed in Chapter 2.1.1. However, we combine *basecolor* and *subsurfacecolor* as *mixed subsurface basecolor* like it is done in the shader internally anyways [3]. For re-rendering, this value can be plugged into *basecolor* and *subsurface color* with no difference to the actual values. For convenience we only talk of *basecolor* in the following, but actually mean the combination of both with it. We seek to achieve the BSDF estimation with a preferably flexible method, that is suited for slightly varying capture setups. It should allow to harness the information of multiple views of a scan if available, but at the same time it should be able to work with only a few views, if not more are known. In general we require one or more calibrated lights and one or more cameras, that can observe the scanned object (and have a known correspondence between the image pixels and the 3D geometry). Moreover, we need a diffuse-white-reference object (for example a sheet of paper), as a means to make a guess of the ambient illumination in terms of color and intensity. In addition we also need a projector if *subsurface* is of interest. If not, the whole procedure would also work without it. Similarly, for *transmission* only, we require a backlight that spans the entire background of the scanned object. This backlight could also serve as the diffuse-white-reference. Without any exceptions we also use the backlight for this purpose in the subsequent. The described setup is also depicted in Figure 2.4. The lightsources, camera(s)

and the projector can be placed freely in front of the object, but their location and orientation needs to be known. The backlight is placed on the opposite side. It is meant to be parallel to the average baseline of all cameras, but this is no hard constraint and placement can be done relatively free. Also we do not require position information of it. The goal of our method is now to use the data, that is recorded with the described setup to infer the mentioned BSDF parameters. In terms of data-acquisition our method can be classified as a rather complex one, but it is still far more flexible and a bit more lightweight in capturing than the goniorefelctometer approach, as described in [21] and summarized in Chapter 2.3.5. For a concise summary, refer to Figure 1 again. It already contains BSDF parameter predictions made with the method we describe in Chapter 3.2 – the outputs of *subsurface* and *transmission* are scaled by two for better visibility.

Real-World Application In a real world scenario our general setup could be directly used. However, some configurations might be more convenient than others. We like to point out two setups, that are particularly interesting due to their widespread use. One setup that fits well for our method is *structured light* (see Figure 2.5b). This setup only needs to be extended by a backlight (or a diffuse-white-reference if *transmissoin* is not of interest) to be usable with our method. Despite no distinct controllable lights in this configuration, the inference can be done nonetheless, by using the projector also as a light source (by showing a uniform white image). Another setup, that can be used for our method is *active stereo*[2] or *passive stereo*[3] without a projector, if the subsurface values are known beforehand.

Especially when implemented with mobile-phones, this setup works well. This is, because in this case we still get by with the backlight as the only additional tool, and that is even only required when we are interested in the *transmission*. For the lights, we can use the phones' flash in this approach. Because of their proximity to the cameras, we can approximate the light locations using the known calibration of the cameras. This setup is shown in Figure 2.5a.



(a) *active stereo* (b) *structured light*

Figure 2.5: Two common scanner-setups, that can be used for our method of inferring BSDF parameters. Both systems could be extended (but do not have to be) with further cameras and or lights to improve the predictions of our method.

[2] **Active-stereo** systems derive 3D geometry from several cameras. Finding correspondences between the cameras is a subtask of it and the more found the denser the reconstructed point-cloud is. In these systems a projector is used to guide the correspondence-finding step and allows for dense reconstructions.

[3] **Passive stereo** systems are such, that derive 3D geometry only from several cameras. The reconstruction density is usually not that high, compared to active approaches.

Chapter 3

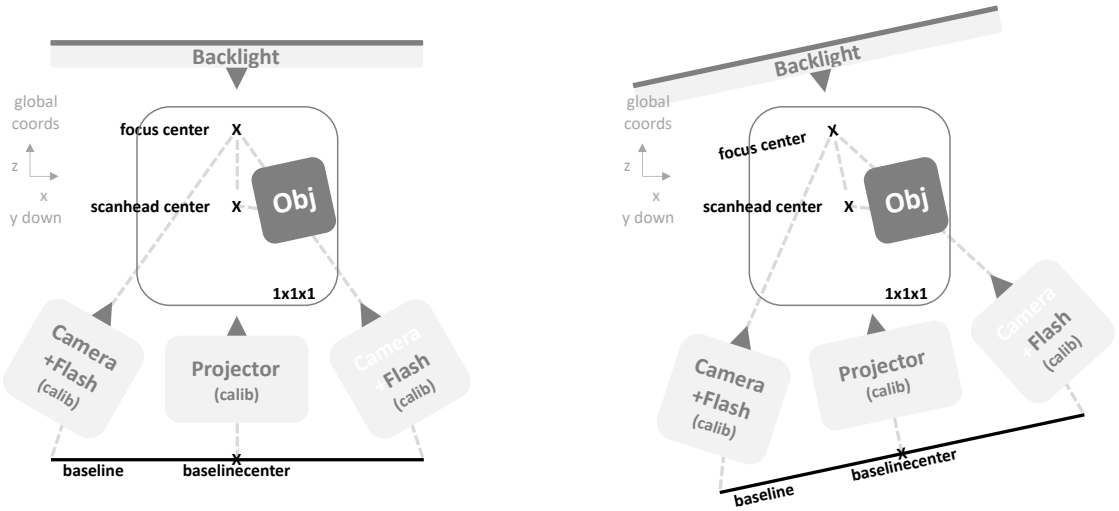
Method

The previously defined problem of predicting BSDF parameters from 3D scanning data will be solved in the following by a machine learning approach in chapter 3.2. We will train the described network in a supervised manner. Therefore, an important preceding step is the generation of training data – pairs of inputs and ground truth predictions. To do this, we create an artificial dataset in the manner being described in 3.1 below, before specifying the details of our architecture.

3.1 Dataset Creation

Our method (described in Chapter 3.2) will be a supervised learning approach. For this it is required to have some input data X and the according labels Y. Y is also called the ground-truth (GT) data and describes the desired outputs of the the learned function, given X. In our case Y are the BRDF parameters that can be used to re-render the inputs. One way to gather X, Y pairs would be, to compute them with a gonioreflectometer or to label real world data manually. However, both is time consuming and the latter is prone to errors as well. In general neither is feasible. Another approach to collect X, Y pairs is to create them synthetically. Again, different strategies are possible. For example, the creation process could be guided by human input or rather be completely randomized. We go into details later. Another difference in the dataset creation (especially in this domain) is to choose either a preprocessing to create X and Y before learning (as we do) or to create them on demand. The latter has the benefit, that creation of the input data X itself can be modeled as a part of the network based on Y. Moreover the predicted BSDF parameters $\hat{Y}$ could be used to compute a prediction of the input $\hat{X}$. This allows to define and optimize a loss based on the difference of X and $\hat{X}$ (as similarly done in [6, 14] – refer to Chapter 2.3). This means minimization directly relies on the perceived difference of input images compared to re-renderings instead of the BRDF parameters. This can be beneficial, as the BRDF parameters are merely a tool to produce visually similar re-renderings compared to the inputs. However, we do not use this strategy – firstly because of ease of implementation and secondly to make reuse of the dataset cheaper. Having a precomputed dataset is helpful for experimenting and moreover, we finally want to train five different networks each for one of the individual BRDF parameters. All of these nets overlap in much of the data contained in their respective X, Y pairs. For our approach it is therefore greatly beneficial to work with a precomputed dataset.

In the following Section 3.1.1 we first describe which variant of our general scanner-setup (Figure 2.4) we use for computing the dataset. Next in Section 3.1.2 we describe which data we exactly require later for our BRDF parameter-inference method and therefore also render from the selected scanner setup. Moreover, we already discuss how we can bring randomness in the renderings – an important step, to avoid overfitting in training later. We discuss this in two sections. How to modify the geometry of the scanned object is explained in Section 3.1.2. How we randomize the materials is illustrated in Section 3.1.3.



(a) scanner setup

(b) scanner setup rotated

Figure 3.1: (a) The scanner setup used for creation of the dataset is a union of the parts that are required for using our method with *structured light*or *active stereo* systems. (b) However, different cameras are modeled by re-using only one and rotating the scan-head around the scanned object. For convenience the object stays within the $1 \times 1 \times 1$ box around the world origin, which is also the location of the scanhead-center.

3.1.1 Scanner Setup

To create the synthetic in- X and outputs Y for training our network, we rebuilt our scanner setup in BLENDER [2] and use it for rendering said data. Since, we intend our method to work especially well with *structured light* and *active stereo* (implemented with mobile phones), we use a union of both systems for our dataset creation. However, as we later like to train with inputs from the object of multiple positions anyways, we only model and render with one camera. This is not really a difference to the initially described system, as the various positions can also be interpreted as further cameras. In that sense the only constraint is, that all cameras have identical intrinsic parameters. This is also a viable approximation for real cameras of the same type, given they are focused on the same distance. Working with this assumption allows to avoid the rendering of the scene from the second camera. This saves a lot of computational effort. However, we also had the intention of using the second camera (/phone) as a light source with known location. We do not skip modeling the light in our synthetic setup. We avoid this, because the first light only is capable of illuminating the object from the exact same direction as the camera observes the scene. The second light however helps to produce renderings with varying light angles. As described in Chapter 2.1 the BRDF also depends on these and training on a well distributed set of light angles will therefore be beneficial for the overall result. Note that a slight variation of the observed light-angles is already caused by different geometries of the scan object. Nonetheless, we use the second light as a means to bring in more variety. It would also be possible to use the projector for achieving this and we also use it in the dataset-creation-process as a light source. Anyway, we use both the projector and the second light, because we also model the light colors according to the real-world-sytems they are referring to. Both, the flash at the first and the flash at the second camera are usually slightly yellow tinted, if mobile phones are used as light sources. We mimic this by setting the according rendered lights to HSV= $_{\square}$ $0.1666, 0.1, 1$. The projector will be rendered as a clean white light, approximately what can be observed from an actual projector. Finally, we could render photos with our camera of the scene, illuminated by yellowish light next to the camera. Alternatively, we can also take a photo with a clean white light from the projector

or again with a yellowish light, from some other (known) location. Moreover, we could also take a picture with all lights turned off – a static ambient illumination then being the only light source for the object. Also we could rotate our scan-head (camera+projector+lights) around the world origin[1]. This allows to take further pictures of the scene – again with different illumination. In an *active stereo* system we could in addition to rotation, also take the pictures from the additional cameras. This is similar to the rotation of the first camera. The only difference is that there the second light always happens to lie on the other camera, but this does not change anything in the subsequent. For most of the datasets we used three random rotations that are relatively close to each other, so that the rendered images have some overlap in the depicted scene. For some datasets we also rendered a whole pass of stepwise rotations and tilts around the object like scanning on a turn-table would do. This however is quite computationally intensive and produces a lot of data depicting the same object in terms of geometry and materials. To have more randomness (for the same computational effort) in our dataset we abandoned this rendering scheme early and rather adhered to the three-position-method mentioned first. An example of a scene rendered from three positions can be seen in Figure 3.2. To summarize, we create a synthetic dataset and therefore we know calibration of everything by creation. Due to this, we get away with rendering less images than an actual implementation of a *structured light* or stereo-system would require to compute geometry information (which is an input to our method). To create the synthetic dataset we only use one camera, a projector and two point-lights – placed where the first camera is and the second would be if we would use one. This is depicted in Figure 3.1.

(a) Position 0 (b) Position 1 (c) Position 2

Figure 3.2: A sence rendered from three different positions. Position 0 is rendered with the same setup as in Figure 3.1a. The other positions are two random rotations like in Figure 3.1b. All lights are turned off. The perceived illumination differences amongst the different positions (especially on the backlight) are caused solely by the sourrounding environment.

3.1.2 What is Rendered

To create our dataset, we use the *path tracing* engine CYCLES (provided by BLENDER [2]) and the scanner setup we described previously. With this we build multiple datasets, but with a big overlap in their creation process. We point out other variants is Section 3.1.7 and begin with explaining the general approach. For each dataset we create multiple different scenes that are usually observed with one camera from three different positions (in some exceptional cases more). These scenes

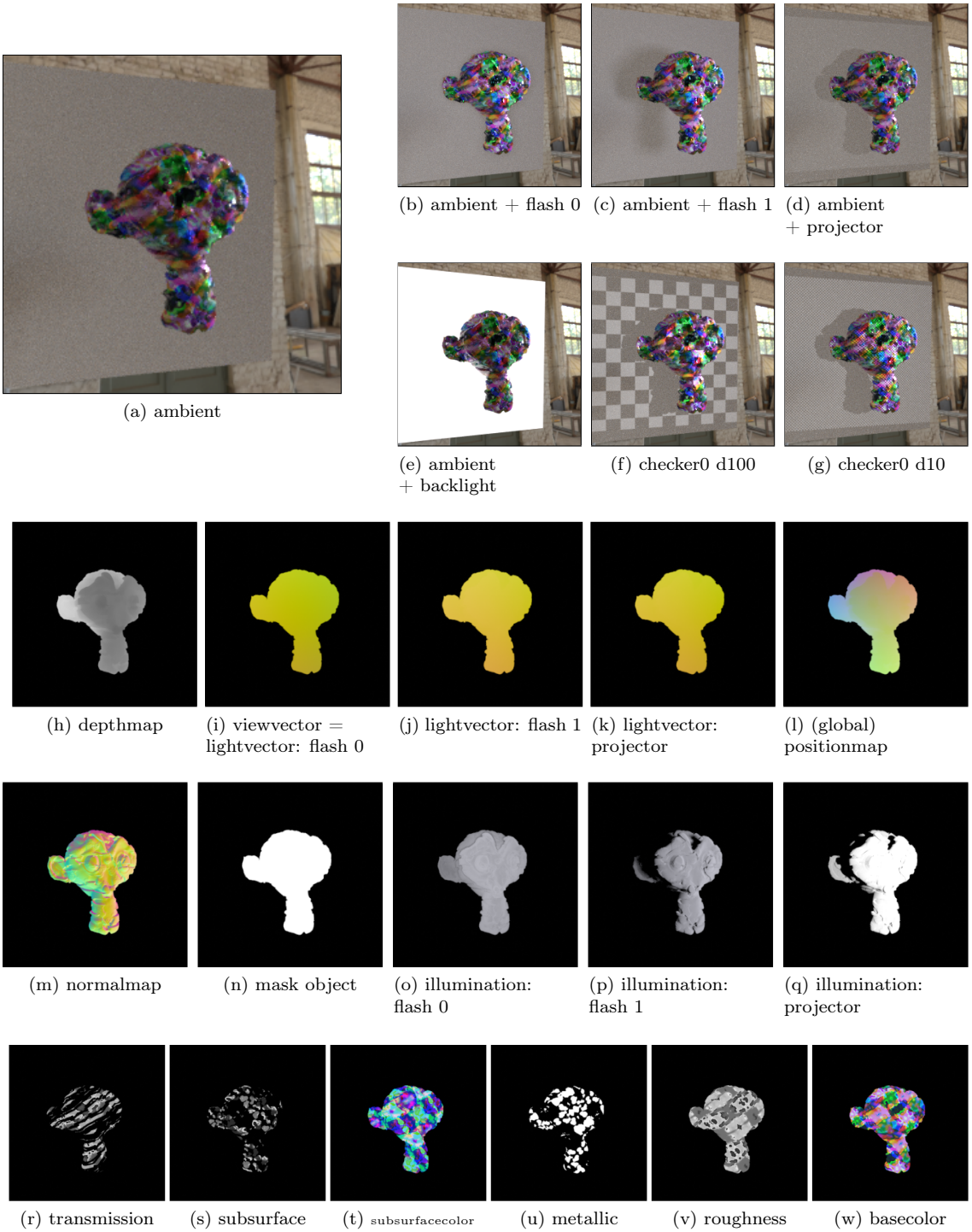
[1] In real-world applications it might be more convenient to rotate the object (e.g. with an automated turn-table) than the scan-head. Note that there is no big difference between the two approaches. Only how reflections of the environment on the object change between different positions varies between both strategies. For example, what lies in the background of the first position can never be part of a reflection, when using the turn-table approach. When using a rotating scan-head, it can very well be content of a reflection. However, this will not be relevant for our approach and for convenience of rendering the positionmap (Chapter 3.1.2) we choose the described strategy.

vary in the scanned-object, the specific settings of the scanner-setup and the surroundings of the scene. So far, we have left out the details on this. In the following, we will make up for that.

Environment-maps In our setup, the surroundings are not explicitly modeled, but rather an environment-map defines the ambient illumination for each incoming direction. For rendering our scenes we randomly picked from 75 preselected environmentmaps. Some of them are shown in Figure 3.3. Using ambient illumination complicates our problem, because the environment introduces new highlights on the scanned object and moreover has an impact on the reflected color. In laboratory settings one could assume a controlled environement, allowing to leave this entire step and expect even better results due to an overall simplified problem. However, using an environment makes the work more practically relevant, as it imitates real-world-scenarios better than for example a constant or no ambient lighting at all would do. We intend to cope with the newly introduced highlights from the environment by using multiple positions (which have different highlights). For an estimation of the average incoming color we use the mean response of the turned-off backlight, which is made of a white diffuse material. We will go into detail later. Like in this section with the environment, we describe the variations in the scanned-object and the specific settings of the scanner-setup in more detail in the following sections. Nonetheless, first we explain what will actually be rendered for each position in one scene of a dataset.

Figure 3.3: Some of the (75 in total) environement maps we used for rendering the scenes – HDR-Images taken from [18]. We picked a mix of in- and outdoor environments with the dominant light sources varying in color and intensity distributions. These images are projected onto a sphere, which then defines the ambient illumination for each incoming light direction.

Renderings with Varying Light-conditions For each position we render various images of the scene. Moreover, we store the ground truth BRDF parmeters that were used to create these images. Examples for all images that we compute with BLENDER are given in Figure 3.4. These images make up the biggest part of our dataset and most of them will be used for training, without any modifications to the rendered. However, we also extend our dataset further in Section 3.1.5 by post-processing and combining some of them. The first render (Figure 3.4a) shows the captured scene only illuminated by the environment, as explained before. The following six images (Figure 3.4b - 3.4g) are created in the same manner, except each with one individual light source in addition. Figure 3.4b - 3.4c show the ambient scene lit in addition with the left and right flash-lights (as defined in Figure 3.1). Similarly Figure 3.4d shows the projector, projecting a spatially constant gray image. Moreover, we render exactly the same configurations again, but this time we replace the original materials with an uniformly diffuse white one and turn off the environment-illumination. This results in the images of Figure 3.4o, 3.4p and 3.4q. They are used as a indication of the local incoming light intensities later. Continuing in the second row of the top right corner of Figure 3.4 we also render the backlight, again with the environment-lighting, as shown in Figure 3.4e.



(a) ambient (b) ambient + flash 0 (c) ambient + flash 1 (d) ambient + projector (e) ambient + backlight (f) checker0 d100 (g) checker0 d10 (h) depthmap (i) viewvector = lightvector: flash 0 (j) lightvector: flash 1 (k) lightvector: projector (l) (global) positionmap (m) normalmap (n) mask object (o) illumination: flash 0 (p) illumination: flash 1 (q) illumination: projector (r) transmission (s) subsurface (t) subsurfacecolor (u) metallic (v) roughness (w) basecolor

Figure 3.4: An example of the images we rendered with BLENDER for each position in each scene in each dataset. The first two top rows and the last three images in the fourth row show renders under varying light conditions. The remaining images in the third and second row show the geometry information we save and the last row shows BRDF parameters that were used to create the renders.

The next two pictures in the second row (Figures 3.4f and 3.4g) show the projector like in Figure 3.4d, but this time projecting a black and white checkerboard of different sizes. The perceived blurryness of the edges are supposed to guide the prediction of *subsurface* values later. The checkerboard with the big squares was originally intended for spatially constant materials. On the example-image it is clearly visible, that the frequency of the checkerboard pattern is far to low for a precise estimation of the material, which changes very often, even within one checkerboard-cell. To make a more precise estimation possible, we also rendered the checkerboard with the smaller squares. However, we keep both in the dataset, so that our net is forced to learn to cope with different sizes automatically. Moreover, this gives more samples to learn from, without computing all the other renderings - like it would be necessary in a new scene.

BRDF parameters (and the uv-map) In addition to the renderings we also save the BRDF parameters, that were used to create these images. As this will be our ground truth data later, we project them into the image-space of the camera (meaning, that they are rendered, just like the images before). The parameters, that are relevant for our training are shown in Figure 3.4r - 3.4w. These are the values (white=1, black=0, RGB as depicted) for the parameters introduced in Chapter 2.1.1. Remember, some values override others like for example a *metallic* material can never be *transparent* or have *subsurface scattering*. Therefore some of the saved BRDF parameters do not describe, what is actually observable in the renderings. However, for now this is not important and we repeat these special cases when it becomes relevant in describing our architecture. The details, on how we defined the BRDF values (which are inputs to the renderings in the previous paragraph), are given in Section 3.1.3.1. To apply these values, a mapping from an input image to the rendered/scanned object is necessary. This is the so called uv-mapping. It is not relevant to know it for training, but for evaluation it is. Therefore, we also render this mapping, by displaying the texture coordinates on the object – but only for the far minority of evaluation-datasets. The rendered uv-mapping or more precise the mapped texture coordinates are not shown in Figure 3.4, but will be explained in Section 3.1.3.1 as well. The remaining *Disney BSDF* parameters are not of interest for us, for the reasons given in Chapter 2.1.1. We set all of them to zero, except for *specular*, which was fixated to the constant value of 0.5 and *IOR* to 1.45. These values approximate most materials reasonable well. For details please refer to Chapter 2.1.1.

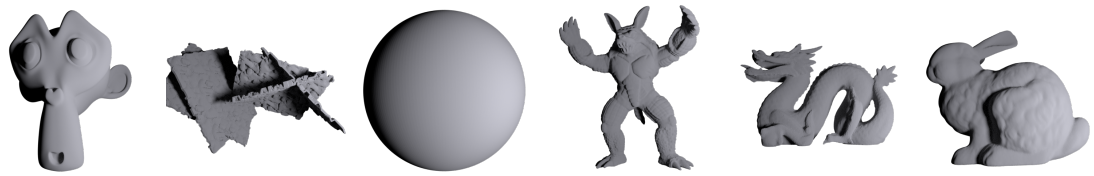
Geometry Information In addition to the renderings and the parameters, that define their appearance, we also save information about the objects geometry (viewed from the camera). The first being stored is the *depthmap* (Figure 3.4h) where distant points are bright and proximate ones are dark. Proximity refers to the $1 \times 1 \times 1$ cube around the global origin as depicted in Figure 3.1. Likewise defined on that cube, is the *positionmap* (Figure 3.4l) it assigns each point in that box an unique color. This means a certain point on an object rendered from different positions, might be in different image-locations, but still has the same color then. We will use this map for creating mappings between the different positions. Further we save the lightvectors for each point that is observable from the camera. To do so, we store the normalized vector, pointing from each object-point to the light as, RGB-triple in the output. We do this for all three lights – the projector and both flashes (see Figures 3.4i, 3.4j and 3.4k). Moreover, we later also require the similarly defined *viewvectors* (pointing towards the observing camera). However, as we defined the first flash to be at the same position as the camera, the *viewvectors* are given with the first *lightvectors* as well (image 3.4i). Similar to the previous we also store the *normals* of each observable surface-point as RGB values. The example of this is provided in Figure 3.4m. Last, we also compute an *object-mask* of the scanned points. This is simply an image of *true*-values (white), where the scanned object lies in the renderings and $false$ (black) everywhere else.

Rendering To summarize the previous: We compute and save for each scene in the dataset the images shown in Figure 3.4. Moreover, we compute all these image for all positions (usually three random ones). An example of this with the *ambient* image (like Figure 3.4a) is given in Figure 3.2. For rendering the different types of the previously introduced images, we adopt the render settings

individually. This is mainly done to save runtime. The main distinction is between the renderings with different illumnination and the remaining. The first ones are computed on GPU, while the second ones are computed on CPU. This is done, since these images are computationally cheap and memory-transport to the device would be the overwhelming part. We made one exception to this. The projector images were also rendered on the CPU. This increased runtime greatly, but seemed unavoidable, because we encountered some problems rendering them on the GPU. Basically, the scene was rendered with the proper illumination of the projector, but the environment lighting failed (only for the projector) on the GPU. To save some runtime and disc-space we skipped rendering some images entirely. In detail, for some reders we have spatially constant BRDF parameters. We do not store them as images, but rather in a text-file and re-create them from it and the *object-mask* on demand again.

3.1.3 Scan-object Randomness

In the previous section we described what kind of images we rendered for a certain scene. However, we did not discuss how this scene actually looks like, besides it being split into background and a scan-object. The background is modeled by an environment-map and provides only variations in incoming light intensities. It does not contain geometry. This is different for the scan-object. It is a meshed geometry. Most of the time we rely on a single model. For training (and testing at training-time) we mainly use "Suzanne" - the generic semi-complex primitive of BLENDER, portraying a cartoonified ape head. In addition we use some "Leaves" - thin sheets randomly arranged. For evaluation (after training) we use a sphere and some other models of the *Stanford 3D Scanning Repository* [20] in addition (as depicted in Figure 3.5). To all these objects we



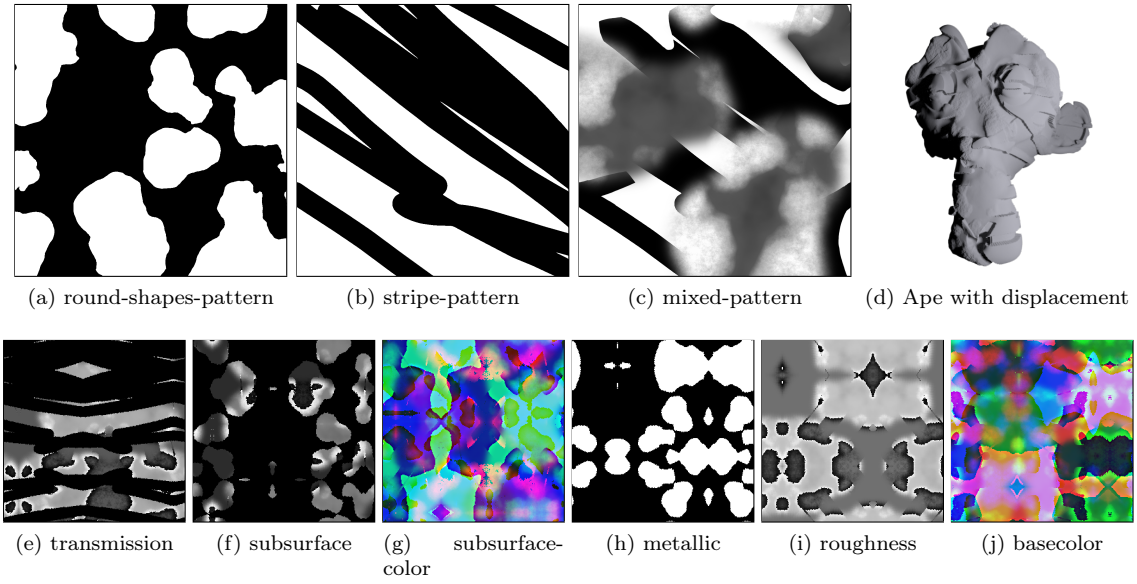
(a) Suzanne/Ape (b) Leaves (c) Sphere (d) Armadillo (e) Dragon (f) Bunny

Figure 3.5: The different meshes used for our datasets, rendered with a diffuse gray material. (a) Is the ape provided by BLENDER and the main object used for training and also for testing and evaluation. (b) Some thin Leaves, only used for training and specifically intended for *transmission*. The sphere (c) and the *stanford*-meshes (d-f) are only used for evaluation.

assign a randomly generated material and a random displacement. Besides that, we also apply subdivision on the ape, the sphere and the bunny beforehand. This means splitting quad-faces into four new faces and tri-faces into tree new ones. We do this to increase the vertex-count and to have a reasonable mesh-resolution for displacement. For the ape this results in now about 500000 vertices instead of the originally 504. This preprocessing was not necessary for the remaining stanford-models, which are already provided in high resolution. Besides these adjustments, we modify the ape further, when it is intended for training or testing. These modifications include random translations, scalings individual to each axis and random rotations. While applying these, we always make sure, that the object stays within the $1 \times 1 \times 1$ box around the world origin (for an easy implementation of rendering the global-position-map). However, the main variations are the ones between the different materials. How we randomly create them is explained in the next section.

3.1.3.1 Random Maps to Define Materials and Displacement

We define both, the individual BRDF parameters and the displacement with randomly generated parameter-maps. Simply put, such maps are just ordinary images. In computer graphics, they are



(a) round-shapes-pattern (b) stripe-pattern (c) mixed-pattern (d) Ape with displacement

(e) transmission (f) subsurface (g) subsurface-color (h) metallic (i) roughness (j) basecolor

Figure 3.6: The first three images in the first row show the hand-drawn maps, that we use to create all random maps with. They were randomly modified and combined to create the example BRDF parameter maps in the second row. The examples in the second row are then used as inputs for rendering Figure 3.4. Mapped on the scan-object, they lead to the pictures in Figure 3.4r - 3.4w. (d) shows the same object as 3.5a, but with an displacement-map applied.

the standard way to efficiently store spatially varying (surface-)information. They can be used to describe all kinds of data, that are required on a object's surface position. The data they sore is mapped via so called u, v-coordinates on to the object. We create our random parameter-maps as random modifications and combinations of the manually designed parameter-maps. For this, we use three different ones. Two binary ones, one mainly made from round shapes and another mainly consisting of a stripe pattern. To generate non-binary BRDF parameter-maps we also use the manually created binary ones, but in addition created another non-binary map manually. This manually created non-binary map combines hard edges like the boolean-maps, but in addition also contains smoother transitions between different areas. Further, we added areas with noise of varying spatial size, to allow mimicking the properties of a bigger variety of real-world materials. These three manually created maps are shown in Figure 3.6a - 3.6c. Before combining some to the random-parameter-maps, we individually modify the manually crafted ones each time we use them. First we apply a random rotation, followed by a cropping, so that the whole new image is without padding again (that was required for the rotation operation). Next we randomly shift the values of the map, by adding a randomly generated offset to each one and only use the floating-point remainder for values greater than one. Alternatively, when using a binary-map as input to create non-binary-maps, we assign random values to the two classes instead of shifting the values. When using manual-binary-maps to also create random-binary-maps, we can not apply the previous, but instead will switch the classes on chance only. Subsequent to these modifications, we make the maps seamlessly-tileable by applying mirroring. This means the map could then be extended infinitely by placing a copy of it next to itself (in any direction). Further, this extension would not create visible seams. Determining where the map was extended is not possible afterwards. We make the maps seamlessly-tileable to avoid artifacts later on the rendered objects. Subsequent to this we apply one last modification. Namely, we randomly translate the map by an individual amount along the x and y axis. Parts of the original map that would lie outside of the new maps are wrapped around.

Materials Examples of random-maps, that are created in the way we described so far are given in Figures 3.6h and 3.6i. 3.6h is the BRDF parameter map, that defines the *metallic* property for rendering Figure 3.4. Respectively Figure 3.6i defines the *roughness* property. In general this already describes the whole procedure to create the *metallic* and *roughness* maps. For *metallic* we use either the round-shapes or the stripe-pattern (Figure 3.6), randomly chosen and with a random inversion as described before. For *roughness* we use the mixed-pattern with randomly shifting the values and all the other non-binary modifications listed before. Very similar to the *roughness*-map we also create the maps for the BRDF parameters *transmission* and *subsurface*. The only difference comes from our desire, to have areas in the rendered object that do not contain eiter *transmission* or *subsurface*. To ensure this, we first create maps for these two parameters like those for *roughness*, but then multiply each by a random binary map (like those for *metallic*). Examples are again given in Figure 3.6. To enable rendering of the scene, all that remains is to define the parameters for *basecolor* and *subsurfacecolor*. These maps differ from the previous ones in the sense that they define colors, i.e. a three-tuple per pixel instead of a single value as before. This already suggests, how to create them. For both we proceed identically. We use the same method as before (for *roughness*) to generate the values individually for each color-channel. Two examples of possible results are provided in Figure 3.6j and 3.6g. These resulting BRDF parameter-maps are then used with the uv-unwrapped mesh to define the values on each surface point of the rendered object. For uv-unwrapping we used the functionality provided by BLENDER. Note, that therefore, we computed the uv-maps for each of the objects (Figure 3.5) beforehand.

Displacement In addition to varying the material parameters, we also integrated some randomness into the geometry of the objects. A reason is to avoid having only relatively smooth surfaces in the training set (due to the geometry of the *ape* model). To simplify the implementation, we just use the hand-drawn map in Figure 3.6c to define the new surface height. White means a displacement along the surface normal, while black means a displacement against the direction of the surface normal. To obtain individual displacements per scene, we randomly modify the precomputed uv-maps by translation, rotation, and scaling (in the range $[1, .., 4]$). We also randomly vary the magnitude of the displacement in the range $[0.005, .., 0.015]$. An example of this is shown in Figure 3.6d. It is a displaced version of the *ape* model we presented in Figure 3.5a.

3.1.4 Induce Randomness to Scanner Configurations

Besides the variations of the scanned-object, we also introduce randomness in each part of the scanning setup itself. With this we mean the light colors and intensities, the positions of the scanner and its individual parts and the camera configuration. For the camera we only vary the focal length. Before rendering each scene, we pick a random value in $[20, 30, 30.99, 40, 50, 60]$ for it. This leads to notably visible variations in the field of view (FOV). We do this with the intention to hinder our network to learn features that are related to 3D-distances, by pixel-distances in the input-images. With the variations in the FOV the network is required to rather use geometry information, that is provided as input for its reasoning (for example the depthmap and the view/light-angles).

Lights For practical interests, but also to encourage generalization in training, we insert randomness in the light sources. Similar to the camera's FOV we randomly vary the size of the area illuminated by the projector. Moreover, we randomly increase the light intensities. We do this for all lights individually by adding at maximum half their initial strength – except for the flash lights. For them we share the random increment to mimic lights, that are identical in construction. However, to simulate slight deviations we add an additional individual random intensity increase that is of smaller magnitude. Combined the flash intensity I can be described with Equation 3.1:

$$I = 1 + 0.5 \cdot R_s + 0.1 \cdot (0.5 \cdot R_s) \cdot R_i \tag{3.1}$$

where R are random values in the set $[0, 1]$ and R_i is individual and R_s is shared between the the flash lights. Besides that we also adjust the light colors per scene. We randomly vary between cold

and warm white emitted light. This is implemented by randomly modifying the saturation values of the backlight or the flashes. We modify the flash light colors in a similar manner as we did with the light intensities – a shared random part and a smaller individual random part. In addition to that we modify the backlight further. When turned off it is depicted as a plane of a white diffuse material (with RGB=$(0.85)^3$). To each color channel of the basecolor of this material we add an individual random value in $[0, .., 0.1]$.

Locations Even further we modify the backlights location, by slight individual random rotations around the x and y axis (ignoring z – the view direction). In terms of relative position, we also adjust the flash lights (and the camera, which is tracked to the position of the first flash) randomly. We do this by first setting a random baseline length and second by randomly individually moving the flashes further (relative to the baseline center). This random translation is at max $\pm 0.1 \cdot |baseline|$ in x direction (the extend of the baseline) and individually at max $\pm 0.05 \cdot |baseline|$ in the direction of the other axes. Similarly, we move the projector by at max $\pm 0.5 \cdot |baseline|$ along the baseline and by at max $\pm 0.05 \cdot |baseline|$ along the perpendicular axes. The cameras rotation is automatically tracked to the focus center. This moreover, is placed at $(R_x \cdot 0.1, R_y \cdot 0.1, R_z \cdot 0.5)$, with R being random values in $[0, .., 1]$ again. Likewise we also randomly translate the baseline center (and the camera and flashes, that are tracked to it) that is initially located at $(0, 0, -1.5)$ by $(R_x \cdot 0.1, R_y \cdot 0.1, R_z \cdot 0.1)$.

World Besides the scanner related modifications, we also apply some randomness to the world that surrounds the setup. Namely, we initialize the environmentmap with some variation in both, the rotation and its illumination strength. This already finalizes the modifications. These random changes are all constant for a complete scene. For different positions within that scene we only rotate the whole scanhead (lights + camera) around the world origin (reconsider Figure 3.1). Most of the time we use three random rotations that are relatively close to each other. More specifically, the second and third positions are relative to the first position within random rotations in the range of ± 36 degree around the x and y axis. This produces renderings that differ significantly enough in viewing positions to lead to notable variations in view- and light-angles, but still contain a reasonable amount of overlap in the captured 3D positions.

3.1.5 Dataset-Values Computed after Rendering

In Section 3.1.2 we described all the images, that we rendered with our scanner-setup implementation in BLENDER. For training our architecture we do not use all of these images. Instead we post-process some of them to get representations of the stored information, that are better suited for learning. This refers to the angles, that we compute from the surface-normals, the view- and lightvectors. In addition, we perform some post-processing to separate information implicit in the renderings from them. For example the light-intensities and colors. Further, we finalize renderings, that we only created as intermediate results (to ease the definition of the shader). Like the matchmaps, that we compute from the positionmaps.

Light Response To predict some of the BRDF parameters later, we require information about the scene illumination. To hold the capturing as simple as possible we only require the response of a diffuse white material, with its normals pointing roughly against viewing direction. We did not make any further, more precise constraints than this. The idea is, that in a real-world-application one could just use a sheet of paper as reference. Even without knowing its exact visual-material-properties. In rendering the dataset, we did not explicitly model a sheet of paper next to the scanned object. We avoided this to keep the scene simple and therefore save runtime[2]. As mentioned before we defined the rendered backlight roughly as a white diffuse material. For generalization purposes we added some variation in its exact color for each scene. For training

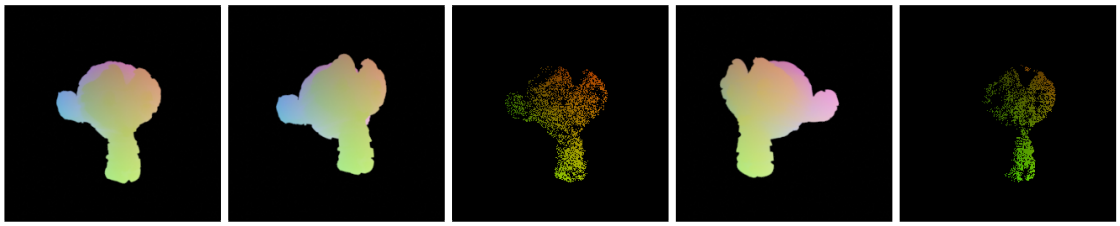
[2]Note, that due to the pathtracing algorithm a second object in proximity is likely to cause additional light-bounces and can therefore become especially costly to use.

(a) ambient (b) ambient + bl (c) ambient + flash (d) masked (halfway) (e) masked (final)

Figure 3.7: Example results of the segmented backlight used for the mean response of the light shined on a whited diffuse material. With the ambient lit scene (a) and the backlight turned on (b) we compute a first mask (d). This mask and the image rendered with the considered light (c) are used to compute the final mask (e). (d) and (e) show the masks applied to (c), which is used for the computation of the mean light response.

(and also in an actual usecase) we can therefore also use the turned-off backlight as the reference sheet of paper. In a postprocessing step we compute the mean value of the backlight surface, when illuminated by the considered light. To do so, we first segment the backlight from the rest of the image. We do this by thresholding on the difference between the ambient image and the one with the backlight turned on. This segmentation strategy is not optimal, as it can cause false positives in regions where the object is transparent or where a dominant fresnel-effect is observable. For an example of the later see Figure 3.7b and the respective errors caused in 3.7d. Nonetheless these areas are usually negligible small compared to the image-space occupied by the backlight. Because of ease of implementation and moreover because this strategy allows to work with already rendered images (not requiring further ones) we rely on this method. After segmentation we use the obtained mask to compute the YCbCr responses for all images illuminated by different lights (flashes / projector). However, for each we need to adjust the mask. This is due to the fact, that each light together with the scanned object casts an individual shadow on the backlight-plane (see Figure 3.7c on the left). To get rid of this shadow we first compute the mean response of the whole area contained in the backlight. Next we use it for a second thresholding and with the brighter regions (within the previously computed mask) we compute the mean again. This leads to some falsely masked pixels and causes the "mean" to be slightly too bright. Nonetheless, this also removes the shadow, whose presence would lead to a far to dim "mean" light response. Figure 3.7 shows an example scene and both masks computed according to the previous algorithm. Finally, we save the mean values in a .txt file and use them to create single colored images on demand as inputs for the network later.

Matchmaps Next we compute the matchmaps. They serve as a means to relate corresponding image locations viewed from different positions to each other. We require them later to modify input-images, so that even if different positions are used for input, all pictures look as if taken from the same single position. To compute the matchmaps we use the previously rendered (global-) positionmaps. With one pass over the first positionmap we store the pixel-coordinates in a dictionary, using the respective global positions as keys. In a another pass over the second positionmap we access the dictionary again. But now with the keys defined by positions in the second map. This returns the corresponding position in the first map. Because our dataset only contains images of 256×256 pixels, we ease our implementation by representing the matchmaps as 8bit images. This means, that we store at each position of the first map the corresponding position of the second map, encoded in two 8bit channels (red, green color). Note that this also limits or implementation in the way, that we can only work with images of size 256×256, if we want to use multiple different positions. With an alternative implementation of the matchmap, this constraint is easy to overcome (e.g. using a higher image depth, or even better a list stored as .txt or binary). We avoided this extra work because for training with our dataset, the mentioned method serves its purpose and the rendered positionmaps are also only in 8^3bit resolution anyways. The depth of

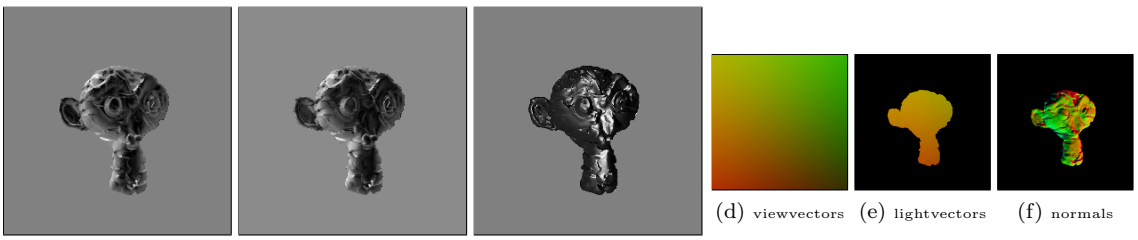


(a) Positionmap 0 (b) Positionmap 1 (c) matchmap $P0 \leftarrow P1$ (d) Positionmap 2 (e) matchmap $P0 \leftarrow P2$

Figure 3.8: Some positionmaps and the two resulting matchmaps. A pixel-position in the matchmap is related to a pixel position in the first positionmap (and all other renderings of the same position). It stores the corresponding pixel-position of the second positionmap encoded as red, green image. The matchmap $P1 \leftarrow P2$ is computed in the same manner, but the symmetric reversed directions are not.

these could be increased as well (BLENDER would allow 16bit outputs), however this would lead to another problem. If the global position would be too accurate, then there would not be any exact correspondences anymore amongst the different positions. This is caused by the different viewing-angles combined with the renderings being an discretization of limited resolution of the viewed scene. It would then be necessary to match global positions by their distance. Using lower precision in the positionmaps acts like a binning beforehand and therefore saves us from the necessity to implement such a functionality as well. We compute all matchmaps as described in a preprocessing step and save them with the previously computed dataset images (reconsider Figure 3.4). However, to save memory we only save (and compute) the matchmaps $A \leftarrow B$ from a position B with index i_b to anoter position A with index i_a if $i_a < i_b$, as the mapping is symmetric. If we later need the $A \rightarrow B$ we simply invert after reading the respective map from the disc. Moreover, we only save matchmaps, that contain a reasonable amount of matches. We define reasonable in this context as 10% of the number of pixels contained in the area of the object mask (reconsider Figure 3.4n). Figure 3.8 shows for an example scene three different positionmaps and the computed matchmaps that relate two of the possible pairs.

BRDF Angles The last values that we compute for training directly from the previous images are the angles between the vectors, that define observed light responses in the renderings. These are the pixel-wise angles $\angle NV$ between the surface-normals N and the viewvecotors V, the angles $\angle NL$ between the surface-normals and the lightvectors L and the angles $\angle VL$ between the light- and viewvectors. We do this, as the response of the (isotropic) BRDF depends on these angles and not the vectors. Moreover, the angles are a more compact representation than the 3-dimensional vectors. Supposedly this concise representation eases learning. However, to save computational cost we do not compute the actual angles, but rather the cosine of them. To be precise we compute



(a) $\angle NV$ (b) $\angle NL$ (c) $\angle VL$ (d) viewvectors (e) lightvectors (f) normals

Figure 3.9: An example to the computation of the BRDF angles. The last three images show the vectors that are used as inputs. They are all defined in the camera-space pointing from the object/scene to the camera. RGB values are used to encode XYZ values. The first three images show the computed angles. White refers to big angles and black to small ones. For implementation we use the cosine of these angles (to spare computation of *acos* for each value).

them as follows in equations 3.2, 3.3 and 3.4.

$$cos(\angle NL) = \frac{NL}{|N|_2|L|_2} \tag{3.2}$$

$$cos(\angle NV) = \frac{NV}{|N|_2|V|_2} \tag{3.3}$$

According to the definition of the BRDF angles, $\angle VL$ is not the direct angle between V and L, but rather the angle between these vectors projected on the objects surface. We therefore compute $\angle VL$ according to Equation 3.4. Meaning we first compute the projections V_S and L_S by subtracting the height of V and L relative to the surface-normal. The height-vector can be computed from the unit-vector in N direction (N is already defined like this) scaled by the scalar projection of V or L on to N.

$$\begin{aligned} V_S &= V - \left(|V|_2 \frac{VN}{|V|_2|N|_2}\right) \frac{N}{|N|_2} = V - \frac{VN}{|N|_2^2} \cdot N \\ L_S &= L - \frac{LN}{|N|_2^2} \cdot N \\ cos(\angle VL) &= \frac{V_S L_S}{|V_S|_2|L_S|_2} \end{aligned} \tag{3.4}$$

Finally, we can compute the maps (as shown in Figure 3.9), that contain the respective angles for each observable image position. Because these computations are not too expensive and re-use many intermediate values, we re-compute the angles each time on demand in favor of saving disc-space.

3.1.6 Filtering Troublesome Data

The final step in creating the dataset is sorting out some of the rendered positions. We do this, because some few renderings contain light configurations, where the environment-lights absolutely dominate over the flashes and the projector. If they were used for training, they would supposedly lead to worse results. The argument for this is, that the net would have to come up with a guess that can not be inferred from its inputs. It would therefore have to learn a reasonable good guessing and would waste resources on that. Even worse, it might use its learned guessing for scenes that provide good inputs. Such scenarios with bad lighting conditions are caused by unlucky (random) orientations of the environment-map, the scanned object and the scanner. The main causes are usually very bright light sources in the environment such as windows or other transitions from a covered place to the outdoors, or sometimes also strong lamps. Detecting such configurations beforehand is not easy to implement. Moreover, this problem usually only arises for one or a few

P0 A P1 A P2 A P3 A P4 A P5 A P6 A P7 A P8 A

P0 AL1 P1 AL1 P2 AL1 P3 AL1 P4 AL1 P5 AL1 P6 AL1 P7 AL1 P8 AL1

Figure 3.10: An example for a scene, that contains positions with bad light conditions. In some the artifical light of the scanner is not visible, because of the bright light from the environment (likely the one visible in position 6). The first row shows the only ambient lit scene. The second shows illumination with the second flash light in addition. Each column is a position in that scene. By the strategy we explained before positions $0-3$ are sorted out, the others were rated as good.

positions within a scene. Sorting them out after rendering is not too wasteful and eases coding. To detect such problematic positions within a scene we simply compare the differences between the *ambient* render and all others with different lights (flash0,1, projector) turned on. We then check if the maximum difference of any of them is below a threshold of 0.1. If so, we store this position (and scene) in an ignore-list in the dataset. When loading the inputs for training we check the list for such and skip these positions. If at least one position remains in the desired scene, we still use it for training. Figure 3.10 contains an example of such a problematic scene.

3.1.7 Further Variants of the Dataset

What we described in the last few sections is only one of multiple datasets that we use for training our network. In terms of the used materials, this is the most complex dataset that we use. Besides it, we created several more datasets, that are all simpler variants of this one.

multimat - The Previously Introduced Variant of the Dataset To recap, the previous dataset was created with totally random materials. For each BRDF parameter we used unique random maps of spatially varying values. This makes it impossible for any BRDF prameter-inference-technique to estimate one parameter using the distribution of others as guidance, as long as they are not inherently correlated. As an example, an area with a constant *metallic*-value does not necessarily need to have the same *basecolor* everywhere – stronger yet, it even is unlikely. For a few examples of different scenes of this dataset, refer to Figure 3.12a. We call this dataset **multimat**. The first variant that we crate is the exact same dataset, but without an environment map (called **multimat-noenv**). Therefore, the incident illumination is always for all directions a constant gray-value. A scenario like in this dataset would be the ideal case for capturing material samples. We want our architecture to work in such controlled environments as well and add this dataset therefore. Moreover, some early experiments with a barley modified U-Net indicated, that the results for the uncontrolled environment improve when using both variants for training. We expect the improvements were caused by the variation in environment for identical scenes, but did not investigate it further. In addition, we would also expect an improvement when using even more varying environments for the same scene. However, this adds computational cost, for which alternatively also completely new scenes (that also use different environments) could be rendered. We did not further investigate which ratio of applying both strategies is best. Moreover, this might also depend strongly on the difference in the envirionements of the acutal use-cases. For example if the program is only used in a completely controlled environment it is likely sufficient to only train with the *-noenv* datasets. Whereas in the general case many variations of different environments for the same scene might be beneficial. If the use-cases can be constrained in some way, for example only office rooms (with windows and lights only on the ceiling), then again it might be sufficient to render the same scene with less environments, than the general case. However, due to the necessary computation-time and because different scenes are already rendered with different environments, we did not render any distinct scene with a different environment again. Likewise we did not render all scenes without the environment again (only a subset of 900). For the same reason we did not re-render any of the following dataset variants without environment. Some exemplars of the variant without an environmentmap are given in Figure 3.12b.

2mat - A Dataset with Only Two Materials Materials with such an amount of randomness, as the ones in the previous two datasets, are supposedly good for training in general. They counteract bad generalizations and overfitting. Still, these kind of materials are far more complex than most real world examples. Very often big parts of an objects surface are made out of the same material. Objects coated with such materials therefore provide multiple samples with different surface-normals and different view- and lightvectors. In theory a network could detect segmented areas and improve predictions for these areas, by combining all contained pixels as input to compute an output, that is constant over the segment. Strictly speaking, such segmented materials could also occur in our *multimat*-dataset. However, these scenarios occur way to rarely

and their spatial extend would very likely be rather small, if they occur at all. To allow our architecture to train with an increased focus on such types of materials, we create another dataset and call it **2mat**. As the name suggests, we refrain from using totally random materials and only use two unique ones for each scene. For each of the two materials we select the BRDF parameters randomly but constant. To divide the BRDF parametermaps into two materials we use the binary manually defined masks (reconsider Figure 3.6a, 3.6b) again, with all the adjustments we described in that section. In addition to that, we create another variant (**2mat-transmission**), with the purpose of containing more transparency and *subsurface scattering*. We do this by disabling *metallic* for one of the two materials and assigning the other BRDF parameters randomly as before. For the second material we assign all parameters as before, meaning that also both materials can have *transmission* and or *subsurface scattering*. Some examples of it can be seen in Figure 3.13b.

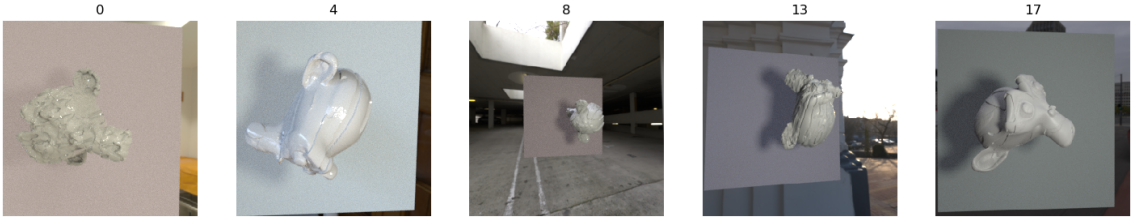
predefined - A Dataset without Randomness Besides the two dataset classes that work with random materials, we also create some sets without. We do this to ensure having samples over the complete BRDF parameter space for training. With this we do not mean just occurrences of said BRDF patameters in a few pixels (which is most likely already provided by the random datasets), but rater greater areas of the object covered with these BRDF parameters. To achieve this, we use spatially constant BRDF parameters for most of these datasets. In detail we create one dataset **predef-subsurf** with varying *subsurface* values and all others fixed at some reasonable average (Figure 3.12c). Similarly, we also do this with another geometry for *transmission* and *subsurface* in the **leaves**-dataset (no Figure). For it we use randomly arranged thin sheets, that also contain some random displacement (as shown in Figure 3.5b). Same, but with the "Ape"-geometry again, we do for the *basecolor* (**predef-basecolor** in Figure 3.13c). Again varying it, fixating all other parameters. The variations are made in the HSV color space with 20 uniform samples for *hue* and *saturation* and *value* in $[.33, .66, .99]$. For *roughness* we do this again, but this time for multiple versions (**predef-roughness-...**) where we fixate the other BRDF parameters at different values. To name it, we vary over *roughness* on a metallic material with each, a white (Figure 3.11a), a gray (no Figure) and a black (Figure 3.11c) *basecolor*. Moreover, we also do this for the non-*metallic* versions (without *subsurface* and *transmission*) of white (no Figure), gray (Figure 3.11b) and black (no Figure) *basecolors*. So far all datasets depict only artificial materials. To also have some spatially varying real-world examples, we rely on BRDF parameters from [27]. However, in terms of our needs, they only provide *basecolor* and *roughness* values. Therefore, we fixate the other BRDF parameters again. We always use a non-*metallic*, non-*transparent* material without *subsurface scattering*. We suppose that these constants are a good choice for most of the 176 contained materials. The majority of them are floor or wall materials like wooden planks, bricks, sand or grass, but also some samples of textiles and leather are contained. Besides the mentioned BRDF parameters, this source also provides displacement maps for all materials. We use them instead of our random-method, to render the dataset – called **textureheaven**. Some of the rendered scenes are shown in Figure 3.11d.

Dataset for Testing and Evaluation For testing we simply create a few more scenes of the *mulitmat* and the *2mat* dataset. For evaluation we create some further scenes of the previous two sets again, but in addition also new ones. For this, we use the *2mat* algorithm, but apply it to other geometries as well. Namely the ones shown in Figure 3.5 before. Besides that, we also create some scenes that contain three different materials. For this we adopt *2mat* by splitting the the second material into two parts again. To each of the split regions we assign individual random *basecolor*, *roughness* and *metallic* values. Renderings of this **3mat**-dataset are shown in Chapter 4, when evaluating the results.

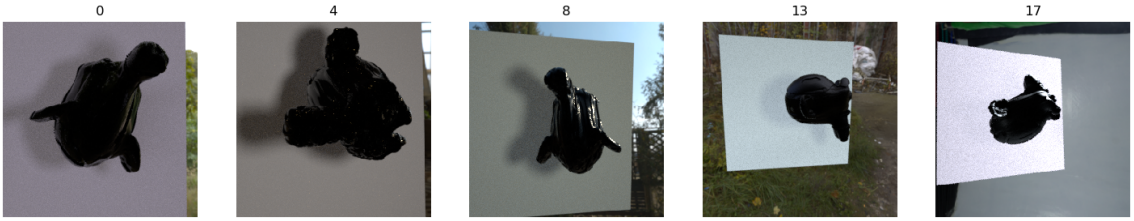
To summarize, we mainly rely on two dataset-types for training and testing. These are *multimat* and *2mat*. For training we have some further variants of them to ensure, capturing the input-space of the BRDF parameters exhaustively. For evaluation we use datasets in addition, that work with geometries and material-creation-techniques, that where unseen in training and

testing. In total, the rendering of all images of the training-dataset took us about one month on a PC with an 4-core Intel Xeon @ 3.40GHz and a Nvidia GTX 980 Ti.

(a) predef-roughness-white-metallic (40 scenes for training)



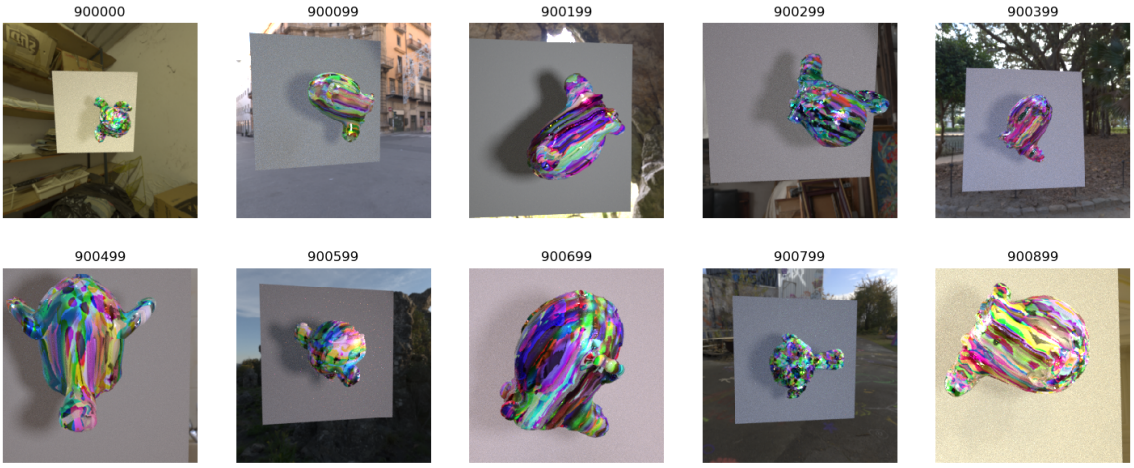
(b) predef-roughness-gray (40 scenes for training)



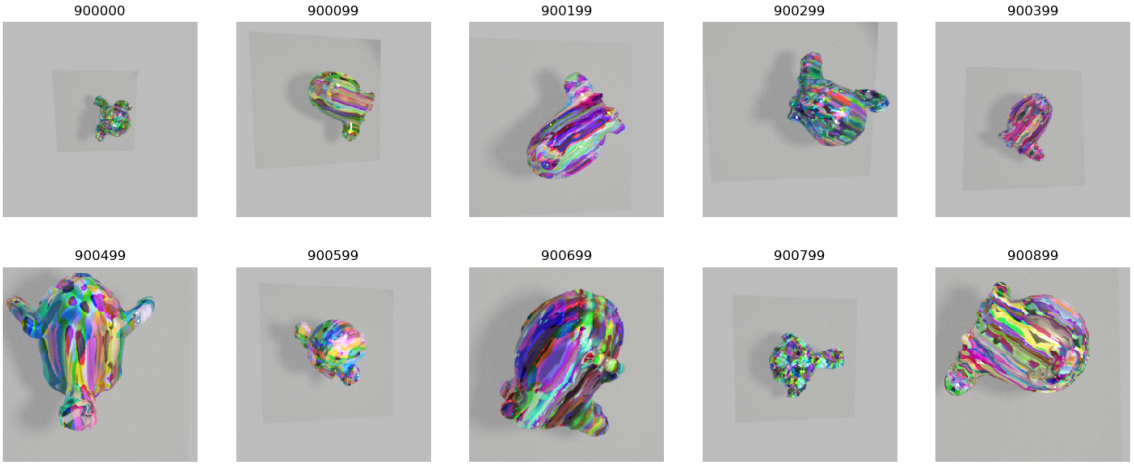
(c) predef-roughness-black-metallic (40 scenes for training)

(d) textureheaven (167 scenes for training)

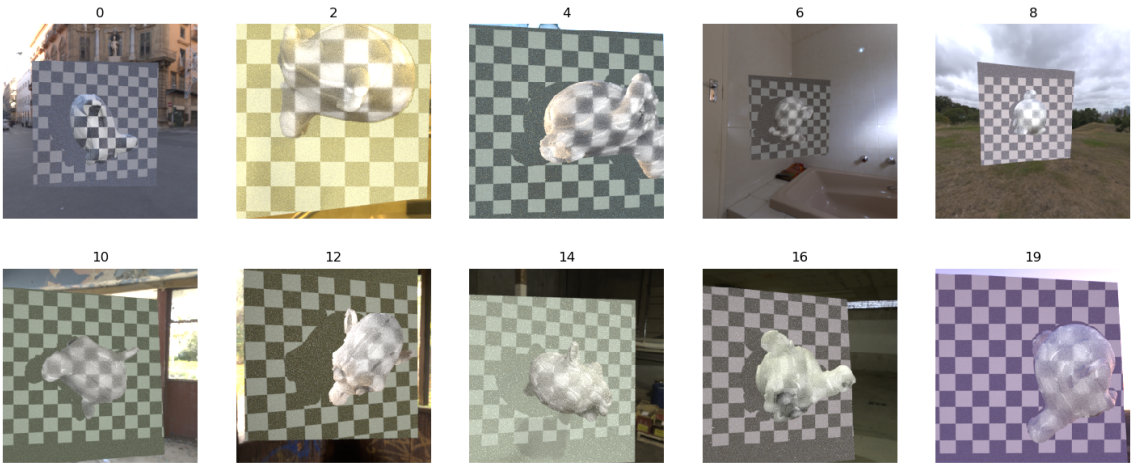
Figure 3.11: Different dataset types. Each visualized by the ambient + flash1 lit renderings of the first position of various scenes. We also created non-*metallic* versions of (a) and (d), again with 40 scenes each.



(a) multimat (384+809+900 scenes for training + additional for testing and evaluation)

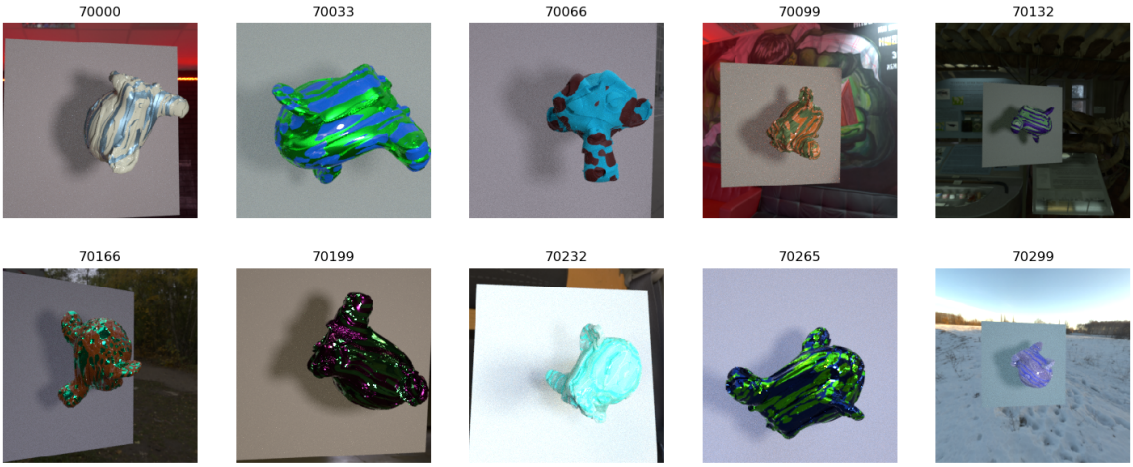


(b) multimat-noenv (900 scenes for training)

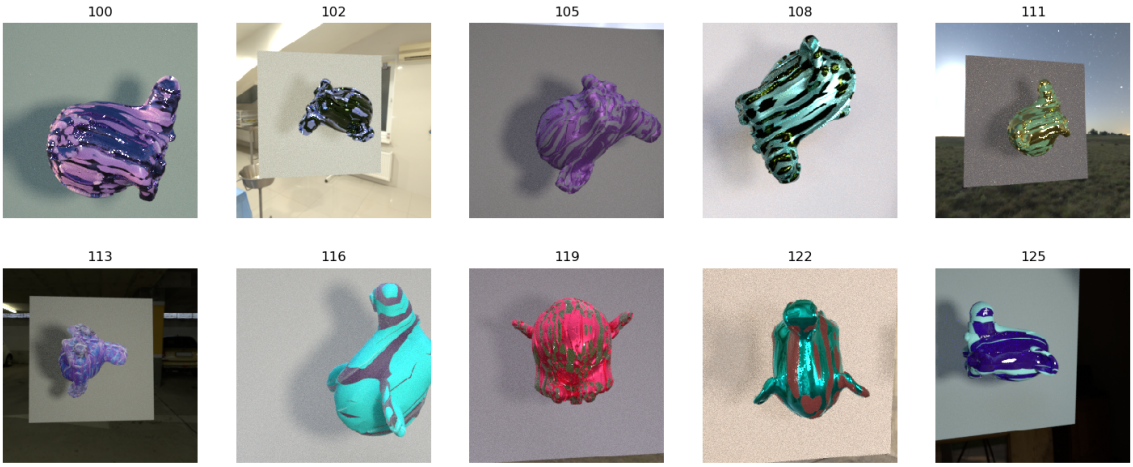


(c) predef-subsurf (20 scenes for training)

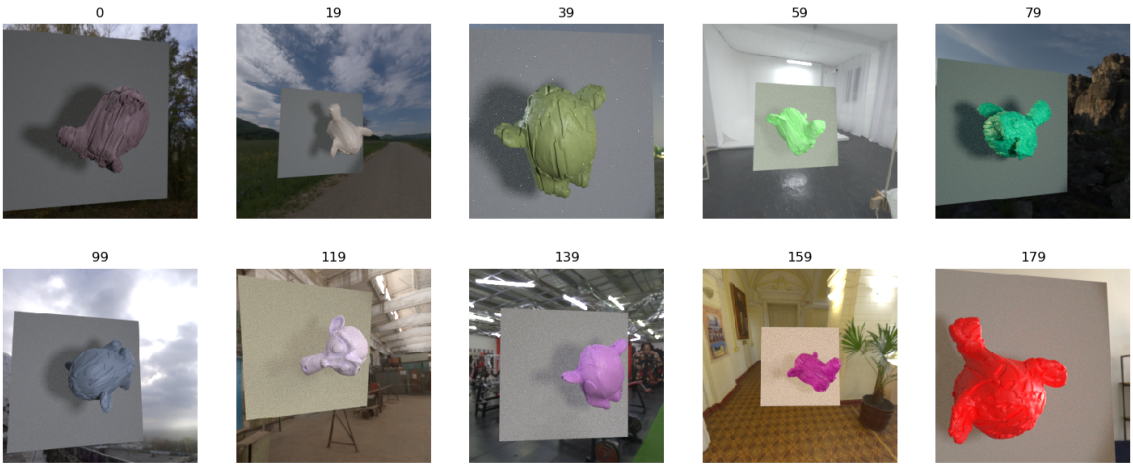
Figure 3.12: Different dataset types. Each visualized by the ambient + flash1 lit renderings of the first position of various scenes. Except for (c) here illumination of the scene is done with the projector showing the checkerboard (d100) image.



(a) 2mat (300 scenes for training + additional for testing and evaluation)



(b) 2mat-transmission (29+26+27 scenes for training)



(c) predef-basecolor (180 scenes for training)

Figure 3.13: Different dataset types. Each visualized by the ambient + flash1 lit renderings of the first position of various scenes.

3.2 BRDF Parameter Inference

To solve the problem, that we described in Chapter 2.4 we split the prediction of the BRDF into the separate prediction of the individual BSDF parameters. While being aware that a combined learning might detect interdependencies between the parameters and yield better results, we incorporated this dividing strategy for several reasons. First of all we have generated a dataset according to the outputs of our 3D-scanner. This gives the option to use many feature maps as a single input to the network, however doing this might result in unnecessary high vram usage. Moreover, certain inputs are less relevant to learn some of the BSDF parameters, than others. To name some examples:

- For inferring the *basecolor* of a opaque material, the backlit photo is most likely neglectable.
- Likewise captured color is irrelevant to most parameters and luminance information suffices.

Providing only the reasonable input-features for a distinct BSDF parameter might also ease the task of the learning algorithm. Further, the networks (one for each BSDF parameter) could later be used together (with or without all available inputs) as inputs to another new (cheaper) network, that ensembles them. Due to time constraints we did not implement this idea, but several times we are using some BRFD-parameters as the input for the prediction of others. Another benefit of this strategy is due to that we require special input for the estimation of some parameters. If this is not available or if said parameters are known beforehand, prediction is still possible and can even be shortened. In detail, we created five individual networks. One for each of the sought BSDF parameters, namely *metallic*,*transmission*, *subsurface*, *roughness* and the *mixed subsurface basecolor* (which we just call *basecolor* for the sake of brevity). Training a net with other BSDF parameters as inputs was done using the groud truth data of them. Naturally when applying the networks to real world data, the earlier inferred BSDF parameters must be used as input for the later networks. Figure 3.14 shows how they depend on each other. The overall architectures used for them are very similar to each other, with only slight modifications adopting to the BSDF parameters. For this reason we first describe the common part in the following. Then we point out the modifications for the separate BSDF parameters. These were mainly implied by the different in- and outputs.

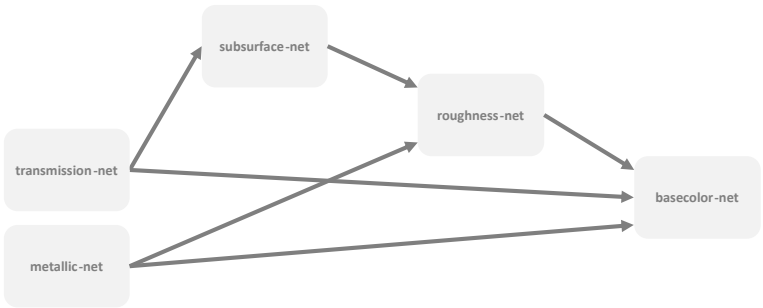


Figure 3.14: We train five individual networks, one for each BSDF parameter, with distinct inputs. This allows separated training and partially independent inference. In addition it has the benefit that, if some inputs in the early stages are missing (eg. to simplify data acquisition) the network is still capable of inferring the other BSDF parameters. Moreover, if some parameters are known beforehand (eg. spatially constant ones like zero transmission etc.), then the inference can be shortened. This graph shows the data-flow and with it the dependencies between the networks.

3.2.1 Architecture

Based on our problem description, we try to build a model that is able to handle arbitrary input sizes (and produce outputs with the same resolution). The intuitive approach is, to compute the

BRDF parameters pixel by pixel. However, due to noise in the inputs and lack of information about the neighborhood, the predictions of such an approach are not expected to be good. An improvement would be to include the information of the neighboring pixels with filters. For example, implemented with an ordinary convolutional neural network (CNN). Still the accessible neighborhood would be limited and the problem would remain. As [14] argued, it is critical for a network in BRDF estimation to have the ability to access global features. They solved this issue by introduction of a fully connected network (FCN) that is initialized from all input-pixels and run in parallel with an U-net (a more advanced CNN, intended for image translation) [25], to support it. A second argument for their FCN is, that they used batch normalization in the U-net, which hinders learning of the mean-color. The FCN helps to retain information about it. However, using a FCN also constraints in- and output sizes to a fixed resolution (they used 256).

We seek for a method that is flexible in resolution. Moreover, we trained our network without any normalization (besides having inputs in the range of $[-1..1]$), like others in the field did before us [21]. Equally, we instead rely on dropout for regularization. Therefore, we stuck to the idea of simply increasing the receptive field of our network, to enable learning of global features. Yet, just deepening a CNN would not suffice. First of all, if the whole input image is expected to lie in the receptive field of each output-pixel, this would be costly for larger resolutions. Second, with the constraint of having the whole image as receptive field, this approach would not even be flexible in resolution. To overcome this, we are using an iterative approach, where we work with the image-pyramid (halving the image-size each level) of the inputs. We first try to predict the BSDF parameters on the highest pyramid-level (the smallest sized inputs). In the following steps we refine our current predictions with the inputs, but twice as high resolved as before. If the pyramid is computed completely, then the highest level is a single pixel (of multiple channels) and global features can be directly computed from it. If some of the input-channels are a color image, its mean-color is already computed by the pyramid. For the reason that for our case the object of interest never fills the image completely, we do not compute the full pyramid, but stop on the level that has resolution 16×16 (with a net that has a receptive field of 5×5 for each pixel).

The second goal for our architecture is, to harness the information of multiple views of an input object. Views in that sense mean three different things. Firstly images taken from the same position, but with different lighting. In addition different views also mean images shot with the same illumination but from different positions, if all corresponding 3D-points are mapped to the same 2D-image-points in the subsequent. Last, with different views we also mean both, varying camera positions and different light configurations together. To summarize, with a view we describe a set of surface samples (stored as a multichannel image). Another view would be the same set, but either with varying view- or lightvectors or both. An example of this is given in Figure 3.15. To achieve the utilization of a variable number of views, it is necessary to have a dynamic model. For this reason we spawn an individual network for each view that is used as input. Note, that all these different instances share the same weights. Next we need to fuse their information at some point. We decided to do this as early as possible and moreover do it in an repetitive alternating manner together with the upscaling, that we mentioned before. This fusion is in general just a max-pooling of the features from each of the per view networks, pairwise executed until only one fused feature-set remains. These fused features are then used to overwrite the features of all individual per-view-networks. The only exception is the architecture, that is used to predict *roughness* – where fusion and updating of the old features are slightly different. This will be discussed in Section 3.2.3.1. After fusion is done, the features are upscaled (with nearest neighbor) and refined again until the lowest pyramid level with the highest resolution is reached. At this point after the last fusion, all per-view-networks store the same feature-sets and an arbitrary one can be picked for further processing. Which boils down to compute the desired outputs from the features. Figure 3.16 summarizes the previously described in a schematic of our network architecture.

In the following we enrich this high-level description with more details. We start from the most detailed view and work our way up, until we arrive at this abstract view again. In the following we only describe the generic architecture, that is used to infer a single BSDF parameter. As mentioned before the nets for the different BSDF parameters are very similar and their variations

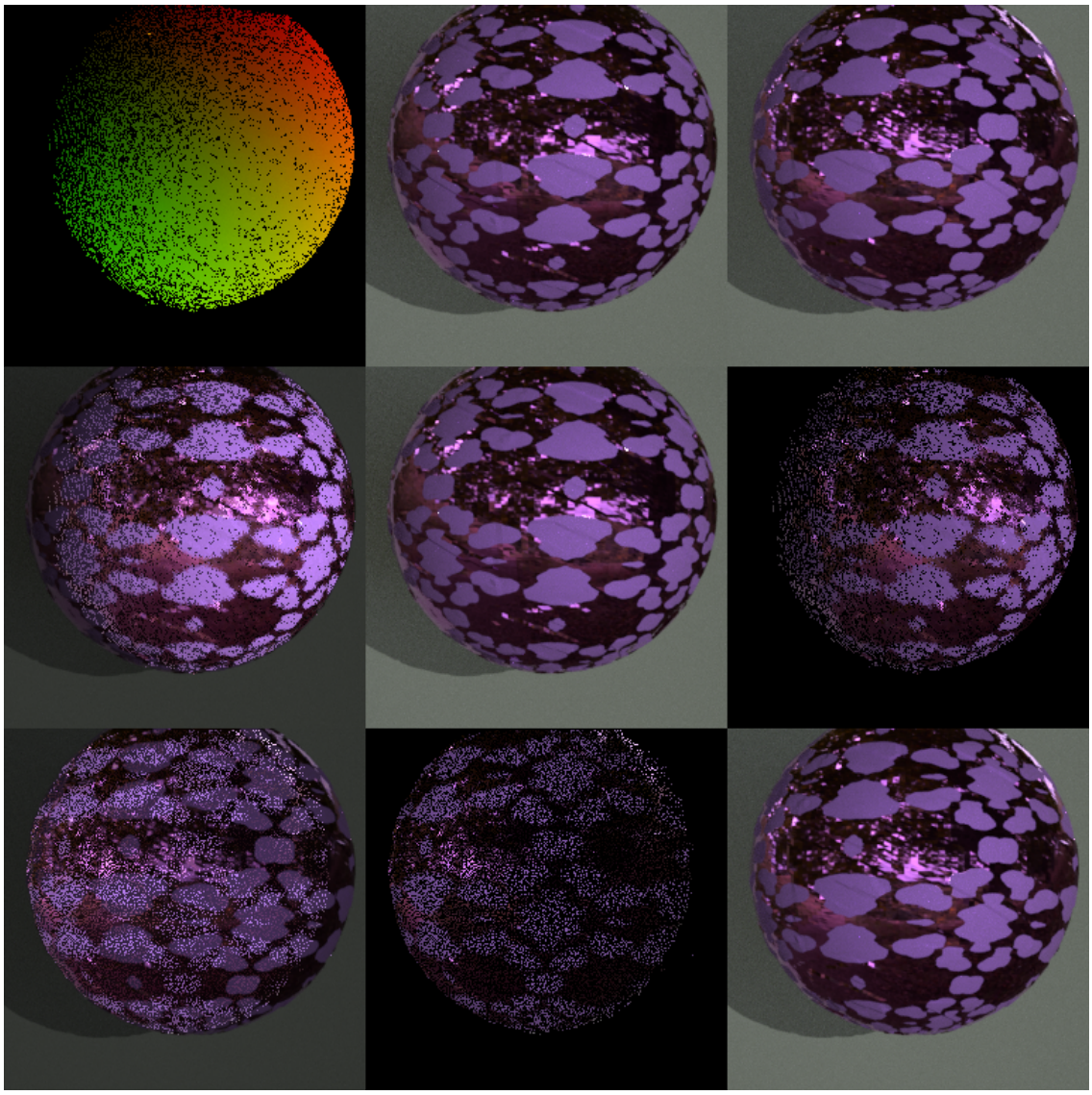


Figure 3.15: A sphere captured from different positions, but under the same light conditions. The two photos on the top right show the original captures. Whereas top left is the mapping that brings corresponding 3D-points to the same image loction. It is applied to the capture (top right) to make it seem like both pictures where captured with the first camera (first row). The first image in that row shows an overlay of both (first original and mapped second capture), but the mapped image was brightened a bit to highlight the overlapping. The third row shows the inverse mapping applied to the first capture, so that it seems everything was photographed with the second camera. Note how the specular highlights moved in the mapped images, despite the same light conditions.

are described in the subsequence. For one BSDF parameter our network is entirely build from four different types of blocks. Moreover, all blocks that reoccur in the net and are of the same type, share weights (however, different weights for the different BSDF parameter-networks). Namely the four different types are the *input stem*, the *core*, *core'* and the *output stem*. Again all four blocks are very similar to another. They are all crafted like one of the main blocks in the *inception network* [26]. Together we use them to build an image-to-image translation network that has a limited receptive field. The *core*, *input stem* and *output stem* are essential for this, whereas the *core'* is only used to deepen the network. This in theory allows the net to learn more complex

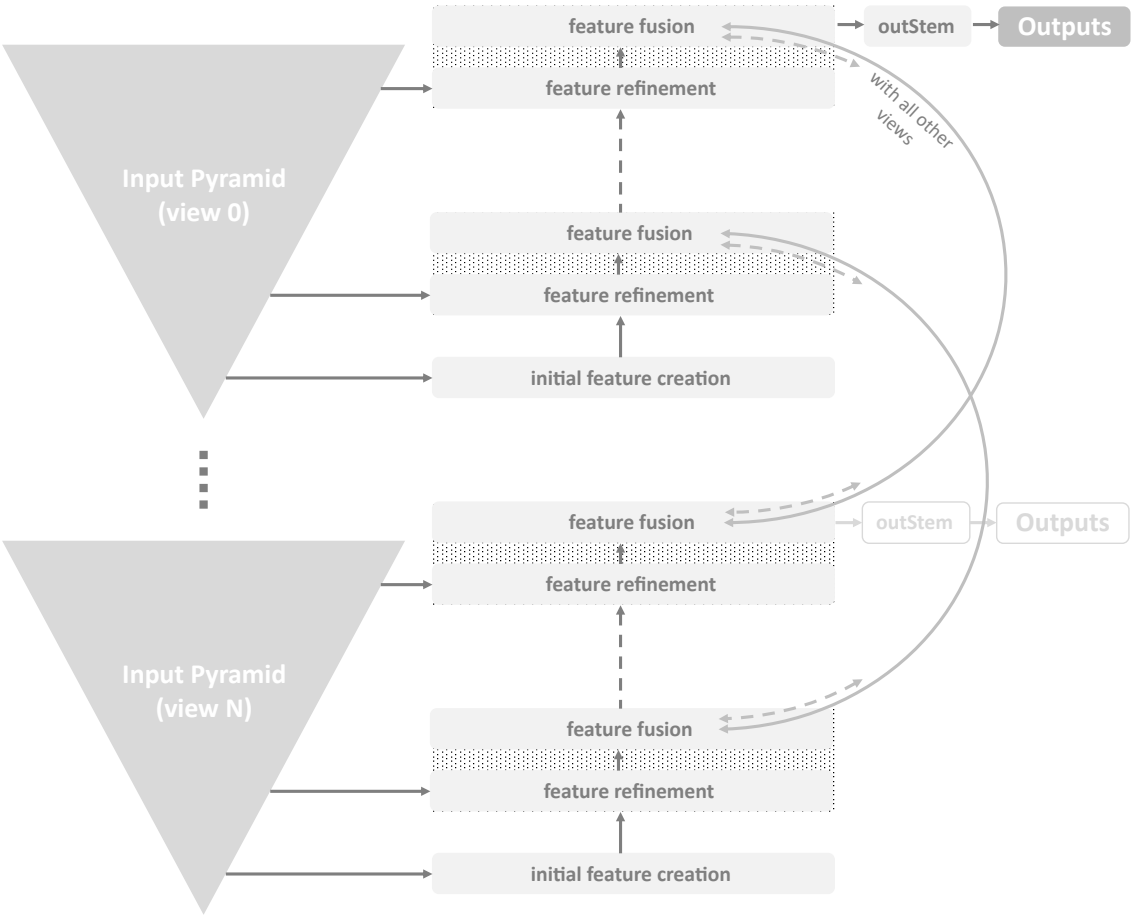


Figure 3.16: High-level overview of the proposed architecture, for estimating a single BSDF parameter. Multiple views of a scan are used together to make a prediction of iteratively increasing resolution. Depending on the number of available views, multiple networks are dynamically spawned, to achieve this. Moreover, their depth is also dynamically adopted, to the input-resolution. Due to feature fusion, all networks produce the same outputs and it can be choosen from an arbitrary view (with a slight exception for *roughness*, which will be explained later in Section 3.2.3.1).

functions. Yet, *core'* could be left out entirely or be used several times (but with individual weights per instance). However, we only used one *core'* block as a compromise between depth and computational effort. The goal of these blocks is, to first compute some hidden features with the *input stem*. These features are then refined with the *core*, which also has access to the original inputs. Moreover, *core'* refines the features further without any additional inputs. Finally, the *output stem* uses these features to compute the desired outputs. To make it work, we need a constant number of channels for the features. We choose 128 – half of the size in [26], to be less demanding on the memory and computing power. This in return allows us to deepen the net with one *core'* block. Besides the number of features, this block is identical to one of the proposed in the *inception* architecture [26]. The intuition is as follows: The input gets directly copied to the output, while a CNN track in parallel just models the changes, that need to be applied to the input to transform it into a good output. This procedure is called residual learning and known to stabilize and ease learning [26]. The mentioned CNN track consists of three combined subnets. The first one has a receptive field of 1×1, the second of 3×3 and the last one of 5×5, so the net itself could choose which filter size is best for its current task. The last ones are composed of 3×3 convolutions, but implemented as spatially separated convolutions of 3×1 and 1×3 for

Figure 3.17: *netCore'*: One of the four basic building blocks of our network. This nets architecture is exactly as *netCore* 3.18, with the only exception, that only the hidden features are used as input. The purpose of this block is to give the network the ability to adjust the features computed by *netCore* further.

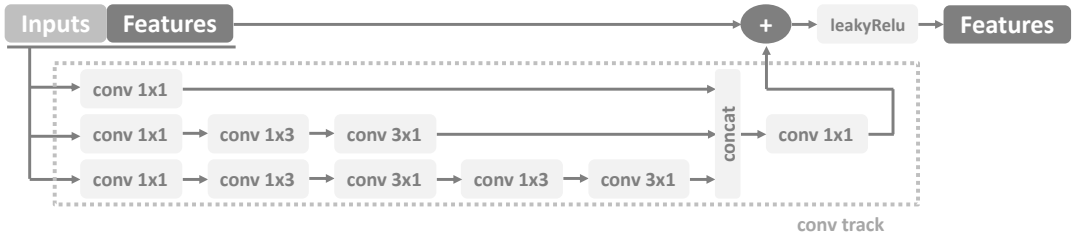


Figure 3.18: *netCore*: One of the four basic building blocks of our network. The intended use of it is, to refine upscaled features of a lower resolution input with the additional information gained from a higher resolution input. This specific architecture block is adopted from [26]. For all our netwoks we use features of 128 channels and work with convolutions 32 channels in- and output, except for the very first and last ones, which are adapted according to the overall number of in- and outputs. The number of these depends on the BSDF parameter we use the network for. Moreover, we use activation scaling (0.1) as suggested in [26] before adding to the original features (omitting the inputs).

computational efficiency. All of these subnets work with 32 channels for in- and output. So, before passing the features (that we seek to refine, consisting of 128 channels) into them, it is necessary to reduce the number of channels. This is done with 1×1 convolutions for each of the subnets at the beginning of the convolutional track. After passing the features through the subnets, their individual results are concatenated ($32 \times 3 = 96$ channels) and the residuals for the features are computed (128 channels again). Moreover, they are scaled (by 0.1 to stabilize learning as suggested in [26]) and added to the original features. Finally, the first non-linearity is applied with a leaky rectifying linear unit (leaky ReLU) with a negative slope of 0.01. This finalizes the *core'* block[3]. Figure 3.17 shows a simplified schematic of it. Figure 3.18 shows a more detailed schematic of the *core* network, which is nearly identical. The only difference is that it has additional inputs (from our 3D-scanner). These extra inputs are ignored in the addition of the features with the residuals, but are used for computing the residuals. Respectively, the only change necessary is, to increase the input size of the 1×1 convolutions at the beginning of the convolutional track accordingly (input size varies between the BSDF parameters and is mentioned later). The *input stem* and the *output stem* as well deviate only slightly more from the previous blocks. Both skip the residual learning, which is not possible here, as we seek to transform the 3D-scanner inputs into the feature set (for the *input stem*) or to transform the features into the outputs (for the *output stem*). All three of these data-tensors have a different number of channels. Therefore, it is necessary like before, to adjust the 1×1 convolution at the beginning of the convolutional track in the *input stem* and the 1×1 convolution at the end of the convolutional track in the *output stem*. This is summarized in Figure 3.19.
Together these four blocks allow us now to build a simple image-to-image translation network.

[3]In addition, our network may also benefit from replacing convolution with *depthwise separable convolution*, as proposed in the *exception* architecture [12] (which is a follow-up work to [26] that inspired our design). *Depthwise separable convolution* tends to reduce prediction quality slightly, but is less demanding on hardware and easier to train. To ameliorate a possible reduction in prediction quality, it might be helpful to use more *core'* blocks. However, due to time constraints, we have not investigated this further.

Figure 3.19: *netInputstem*(top), *netInputstem*(bottom): Two of the four basic building blocks of our network. The *netInputstem* is designed, to estimate some initial features, which can then be further refined by *netcore*. This block does not use residual learning (due to the different in- and output-sizes), but besides that it is identical to *netCore*. The *netOutputstem* is identical, besides in- and output-sizes. Its objective is to transform the features, that were generated by the *netInputstem* and refined with *netCore* and *netCore'*, to the sought BSDF parameter.

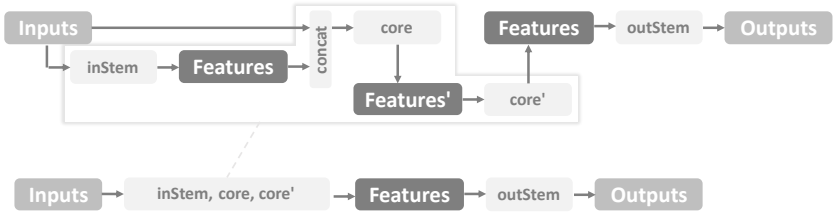


Figure 3.20: An image-to-image translation network. Built, by connecting our four basic building blocks. A similar architecture would be possible by omitting *netCore'*. However, using it gives additional depth and allows more complex funtions to be learned. In our final net we use the *input stem* only once and repeat only the *core* and *core'* blocks. *Features* store the information to retrieve the outputs. *Features'* are of the same shape, and would likewise be able to do so, but are not explicitly trained for it. The second row shows a network, that is identical to the first, but in a more concise notation.

For the goal to transform 3D-scanner inputs to outputs, that represent a BSDF parameter. To do so we first apply the *input stem* to the inputs and receive some initial features. The features are then used together with the original inputs as input to the *core* net, to obtain refined features. Passing them through the *core'* net refines them further and the *output stem* finally can compute the output BSDF parameter. We summarize this in Figure 3.20, which additionally introduces a simplified schematic of it, used to explain the next steps. These follow-up steps are about fusing the features computed in the individual per-view-networks. We fuse the features by grouping them in pairs and using the *max* of each. We iterate until only one feature-set is remaining. If an uneven number of views is given, we hold back its feature fusion until it is possible. However, we avoid holding back one feature-set until the very end and rather balance the fusion-tree. We do this by alternation, leaving out the last or first feature-set (if necessary). This way the fusion-tree does not degenerate, even if the number of feature-sets continues to be uneven after repeatingly applying fusion. Finally, we receive one fused-feature-set. This is then taken to override the originally computed features in all of the networks for each view. The procedure so far is visualized in Figure 3.21 with the simple image-to-image network we defined before. There, the overriding of the old features has no meaning, however, for our actual network we continue refining them in the next pyramid level of the inputs. Therefore, overriding enables one view-network to utilize insights gained from other views in its next refinement step. Due to a fusion step before applying the *output stem*, the results are the same for every view. For a cleaner overview we omit the data flow of the feature-fusion in Figure 3.22 and only show the upscaling process. This is how we use the fused-features together with inputs of increased resolution in order to compute refined features for the current view. Independent of the input-resolution, we start at a high pyramid level of inputs with a very low resolution of 16×16[4] and apply the *input stem* to it. We start at this

[4]For this reason and to avoid misalignment in downscaling, we require input-sizes that are evenly divisible by two until a size of 16×16 or less is reached. For example sizes, that are powers of two. To ensure this, the inputs can be zero padded or cropped, whichever fits better.

high pyramid-level, to have a chance to learn global features. With a perceptive field of 5×5 the *input stem* can easily learn from very big regions in the image, but without any fine details (for example the mean color of the image). We increase the details iteratively by applying *core* (and *core'*) to the inputs in the next level of the pyramid and supported by the current features (fused as described before and upscaled via nearest neighbor). As last step the features are passed to the *output stem*, which then computes the BSDF parmeter. This completes the network-architecture of a single view. When working with multiple views, one such net needs to be spawned for each view and fusion must be executed as described before.

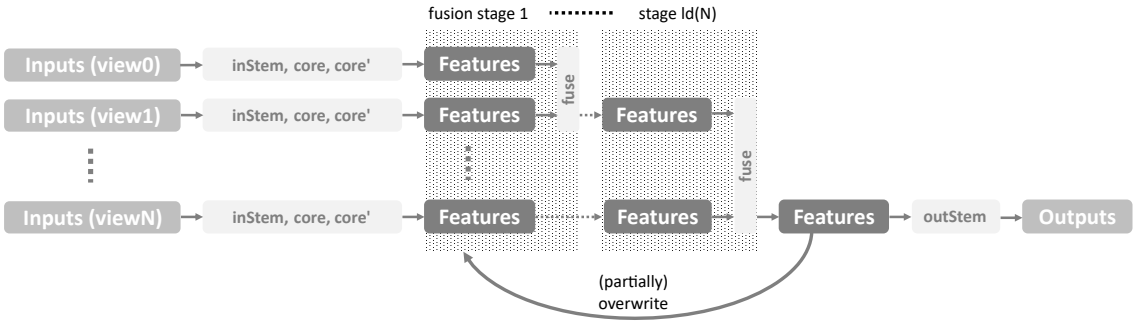


Figure 3.21: Fusion of multiple networks as defined in Figure 3.20. Again, in our final net we use the *input stem* only once at the highest pyramid level. We fuse the features created from all the views pairwise, until only one feature-set is left. Fusion is done by simply taking the *max* values. The combined features are then used to update the features in the networks of the individual views. For all BSDF parameter-networks (except *roughness*/illumination) updating means simply replacing them. For the latter we proceed differently. We describe this later in Section 3.2.3.1. Updating is done, because in our final net (unlike in this toy example) we continue working with these features.

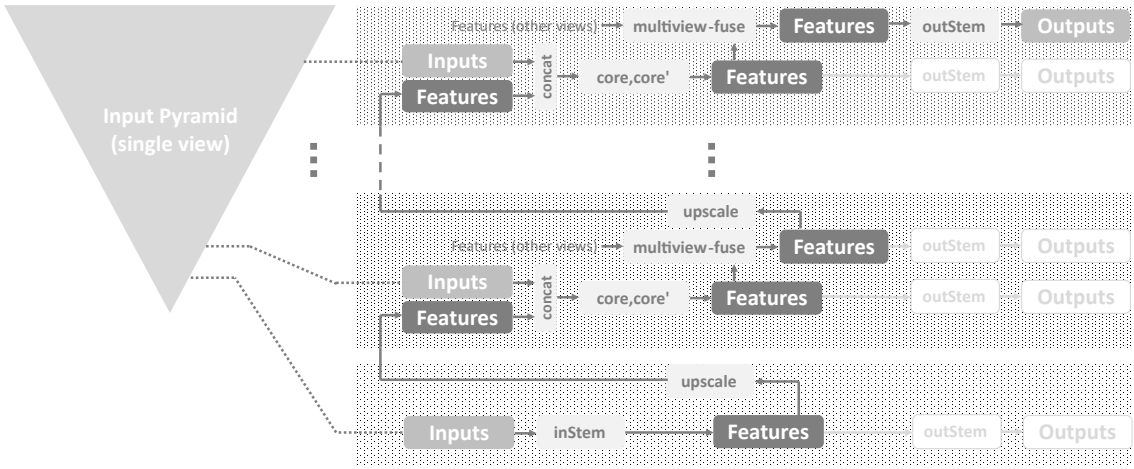


Figure 3.22: An overview (considering only a single input-view) of the proposed network architecture: The network first predicts some features from the highest pyramid level of the inputs. These features could already be used (with the output-stem) to infer the sought BSDF prameters. However, we refine them further. First by fusing the current knowledge with the one gained from other views. Second by using information of the input in the next lower pyramid level. We iterate this procedure until we reach the lowest pyramid-level. Applying the output-stem to the features, we get BSDF parameter predictions in the original resolution of the input, harnessing the insights of all given views, with and receptive field of the whole input-resolution.

3.2.2 Training

In theory the architecture that was described in Chapter 3.2.1 allows to infer BSDF parameter predictions for each level of the input pyramid. For inference this is not really interesting, but for training we make use of it. The natural way of training the net would be to compute the lowest level (highest resolution) outputs, compare them to the ground-truth BSDF parameters and adjust the weights of the net accordingly. To accelerate learning we also try to learn the translation of the input to the BSDF parameters only with the *in-* and *output stem*. We do this for the first 100 batches. Moreover, for the first 2000 batches we also train with the full resolution but only for a single view. Afterwards we only train with a pass trough the full pyramid (i.e. with the highest resolution outputs) and with all available views and also with a random selection of pairs of all possible views. Depending on the data set, in most cases this was three positions (but in some it was up to 13) and three lights. This makes nine views for most scenes. Until the previously stated number of batches we use each of the named strategies for learning once and then continue by repeating each again. For all of the previous strategies and all of the different BSDF parameter-networks we use a learning rate of 0.001 and the *Adam* learner, which adjusts the learning rate for the weights individually while training. The individual loss functions are provided in the subsequent sections. Training durations and results will be listed in Chapter 4. As input to training we use the renderings created in Chapter 3.1. Most of them are used for all five of our networks, but there are slight variations of which images are used for which network. We provide details in the following. However, identical in all of our networks is, that we always use some augmentations. These include a zero-centering in $[-1, .., 1]$ and a random dropout-like removal of some of the input pixels.

3.2.3 Individual In- and Outputs, Architecture Deviations and Loss Functions

In the following we describe for each of the BSDF parameter-networks the adjustments that are necessary compared to the general network, that we described before. For each parameter, we split the description in architecture changes, differences in inputs and the individual loss functions.

3.2.3.1 Roughness

Architecture Changes Of all the BSDF parameters, the architecture for *roughness* deviates the most from the generic network. This is, because we are not only trying to estimate the *roughness* values with this network. More so, we are also trying to derive an image that shows the same scene as the original, but consists only of a white diffuse material. This result would implicitly provide an estimate of varying local illumination. We have two reasons for this type of parallel learning. First, the features learned for this task could also be useful for inference of *roughness* (shown in Chapter 4.4) and second, for the overall predicted material appearance. In addition, knowledge of local illumination also supports the inference of the *basecolor* subsequent to *roughness* (as shown in Chapter 4.2). Outgoing illumination (provided by the captured scene) and ingoing illumination information (given by the previously defined prediction), together with the known light- and viewangles provide all the information, that is necessary to define a BSDF. This makes learning *roughness* possible. The angles are known from the normals, view- and lightvectors provided by the 3D-scanner and computed as discussed in Chapter 3.1. Alternatively, the available inputs would also allow for another (probably easier to learn) approach. Instead of learning the local varying illumination directly it would also be possible to just learn the overall lightintensity and then use the known geometry and light location to re-render the scene with a diffuse white material. It is likely to give better results, but due to ease of implementation we favored the first approach of learning the local illumination directly. To learn both the *roughness* and the diffuse white object, we need to adopt our net slightly. First we need to split the features into two parts, one for *roughness* and the other one for the diffuse white object. This is necessary, because the *roughness* does not depend on the light conditions of the current view, whereas the render

of the diffuse white object (depicting the local illumination) clearly does. This in return means, that we can not fuse the features as before, but rather need to allow each per-view-network to maintain some features that are independent of the other networks. For said reasons we define the mentioned split of the features. Moreover, we change the fusion procedure in that way, that we only overwrite the first half of the the features from a single view, with the fused ones. As a result, the first half of the features are independent of the view, whereas the second half is not. Accordingly, the first half of the features is meant to store information about *roughness*, whereas the second half stores information about the diffuse white object. Correspondingly we adopt the output stem (as visualized in Figure 3.23). We introduce a second convolutional track (which is exactly the same as the first one, but with its own independently learned weights). Now each track handles one half of the features and one of the output channels. The *output stem* has therefore no possibility to use features dedicated for one output to compute the other output with. This means, that after each (and especially the last) fusion the outputs for the *roughness* are the same for all views. However, in contrast the outputs for the diffuse white object are computed from the view-depended-features and therefore the outputs are individual. Even though there is no way to exchange information of the two outputs in the *output stem*, the net still has this ability. It can achieve this via the *core* or *core'* blocks, as we use the generic ones defined earlier. These blocks only have one convolutional track and it takes all features as input and computes the refinement for all of them. The only remaining modifiaction is to adjust the in and output sizes. For output we have two channels, one for the *roughness* and the second one for illumination. For input we use the twelve channels listed below.

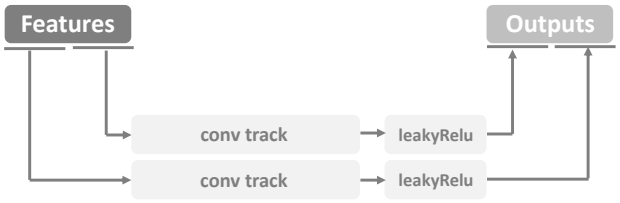


Figure 3.23: The modified output stem, used to infer *roughness* and local illumination. Prediction is split in two separate tracks, working on their individual halves of the features.

Inputs As stated before, to ease learning and to reduce computational cost, we seek to use only meaningful inputs. These inputs are just as defined in Chapter 3.1, if not stated differently. Like for any other of the networks the first input is the mapped-mask, followed by a picture of the ambient lit scene. However, like for most of the other networks we only use grayscale images here. For the third channel, we use an image of the scene lit with one light in addition. This can be either one of the flashes or the projector, showing constant gray/white. We do not use this image directly, but the square root of the clipped difference between the ambient and the photo with the additional light. This difference is always zero or positive, when ignoring noise. We apply the square root, so that specular highlights do not overwhelm the image and that less illuminated parts have a chance to contribute to the output as well[5]. In the next three channels we provide the cosine (for computational efficiency) of the normal-to-light, the normal-to-view and the view-to-light angles. We computed them from the normals, view- and lightvectors given by our training dataset / the 3D-scanner. We make this transformation to save six channels[6] and to have a representation that is more closely related to the inputs of a BSDF. To support the net in estimating the illumination in the scene, we further provide a layer filled with the constant value of the mean response of a white diffuse object in the scene. For training we used the turned off backlight as the white diffuse object. In a real-world application one might do the same or just use a sheet of paper close to the object of interest as a reference. Moreover, we also provide

[5]For exactly this reason, nonlinear scaling the inputs is a common practice in this field of research [15, 14]
[6]If the normals, view- and lightvectors were used, nine channels would be necessary instead of just three for the respective angles.

the difference of this in the ambient image compared to the additionally lit scene in the following channel. Next we provide the depthmap, with the intention to guide the detection of shadowed regions. Last we provide the estimated (or while training the ground truth) values for *metallic*, *transmssion* and *subsurface*. Finally, all the mentioned images are stacked and feed as inputs to the network. However, care must be taken with *transmission* and *subsurface*, because if used as input values to the shading program, they might be overridden by other parameters. The actually used final-transmission T_{final} is computed as in Equation 3.5. We therefore provide the net with the final-transmission, instead of the (user defined) ground truth transmission input T_{input} (from our dataset). For its computation we only need the ground truth metallic M_{input} in addition.

$$T_{final} = T_{input} \cdot (1 - M_{input}) \tag{3.5}$$

As defined in the source-code [3], *subsurface* behaves similar. The actually applied and rendered *subsurface* value S_{out} is as defined in Equation 3.6:

$$S_{out} = S_{input} \cdot (1 - M_{input}) \cdot (1 - T_{input}) \tag{3.6}$$

Again we use this value instead of the ground truth for input in the network.

Lossfunction The loss for this net is basically a mean square error (MSE) between predictions $\hat{Y}$ and ground truth values Y. However, we weight individual pixels of both outputs differently. For illumination we desire to optimize the output with respect to all valid pixels (i.e. where 3D-scanner inputs are available). For this reason we compute MSE only for the regions that are permitted in the mapped-mask M_{mapped}, which is defined as the object mask (Figure 3.4n), observed from the current view. Or in other words, pixels in the matchmap (Figure 3.8) that point to permitted ones in the object mask of the used position. Finally, we divide the summed errors of the used pixels by their number and by the batchsize b.

$$Loss_{illumination} = \frac{\sum_{pix}(M_{mapepd} \cdot (Y_{illum} - \hat{Y}_{illum})^2)}{\sum_{pix}(M_{mapped}) \cdot b} \tag{3.7}$$

In a similar manner, we compute the loss for *roughness* (Equation 3.8). However, as weights we use the ground truth illumination provided by the white object, that we rendered in our dataset before. The reason for this is, that we do not want to punish the network for false predictions, in regions where it has no chance to do well. For example shadowed areas. Without any further measures, the downside of this would be that the network might not learn to infer meaningful prediction, even when it would have the chance to do so by the information given from other views. To cope with that, we extend our fusion step. Instead of just fusing the features, we also fuse the ground truth data, that is used for learning only. Precisely, we fuse the render of the diffuse white object (that represents illumination) M_{illum} as well. Again we use max for this. Therefore, a network using multiple views has to optimize its predictions in regard to the best available illumination it gets as input.

$$Loss_R = \frac{\sum_{pix}(M_{illum} \cdot (Y_R - \hat{Y}_R)^2)}{\sum_{pix}(M_{illum}) \cdot b} \tag{3.8}$$

Our final loss for this network (Equation 3.9) is then simply the sum of both previous losses. We reassure the usefulness of the illumination loss in Chapter 4.4.

$$Loss = Loss_R + Loss_{illumination} \tag{3.9}$$

As mentioned before, we use *Adam* to optimize it. Briefly said, it is a fancier variant of *gradient descent*, that uses individual adaptive momentums for the weight updates, to learn faster.

3.2.3.2 Transmission

Inputs In regard to the network architecture, only the in- and output sizes need to be adjusted. Besides that, the *transmission*-network is identical to our generic one. The inputs for this network are five channels and for outputs we only have one for transmission. However, as explained previously input parameters to the shader are not necessarily what is rendered. Therefore, we again adjust the ground truth input-roughness, to what is actually observable. Due to how the sub-shaders are mixed (Equation 3.5) and our assumption, that a material can either be *metallic* or not, it follows that metallic areas can not be transparent. We therefore override ground truth *transmission* accordingly. Note that the modified values will not cause any differences in rendering compared to the original values. As inputs for this net we require less parameters than for *roughness*, but require the backlight as an additional tool. We provide the same first three input channels as in the *roughness* network. Namely the mapped-mask, and the picture of the only ambient lit scene in the first two input-channels. Followed by the square root of the difference of the ambient image to an image of the scene illuminated with an additional light. The next channel is for the backlight image, it is preprocessed as the one with the additional light (i.e. the square root of the difference). Last we input the cosine of the normal-view-vectors, but not the NL or LV angles. We refrain from having the later two as inputs, so that an uncalibrated light source could be used for transmission as well. We expect the main information to be provided by the difference-backlight-image anyways. The NV angles however might support the net in avoiding false positive transmission. In areas with grazing angles, the fresnel effect causes the object to reflect light from the back to the observer and leads to an appearance, that is similar to the one of transparent ares. The known NV angles allow to identify these areas.

Lossfunction For the loss, we again rely on the MSE, but this time the main source of information is very likely the backlight. Problematic areas for this net are, the previously discussed sides of an object, but also areas where the opaque backside blocks light, that otherwise would pass through a transparent front. To cope with the latter we have no options, but to use multiple views and hope for at least one that is suited for identification of the *transmission* value. However, we do not have a meaningful way to weight the loss in these areas (like the illumination loss in roughness). We therefore simply use the mapped-mask. Meaning we give an equal weight to the loss in all areas, that belong to the object. We therefore just use the MSE of the prediction $\hat{Y}$ to the ground truth roughness Y, but we also add an extra punishment for false positive transmission (weighted by 0.5, so that it does not dominate the other part of the loss). We do this by the assumption that most materials are actually non-transparent and because false positve *transmission* can cause a notably visible change in the re-render. More over, but less important, rendering transmission adds additional computational cost. Our final loss function for *transmission* is therefore as defined in Equation 3.10 for a batch of size b.

$$
\begin{aligned}
n &= \sum_{pix} M_{mapped} \\
Loss_{fp} &= 0.5 \cdot \sum_{pix} \frac{fp}{n} \quad \text{for} \quad fp = \begin{cases} 1, & \text{if } \hat{Y}_T > 0 \text{ and } Y_T == 0 \\ 0, & \text{otherwise} \end{cases} \\
Loss_T &= \frac{\sum_{pix} (M_{mapped} \cdot (Y_T - \hat{Y}_T)^2)}{n} \\
Loss &= \frac{Loss_{fp} + Loss_T}{b}
\end{aligned}
\tag{3.10}
$$

3.2.3.3 Subsurface

Inputs Again, only the in- and outputs need to be adjusted compared to the generic network architecture. For input we have seven channels and one for output. As before, when learning from the training-set, we do not seek to infer the ground-truth *subsurface* values set by the user

S_{input}, but rather the rendered ones S_{out}. To compute S_{input}, that defines the desired net-output, we again use Equation 3.6. It is based on the user set *metallic* M_{input} and *transmission* T_{input} values. We use the same input as for *transmission* and in addition T_{final} (the rendered *transmission*), as defined in Equation 3.5. Moreover, we provide the net an image where the (potentially uncalibrated) projector shows a checkerboard pattern. All of the inputs are grayscale images again. The checkerboard pattern is meant to be the main source of information for this network. It is not meant to have any specific size, as this would not be easy to control. The network itself should learn to identify edges in the pattern and depending on their blurriness determine the subsurface values.

Lossfunction We compute the loss for *subsurface* (Equation 3.11) in the same manner as the loss for *transmission* (Equation 3.10) and for the same reasons as mentioned there. However, we use the illumination map again (only for computing the). Further, we only train with the projector as light, so that the lightvectors (or rather the angles computed from them) are valid for the checkerpattern as well. It is the only difference to the *transmission*-loss.

$$
\begin{aligned}
Loss_{fp} &= 0.5 \cdot \frac{\sum\limits_{pix} fp}{\sum\limits_{pix} M_{mapped}} \quad \text{for} \quad fp = \begin{cases} 1, & \text{if } \hat{Y} > 0 \text{ and } Y == 0 \\ 0, & \text{otherwise} \end{cases} \\
Loss_S &= \frac{\sum\limits_{pix} (M_{illum} \cdot (Y_S - \hat{Y}_S)^2)}{\sum\limits_{pix} M_{illum}} \\
Loss &= \frac{Loss_{fp} + Loss_S}{b}
\end{aligned}
\tag{3.11}
$$

3.2.3.4 Metallic

Inputs For *metallic*, we use the generic net as well. The in- and output sizes for this net are ten and two. As output we seek to learn *metallic* M and its inverse M_{inv}. We do this in the hope to improve prediction quality with it. This time we rely on color information for the inputs, because metallic is characterized by tinted reflections. These are supposed to be our main source of information in the inference. We therefore provide the following inputs. The first nine channels are the same as in *transmission* and *subsurface*, however, this time encoded in four more channels due to the color information, which is provided in the YCbCr color space. This color space separates luminance and chroma information. Especially the latter might be helpful to detect colored highlights. To ease the detection of these reflections even more, we additionally input an difference image of the ambient lit scene and the scene lit by an additional light. Moreover, we apply the square-root function to it, to downscale the response of specular highlights compared to other areas.

Lossfunction For the loss, we compute the MSE and use a different pixelwise-weight w again. We keep the illumination part, but add the inverted difference of the luminance channel of the ambient image and the one with the additional light. The second part is for putting a higher emphasis on areas that are not highlights. We focus on these areas, as we suppose them to be the harder ones to learn. The loss formula is finally defined as Equation 3.13. The sum of a loss for the prediction of *metallic* areas and a loss for non-*metallic* areas. In addition everything is scaled by 0.5, for a more meaningful comparison in 4.3. To anticipate, what we found there: Contrary

to our implementation, we recommend to prefer Equation 3.12 as loss over Equation 3.13

$$W = M_{illum} \cdot (1 + luminance(1 - (ambiL - ambi)))$$
$$Loss_{M_{inv}} = \frac{\sum_{pix}(W \cdot (inv(Y_M) - \hat{Y}_{M_{inv}})^2)}{\sum_{pix} W \cdot b}$$
$$Loss_M = \frac{\sum_{pix}(W \cdot (Y_M - \hat{Y}_M)^2)}{\sum_{pix} W \cdot b} \tag{3.12}$$
$$Loss = 0.5 \cdot (Loss_M + Loss_{M_{inv}}) \tag{3.13}$$

3.2.3.5 Basecolor

Inputs In terms of architecture the network for *basecolor* is no exception. As for all other BRDF parametes (except *roughness*) we stick to the generic network architecture. This net has twenty-one input channels and three for the YCbCr encoded color-outputs. However, as mentioned before, we do not really seek to learn the basecolor C_B (which is impossible when *subsurface* S is high). And vice-versa we are not interested in the *subsurfacecolor* C_S. Rather we learn the inference of the *mixed subsurface basecolor* C_{SB}, which is the actual input to the used shaders. It is computed as shown in Equation 3.14. The seperation of it in subsurface- and basecolor has benefits in artistic tasks but again, seperating them is not possible for us. When re-rendering with the *Principled BSDF*, we therefore insert the inferred *mixed subsurface basecolor* for both instead. According to Equation 3.14 this makes no difference to the intended inputs.

$$C_{SB} = C_S \cdot S + C_B \cdot (1 - S) \tag{3.14}$$

So, C_{SB} encoded in the YCbCr color space is, what we try to learn. We use the YCbCr color space, because it is perceptually uniform. This means, that equal value-changes in it cause equally strong perceived changes in human vision. This is not the case with color spaces in general. So when MSE is applied later, the loss automatically adheres to this perceptual uniformity. As inputs, we again take the mapped-mask, the ambient lit image and the image with the additional light. Where the latter two are also in the YCbCr color space. In addition we provide the cosines of the NL, NV and VL angles again. Followed by the response of a diffuse white material under ambient illumination and under ambient illumination with the additional light. Both we encode in six channels as YCbCr. So far the inputs are quite similar to the ones for *roughness* and we also provide *metallic* and *transmission* (T_{final} as in Equation 3.5) in the last two input channels. Different to before, we now also provide *roughness* values and the illumination map as inputs. Moreover, we also provide the inverted illumination map, so that the net itself can choose its preferred representation. The latter could for example, help to facilitate learning how to brighten shaded areas, whereas the former could help to identify areas that are already too bright.

Lossfunction Pointed out before, we again use the MSE with an individual per-pixel-weight. It is similar to the weight in the loss for *metallic* (Equation 3.13). However, this time we actually want to focus training on the specular highlight, because these are the areas where the color is difficult to detect. Hence we use the illumination together with the difference of luminance of the two differently lit photos as weight. The remaining loss is defined as before – shown in Equation 3.15:

$$W = M_{illum} \cdot (1 + luminance(ambiL - ambi))$$
$$Loss = \frac{\sum_{pix}(W \cdot (Y_{C_{SB}} - \hat{Y}_{C_{SB}})^2)}{\sum_{pix} W \cdot b} \tag{3.15}$$

Chapter 4

Results and Further Discussion

4.1 Training History

Parallel to training the networks, we also monitored their performance after every hundredth batch. The measurements are depicted in Figures 4.4 - 4.7. For this, we computed the loss on a dataset, that was specifically created for this task. It was never shown to the respective networks, but used as a guidance to adjust the network and for selecting hyper-parameters in the design process. This set contains 41 scenes with all the variants of the different materials used in training. Some scenes of it are shown in Figure 4.1. This test-set is relatively small compared to the training-set. We did not use further scenes because of the costly creation process. However, we manually selected some scenes to ensure, that the test-set approximately covers the diversity of the training-set. An indication, that this is the case, is, that the monitored training-/testing-losses have very similar plots. Moreover, this is the case even though that the training-set is much bigger. The only difference, is that errors on the training set are slightly lower, than in the testing-set (as expected). To show the similarities between training- and testing-loss, we also plotted the history of the training-loss for *roughness* in Figure 4.2. To avoid redundancy, we skip the plots of training-losses for the other BSDF parameters. A possible alternative to this testing-strategy, would also be the usage of *cross validation* (i.e. splitting the training set into several chucks and using one of them in an alternating manner for testing, but at that time not for training). Coming back to the test-losses: All plots show, that all networks do learn and finally also converge after some time. In detail we trained the networks for the following durations:

- *roughness*: 5d 19h
- *basecolor*: 5d 3h
- *metallic*: 5d 1h
- *transmission*: 1d 10h
- *subsurface*: 1d 7h

We trained all of the nets on a NVIDIA GTX 1080ti, each for 150 epochs. Differences in the number of batches given in the plots stem from slightly varying training-set-sizes and different batch-sizes.

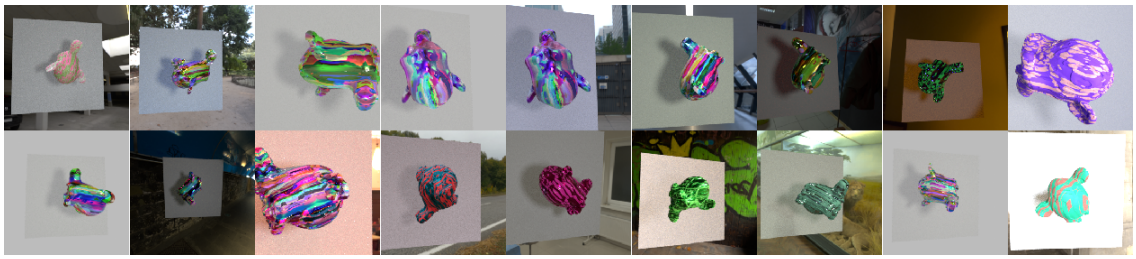


Figure 4.1: Some of the 41 scenes used for testing. The picture shows the ambient + flash1 lit render of the first position. The test-set consists of the different dataset-types *multimat* and *2mat*.

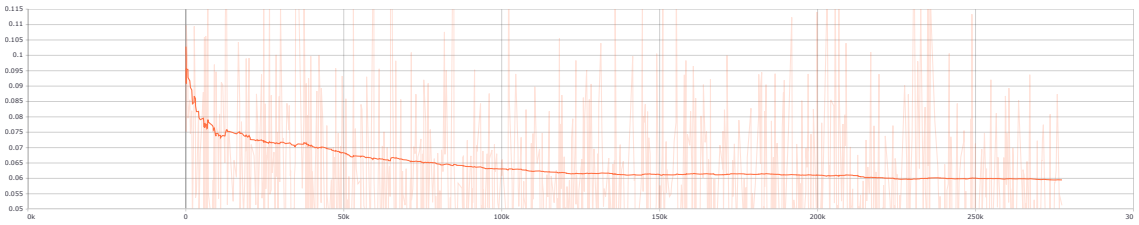


Figure 4.2: *Roughness*-network performance on the training-set – monitored over the training duration. The y-axis shows the loss, the x-axis the batch count.

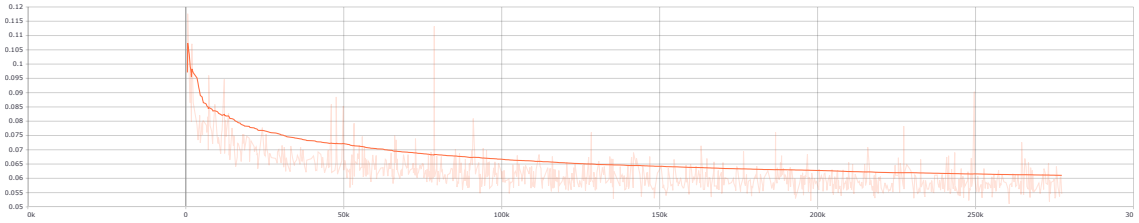


Figure 4.3: *Roughness*-network performance on the test-set – monitored over the training duration. The y-axis shows the loss, the x-axis the batch count.

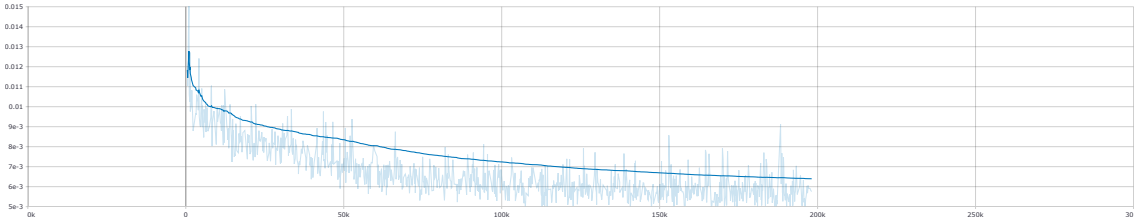


Figure 4.4: *Basecolor*-network performance on the test-set – monitored over the training duration. The y-axis shows the loss, the x-axis the batch count.

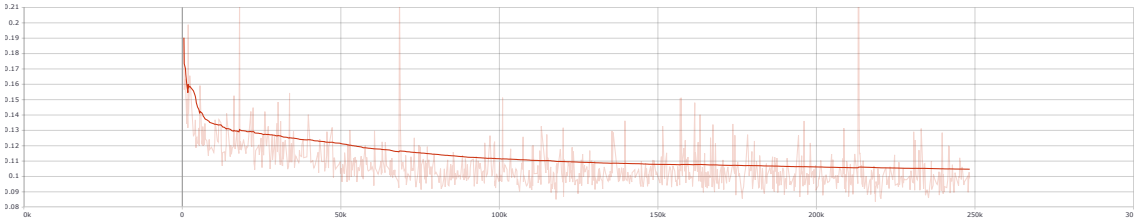


Figure 4.5: *Metallic*-network performance on the test-set – monitored over the training duration. The y-axis shows the loss, the x-axis the batch count.

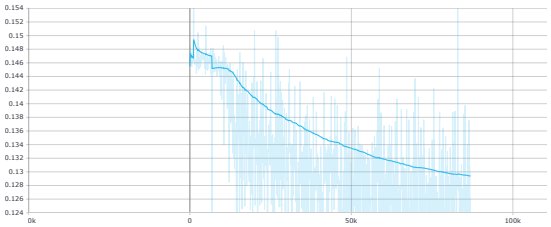


Figure 4.6: *Transmission*-network performance on the test-set – monitored over the training duration. The y-axis shows the loss, the x-axis the batch count.

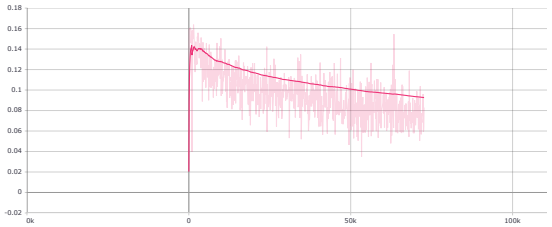


Figure 4.7: *Subsurface*-network performance on the test-set – monitored over the training duration. The y-axis shows the loss, the x-axis the batch count.

4.2 Basecolor Improvements with the Illumination Map

To verify, that the illumination map provided to the network is beneficial for inferring the *basecolor*, we trained a duplicate of the same net. We modified this duplicate in a way, that we deny it access to the illumination information as input. This means both networks are identical with the exception, that the modified one only has one input-channel less. The loss of the modified network (that is used for training and plotted in Figure 4.8(light gray)) is nonetheless still computed with the illumination as weight. It is calculated for both network variants in the same manner (as explained in Section 3.2.3.5). In Figure 4.8 we show the history of the performance on the test-set for both network variants. It is observable, that at all times our default network yields a lower loss. This means, that the illumination indeed provides valuable information. Therefore it should be kept as input. Because this circumstance is obvious early on in training, we stopped early and did not waste resources to compute all 150 epochs (like we did for the original variant).

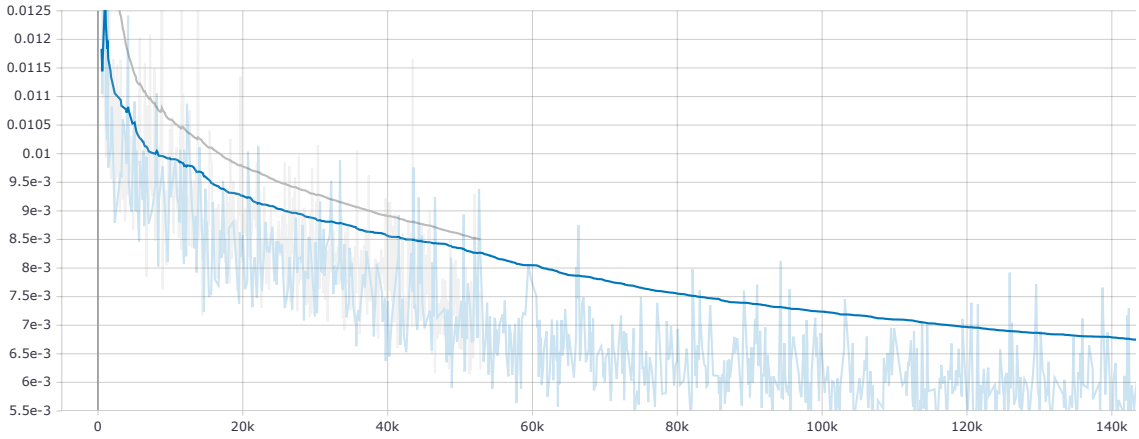


Figure 4.8: Training History: Test-set Losses for different networks, intended for the inference of *basecolor* x-axis: batch number. y-axis: loss. Blue: the net, as defined in Section 3.2.3.5 for predicting the *basecolor*. Light gray: the same net, but without access to the illumination-information as inputs.

4.3 The (Non-)Metallic Loss

For training *metallic*, our loss and outputs are composed of two parts (as described in Chapter 3.2.3.4). One part is the *metallic*-parameter-map and the other is the inverted version of it. The

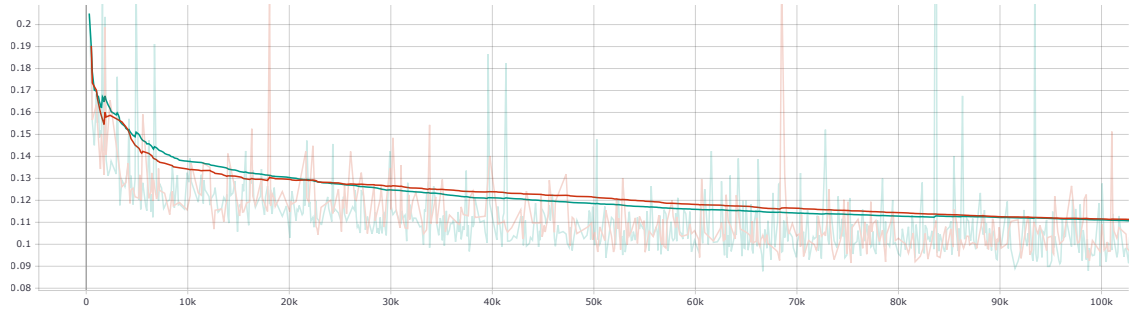


Figure 4.9: Training History: Test-set losses for different networks, intended for the inference of *metallic*. x-axis: batch number. y-axis: loss. Red: the net, as defined in Section 3.2.3.4 for predicting both *metallic* and not-*metallic*. Green: the net without the no-metallic loss.

loss is the summed MSE of both divided by two. To test the usefulness of this combined loss approach for our task, we trained the exact same net, but without predicting and optimizing for the inverted version (and likewise skipping the scaling of the loss). The test-losses are shown in Figure 4.9. It turned out, that the strategy of learning the inverse does not result in notable benefits (nor drawbacks) to our problem. At best one could argue, that the described loss formula slightly speeds up the early learning phase. However, to save resources we would recommend to prefer the simpler loss (without the inverted *metallic*-parameter-map).

4.4 Discussion about the Roughness-Network

Benefits from the Illumination Loss In Section 3.2.3.1, we defined the inference (and training) of *roughness* and illumination in a combined manner. We created a single network, that performs both tasks at once. Alternatively, it would also be possible to create two separate networks, each taking care of one of the parameters (either illumination or *roughness*). However, we argue that it is beneficial for the prediction quality of both, to train/predict both values in a combined manner. To confirm this assumption, we created two networks, by splitting the original into two parts. These parts are identical to the ones, that are used in the original net, but without their interconnections. In detail, we created the two networks from the original *roughness*-net (as defined in Section 3.2.3.1) as follows. For each of them, we split the hidden features in half and further modified the output-stem to only infer one parameter-channel. The output-stem is therefore defined like for all other BRDF parameters. For the modified net, that is supposed to only predict *roughness*, we fuse the features (in each pyramid-level) by completely replacing the view-dependent ones with the fused ones. We do so, because the *roughness*-parameters are view-independent. This is in accordance with the design of the original net. With the same motivation, we do not fuse any features for the second modified network, that is supposed to only infer the completely view-dependent illumination. Figure 4.10 shows the test-loss of all three variants. The loss of the original network (red) is the sum of a *roughness*- and an illumination-loss. Naturally, it is bigger than the losses of the modified networks, that each only works with their individual part of the original loss. For a more meaningful comparison we also extracted the individual *roughness*- and an illumination parts out of the original network. In Figure 4.11 we compare the *roughness*-loss of the original net (red) to the loss of the modified net, that only predicts *roughness* (light blue). Similarly, we compare the illumination-loss of the original net (red) to the loss of the net, that is only supposed to learn illumination (blue), in Figure 4.12. For both cases the original method outperforms the modified networks. This means, that combined learning of both values is indeed beneficial to the quality of both.

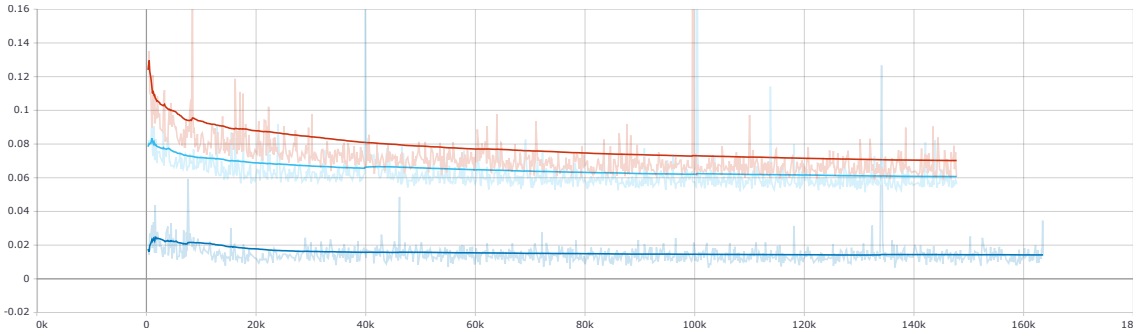


Figure 4.10: Training History: Test-set losses for different networks, intended for the inference of *roughness* and illumination. x-axis: batch number. y-axis: loss. Blue: a net that only predicts illumination. Light blue: a net that only predicts *roughness*. Red: the net, as defined in Section 3.2.3.1 for predicting both *roughness* and illumination. Its loss is a sum of illumination- and *roughness*-loss. Naturally, it is larger than the losses of the other two nets.

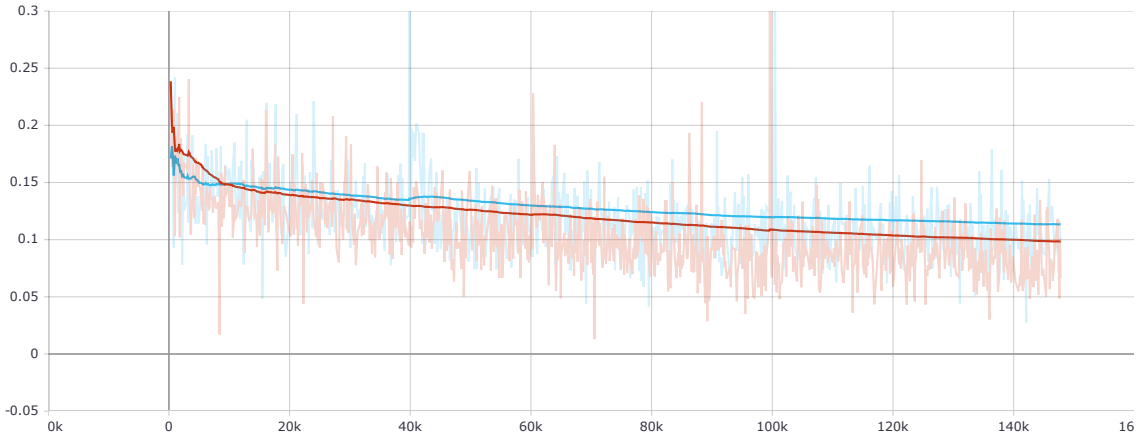


Figure 4.11: Training History: Test-set losses for different networks, with regard to *roughness*. x-axis: batch number. y-axis: loss. Light blue: a net that only predicts *roughness*. Red: the net, as defined in Section 3.2.3.1 for predicting both *roughness* and illumination. However, only the *roughness* part of the loss is shown. This sub-loss is defined exactly like the loss of the other net.

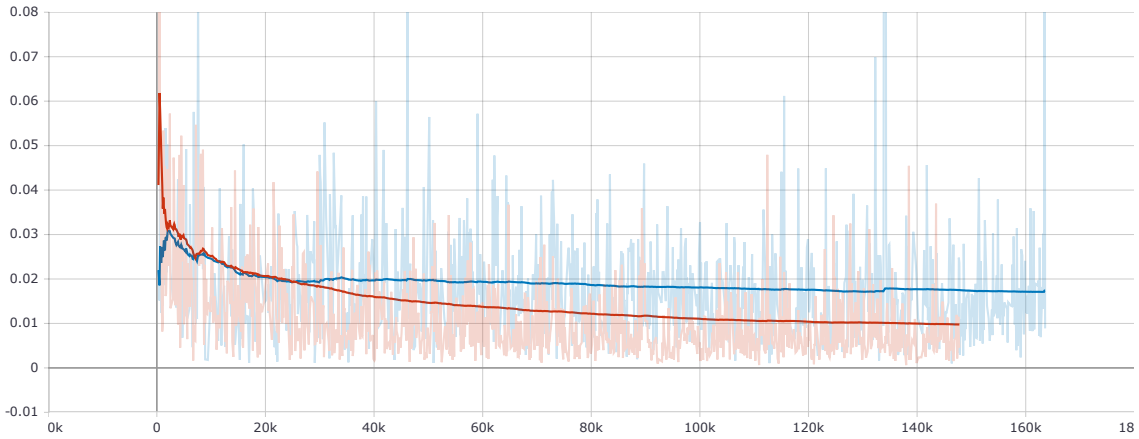


Figure 4.12: Training History: Test-set losses for different networks, with regard to illumination. x-axis: batch number. y-axis: loss. Blue: a net that only predicts illumination. Red: the net, as defined in Section 3.2.3.1 for predicting both *roughness* and illumination. However, only the illumination part of the loss is shown. This sub-loss is defined exactly like the other net's loss.

Number of Hidden Features From the previous paragraph, where we modified the number of hidden features in accordance to represent different networks, another question arises. What is a good number of hidden features? We did not fully explore this question, but to get at least some insight we trained a net like the one of the previous paragraph, that only infers *roughness*. However, this time with double the number of hidden features. This means, it has the same number of hidden features as the original network again. In Figure 4.13 we compare the losses of this modification (light green) to the losses of the previously described net (light blue). For a reference we also added the loss of the original net (red). Again this (red) is a sum of two sub-losses and not comparable to the others. What is interesting however, is, that increasing the number of hidden features still lead to a better performance. Moreover, the net, that uses double-sized hidden features, but no illumination loss, would outperform the original net (whose loss is not plotted here) as well. This means that increasing the number of hidden features is beneficial and with the current number of features, even more beneficial than using the illumination loss.

However, this does not mean that the illumination loss is not meaningful. We explicitly showed that before. This only means that a better number of hidden features for the *roughness*-part must be found. Or more generally speaking the trade off in the ratio of features used for illumination compared to *roughness* needs further tweaking. As explained in Chapter 3.2.3.1, we simply used a split in half. This experiment however indicates, that dedicating a bigger fraction to *roughness* seems reasonable. Due to time and computation-costs, we did not explore this further.

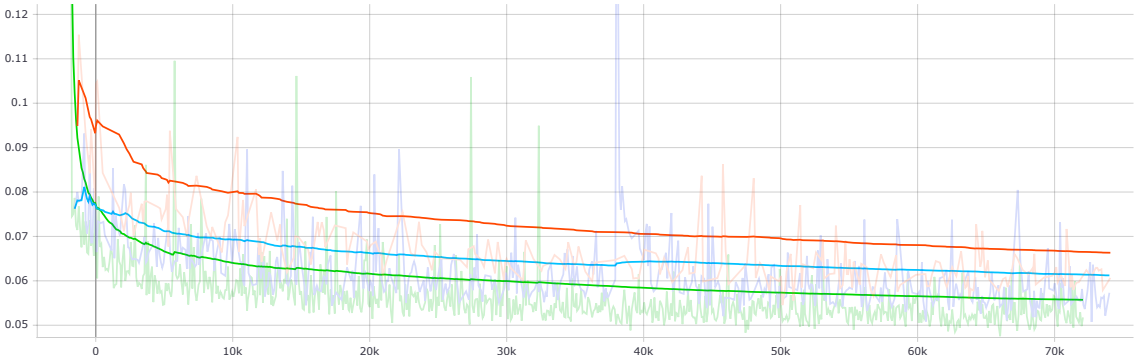


Figure 4.13: Training History: Test-set Losses for different networks, intended for the inference of *roughness* x-axis: batch number. y-axis: loss. Red: the net, as defined in Section 3.2.3.1 for predicting both *roughness* and illumination (with 128 hidden features in total, used for both). Light blue: a net that only predicts *roughness* (with 64 hidden features). Light green: a net that only predicts *roughness* (with 128 hidden features).

4.5 Evaluation Dataset

To evaluate the performance of our network, we created some additional datasets. None of them was shown to the net while training, nor were they used for testing at training-time to tweak the network further. These datasets were created with the different geometries *Ape*, *Armadillo*, *Dragon*, *Bunny* and *Sphere*, that had been introduced in Figure 3.5. For creation, we used the methods *multimat*, *2mat* and *3mat*, as described in Section 3.1.7. With the exception of skipping the displacement modification for the *Bunny*. In total we created 61 further scenes for evaluation.

Figure 4.14: Some of the 61 scenes used for evaluation. The picture shows the flash1 lit render of the first position. The evaluation-set consists of different dataset-types (*multimat*, *2mat*, *3mat* for varying geometries).

Some of them are shown in Figure 4.14. To measure the performance of our five networks, we computed some error metrics of the results. To do this, we evaluated them independently, meaning that we used the ground truth of the required BSDF parameters as inputs instead of the values predicted by the otherwise preceding networks. For the dependency chain of the networks, reconsider Figure 3.14. However, in Section 4.8 we also do evaluate the performance of the combined networks. We computed the *mean absolute deviation* (MAD), the *mean squared error* (MSE) and the *peak signal to noise ratio* (PSNR) for the masked area (Figure 3.4n) of each scene. We grouped the results by the different evaluation-datasets and listed the mean errors in Tables 4.1(MAD), 4.2(MSE) and 4.3(PSNR). The tailing number behind the dataset names gives the count of scenes contained in each. In addition to that, we also plotted for some scenes the predictions below the ground truth values for all BSDF parameters in Figures 4.15 to 4.20. Considering the mean errors of all datasets, the results show, that the networks struggle especially with *metallic* and *roughness*. For *roughness* this might be the case, because this parameter is probably the toughest

dataset	#	basecolor	roughness	metallic	transmission	subsurface
ape 3mat	10	0.0575	0.0555	0.0707	0.0479	0.0271
ape multimat	15	0.0474	0.0515	0.0621	0.0423	0.0226
ape 2mat	3	0.0549	0.0680	0.0356	0.0631	0.0356
armadillo 2mat	5	0.0398	0.0724	0.0780	0.0378	0.0044
bunny multimap (no disp)	6	0.0478	0.0555	0.0608	0.0457	0.0197
bunny 3mat (no disp)	6	0.0522	0.0461	0.0747	0.0347	0.0113
bunny 2mat (no disp)	6	0.0532	0.0617	0.0597	0.0611	0.0205
dragon 2mat	5	0.0504	0.0734	0.0625	0.0448	0.0290
sphere 2mat	5	0.0534	0.0480	0.0773	0.0494	0.0205
ALL	61	0.0506	0.0570	0.0656	0.0461	0.0212

Table 4.1: Mean Absolute Deviation (MAD) of the evaluation dataset.

dataset	#	basecolor	roughness	metallic	transmission	subsurface
ape 3mat	10	0.0061	0.0150	0.0328	0.0131	0.0067
ape multimat	15	0.0045	0.0126	0.0249	0.0118	0.0051
ape 2mat	3	0.0057	0.0209	0.0090	0.0173	0.0083
armadillo 2mat	5	0.0030	0.0229	0.0448	0.0082	0.0004
bunny multimap (no disp)	6	0.0045	0.0142	0.0261	0.0132	0.0040
bunny 3mat (no disp)	6	0.0063	0.0105	0.0418	0.0080	0.0020
bunny 2mat (no disp)	6	0.0062	0.0169	0.0247	0.0254	0.0052
dragon 2mat	5	0.0053	0.0227	0.0220	0.0119	0.0082
sphere 2mat	5	0.0068	0.0113	0.0446	0.0148	0.0044
ALL	61	0.0053	0.0153	0.0302	0.0133	0.0049

Table 4.2: Mean Squared Error (MSE) of the evaluation dataset.

dataset	#	basecolor	roughness	metallic	transmission	subsurface
ape 3mat	10	22.64	19.32	16.02	20.70	24.68
ape multimat	15	23.60	19.24	16.23	19.47	23.16
ape 2mat	3	24,09	17.29	20.51	17.84	23.39
armadillo 2mat	5	25.82	17.14	16.92	22.41	35.88
bunny multimap (no disp)	6	23.57	18.62	16,07	18.95	24.13
bunny 3mat (no disp)	6	22.83	21,02	16,09	22.76	29.65
bunny 2mat (no disp)	6	23.38	19.23	18.21	19.20	28.93
dragon 2mat	5	23.14	17.35	17.18	20.39	25,04
sphere 2mat	5	23.80	19.98	15.66	21.38	29.38
ALL	61	23.53	19,01	16.66	20.31	26.43

Table 4.3: Peak Signal to Noise Ratio (PSNR) of the evaluation dataset.

(of the five in interest) to infer. That is, because unlike for the other parameters there is not a single feature, that directly indicates the *roughness* value. Remember, *basecolor* is approximately directly observable in a sufficiently well lit image. For *metallic* tinted specular highlights serve as features. For *transmission* the backlight eases inference and for subsurface we utilize the checkerpattern. Differences in light intensities under varying illumination could serve as features to estimate *roughness*. However, light intensity changes are not directly observable in the input images and moreover might be confused with variations in the *basecolor*. Besides the problem of *roughness*, probably being the toughest parameter to infer, *roughness* has another disadvantage in this comparison. The network that predicts *roughness*, also predicts the illumination. Moreover, the network was weakly split into two parts. One of these parts predominantly takes care of features, that are individual to one view. Previously we showed, that these features do contribute to the inference of *roughness*, however they are not completely dedicated to it. The net can be seen as only having half as many features available (plus the supporting illumination-features) for computing *roughness*, than the networks for the other BSDF parameters have for their respective ones. Increasing the *roughness*-features further, might make it more competitive to the other nets.

Still, the worst performance shows *metallic*. We suspect, that this is caused by the fact that tinted highlights are a good feature to detect *metallic*. The problem with this is, that on the one hand these highlight might not appear in high enough numbers. On the other hand, and even more important, these highlights are also spatially distant to another. The problem with this is, that our network has a global view on the input images, however it is build iteratively from neighboring areas (with increasing resolution). This means that the network might struggle more to fuse spatially distant features than for example the net of [15] with its fully connected layers. To overcome this problem, it might be helpful to extend the net for *metallic*. Originally we create outputs by traversing the input-pyramid downwards, towards the highest resolution. An additional upward-pass in the same manner might help to fuse distant information better. And a second downward-pass could bring the intermediate results back to the desired resolution again. However, this is purely speculation and so far we have not tried this approach. In theory the same problems could hinder the *subsurface*-network from good results. However, the edges of the grid pattern are spatially connected and occur more frequently than specular highlights (for *metallic*). This might help to explain the better performance of *subsurface* compared to *metallic*.

Next to the findings in different performances for the different BSDF parameters, the error tables give a further insight to the networks. The errors do not show notable differences between the different geometries, even though training only happened with the *Ape*. This means all networks generalize well with respect to different objects in terms of geometry. Moreover, the same is observable for the different material types. The networks were only trained with *multimat* and *2mat*, but the performance on *3mat* is as good and even better in some cases – considering the *Bunny*-dataset.

To make these findings more tangible, we plotted the results of some (as diverse as possible) evaluation scenes in the subsequent figures. In general they all show reasonable good predictions, with only few exceptions. In Figure 4.15 the left side of *roughness* and the predictions for *transmission* are not that good. Besides some fine details in *metallic* the other parameters are well predicted. In Figure 4.16 the contours of *metallic* are too blurry and *transmission* is too weak. In *subsurface* some very subtle false-positives occur and *basecolor* and *roughness* contain slight noise. Besides that, the predictions are fine again. The results in Figure 4.17 contain notable errors in *roughness* probably caused by the rather dark color. Moreover, there is a small area with wrong *subsurface* on the left. Besides little noise in the *basecolor* the remaining parameters seem fine. Figure 4.18 also has some slight noise in the *basecolor*. However, it mainly struggled with the matte turquois metallic area. Here *metallic* and *transmission* have false predictions. But all other parameters have good predictions again. Note that especially the very shiny area is recognized as such (with a low *roughness* value). In Figure 4.19 the only problem seems to be the estimation for the *metallic* value being to weak. And in Figure 4.20 especially *roughness* and *transmission* are deviating from the ground truth. The reason for this is probably the tough input, which only shows subtle differences in illumination. Further, the many small highlights (from low frequency displacement) make already the inputs very noisy.

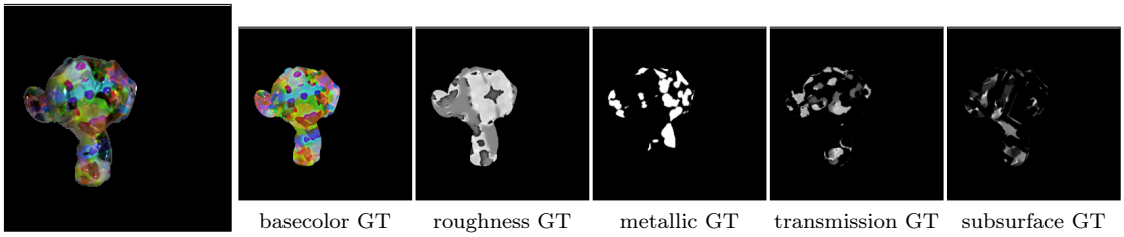


Figure 4.15: Ape multimat (Scene 4) Ground truth (GT) values and predictions (P) for the inputs on the left from our architecture.

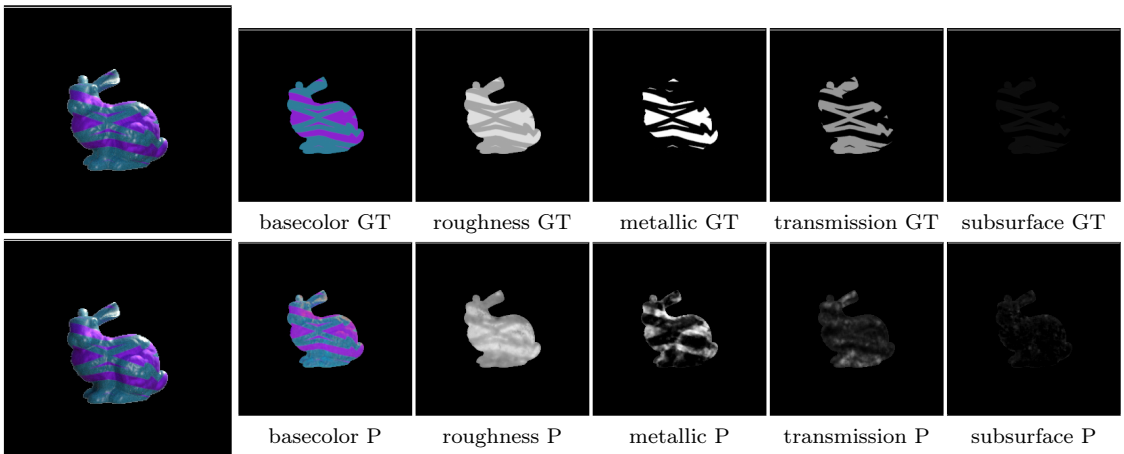


Figure 4.16: Bunny 2mat (Scene 101) Ground truth (GT) values and predictions (P) for the inputs on the left from our architecture.

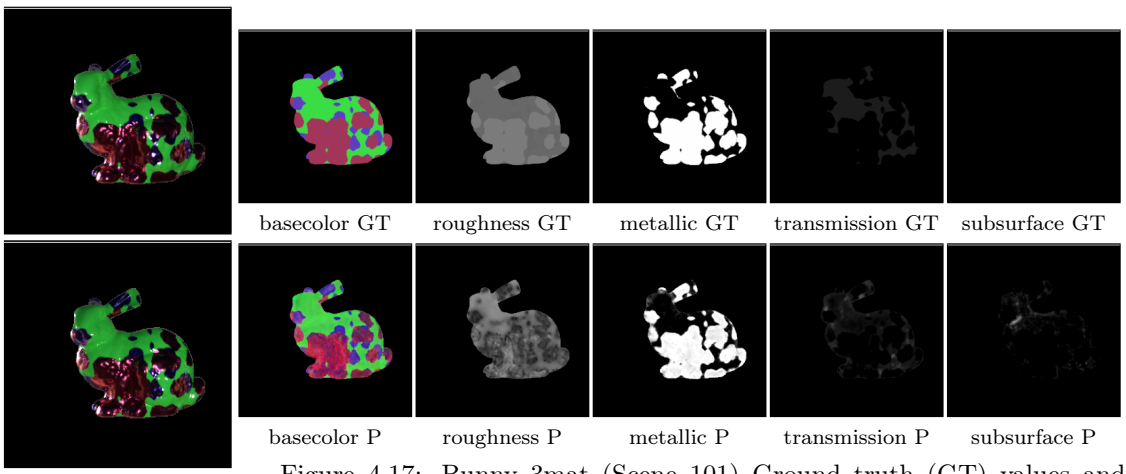


Figure 4.17: Bunny 3mat (Scene 101) Ground truth (GT) values and predictions (P) for the inputs on the left from our architecture.

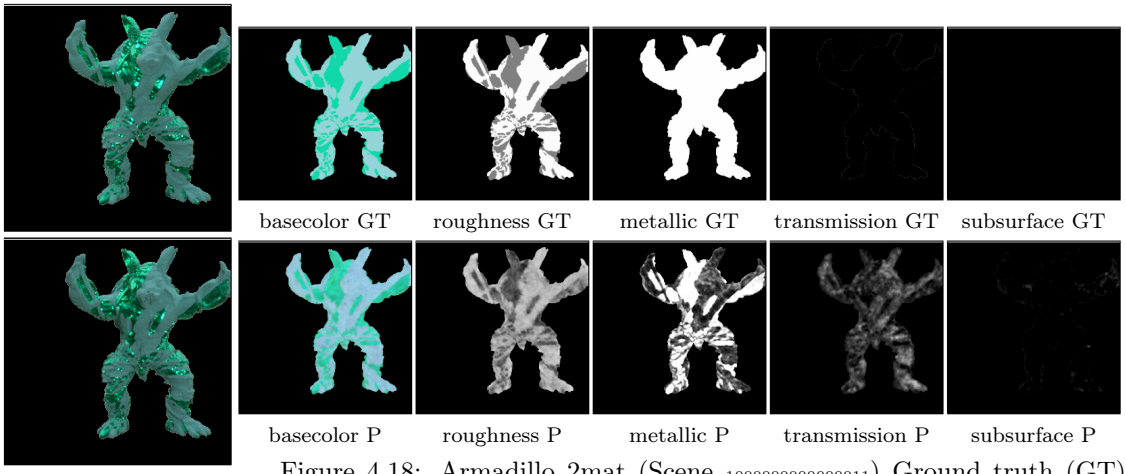


Figure 4.18: Armadillo 2mat (Scene 1000000000000011) Ground truth (GT) values and predictions (P) for the inputs on the left from our architecture.

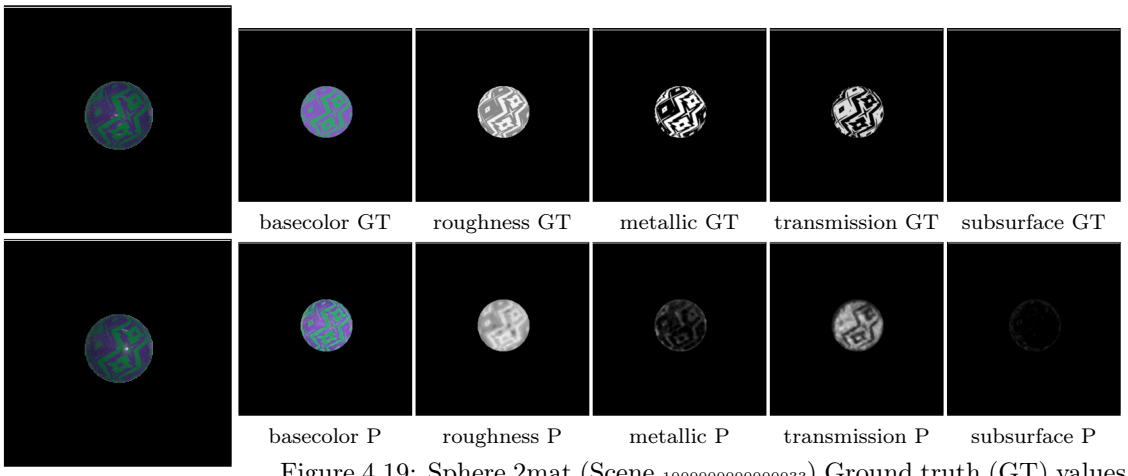


Figure 4.19: Sphere 2mat (Scene 1000000000000033) Ground truth (GT) values and predictions (P) for the inputs on the left from our architecture.

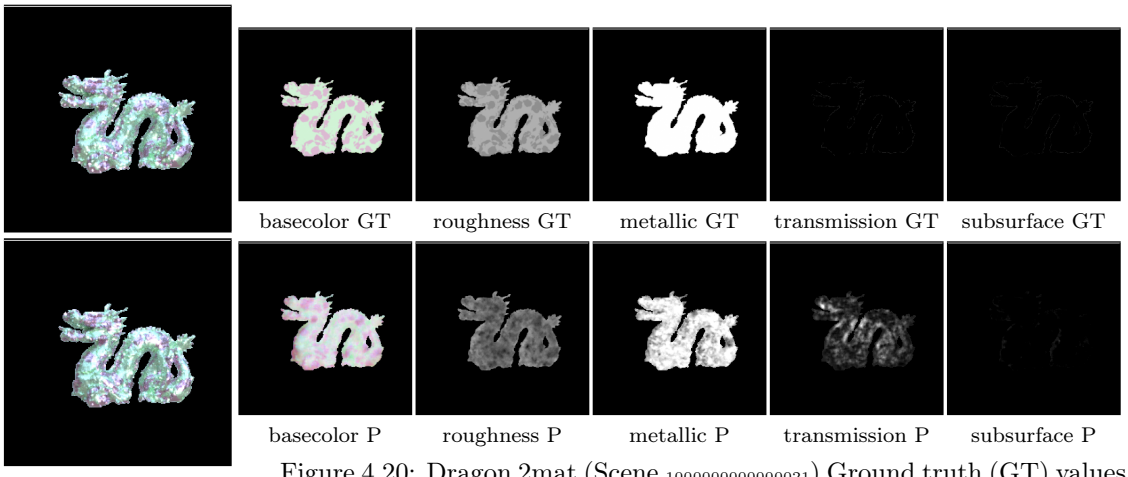


Figure 4.20: Dragon 2mat (Scene 1000000000000021) Ground truth (GT) values and predictions (P) for the inputs on the left from our architecture.

4.6 Improvements by Using Multiple Views

To confirm, that our architecture indeed benefits from using a higher number of views as inputs, we computed some error-metrics for a varying number of views. To repeat, a novel view is defined, as either a variation in lightvectors (i.e. the light source was moved) or a variation in viewvectors or both. The latter (but also the first) can be achieved by moving the camera or the scanned object. This means we have two ways to extend the inputs. In Figure 4.23 we computed the errors for the predicted *basecolor* and *roughness* of one scene (that was already shown before, in Figure 4.15). In detail we computed the *root mean squared error* (RMSE), the *peak signal to noise ratio* (PSNR) and the *structural similarity* (SSIM). We proceeded in the same way for both BRDF parameters. First we compute the predictions with a single view only. We used the first position, illuminated by the second flash-light. This caused the appearance of a shadow on the ear on the left side in the image. For both, *basecolor* and *roughness*, it is clearly visible that the prediction quality suffers in this area. Next, we used only the first position again. But this time with all available light sources – both the flashes and the projector. This caused a notable improvement in the prediction quality of both BRDF parameters. To make it clearer we have plotted the different views that were used as inputs over each of the predictions in Figure 4.15.

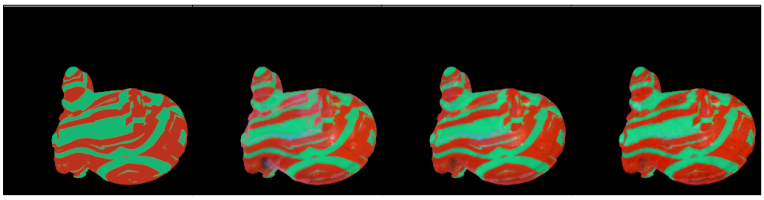
Subsequently, we also made predictions using all available views. These were the three light configurations observed from three different positions, with all pixels mapped to the corresponding pixels in the image of the first position. The prediction of *basecolor* improved again. This time, however, it did not improve as significantly as in the previous, when increasing from one to three views. For *roughness*, the prediction was slightly worse (in terms of RMSE and PSNR) than for the three views from only one position. We assume that this is the case because, on the one hand, the left side, which was already wrong before, is now slightly worse and, on the other hand, because the dark spot at the lower right (also wrong before) is now slightly worse as well. These problems are not that visible when just looking at the predictions. On the contrary, the prediction with 9 views even looks a bit better. For an example, have a look at the dark (low roughness) area around the apes mouth, at the lower middle region. This finding is also confirmed with the SSIM[1] metric, which is more in line with human perception than the other two. Nonetheless, there was a decrease in RMSE and PSNR. We think that this is caused by, how the inputs are presented to the net. Remember, we compute a mapping of all inputs into the image of the first position, to have measurements of one object-point in the same input locations. However, for positions other than the first one, these mappings usually have missing parts. Consider the second and third rows (that show the respective positions) of the top right image-grid in Figure 4.15. These holes are part of the input and our network has to learn to compute and apply an *image inpainting* (which is a whole field of research by its own) in addition to its original task of predicting a BSDF parameter. We assume, that it would be beneficial for the BSDF predictions, to ease this task for the architecture. Possible options would be to fill these holes beforehand with fake samples, either by interpolation or fancier *inpainting* techniques. An alternative solution could be, to apply a kind of normalization to the first hidden layer of the network. To be more precise, the kernel sizes for the first layer filters (and of course all others) in the network are known by definition. Moreover, the mask, that was computed with the mapping (and is used as input to the net already) could be used in combination with the kernel size to determine the number of valid input-pixels used for every output-pixel. The output-pixel can then be normalized with this number. Another approach that could help to further improve predictions for multiple positions would be to improve feature fusion (between the different positions). We simply used max-pooling for this task, but a learned fusion would also be conceivable. However, all three strategies are out of scope for this work and are only ideas for possible improvement.

Due to the negative sentiment of the previous paragraph about the use of multiple positions, we think it is necessary to emphasize usefulness a little more. For this, take a look at Figure 4.21. The *basecolor*-predictions from one position (first row, the second left and second right images),

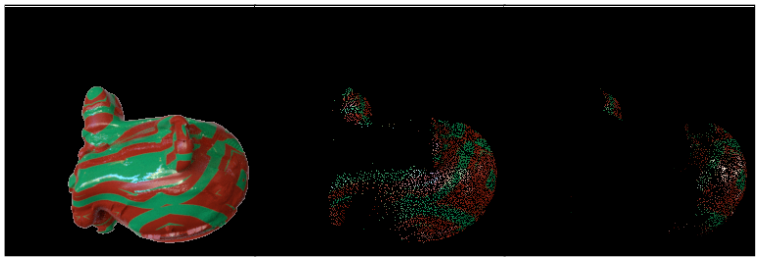
[1] Note that we computed SSIM on the whole image (including) the background. It would not make sense to compare different scenes in that manner, but it is reasonable for identical ones.

regardless of the number of different lights used, contain severe errors. These errors are caused by a bright light in the environment. The reflection of this light is clearly visible in the input ambient-image of position zero, at lower left in the figure. The network fails to detect it as such and consequently does not remove it. Only the inputs of further positions, that do not show this highlight, lead to a successful prediction (top right image).

Notice that in the previous two paragraphs we covered two slightly different reasons for the improvement from the usage of additional positions. In the first, a region without any proper samples (the shadowed ear of the ape) was improved by novel positions, that contained valuable information for this area. In the second a region that is well lit by the artificial light was wrongly estimated, because of the disturbing influence of the scene surroundings. Still, the additional positions helped here as well to improve the results. For completeness, we computed the errors of Figure 4.15 again. But this time only in areas, that are properly lit from most positions. For this, we manually draw a mask. It is shown as overlay on the scene in the top left image. This mask approximately conforms to the bright areas in Figure 4.22. There we counted the number of positions that are observable after mapping all to the first position. Note that an observable pixel does not imply a well lit one. We put the found errors in brackets next to the previously computed ones. As before they show an increased prediction-quality with the usage of more positions. However, the improvements in predictions here are less significant than before. As expected, benefits from novel samples have more impact, than multiple samples of the same position. Still, we showed (by the error values and with Figure 4.22), that also the latter improvements exist.



GT | Prediction from 1 view | Prediction from 3 views (same Position) | Prediction from 9 views (3 Positions)



Position 0: ambient | Position 1: ambient | Position 2: ambient

Figure 4.21: Predictions for the *basecolor* for Ape 2mat (Scene 1000015) of the testing-set and the ambient images of all used positions mapped to the first one. Using more than one position helps to remove the error, that was caused by a strong environment light. Position 2 is probably not as useful as Position 1.

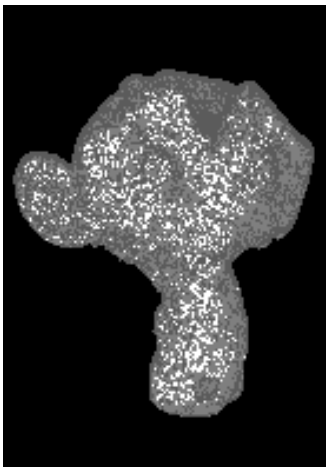


Figure 4.22: Ape multimat (Scene 4): Number of available positions. 3 positions (white), 2 positions (gray), 1 position (dark gray). If a pixel is available for a position, this does not necessarily mean, that it contains valuable information.

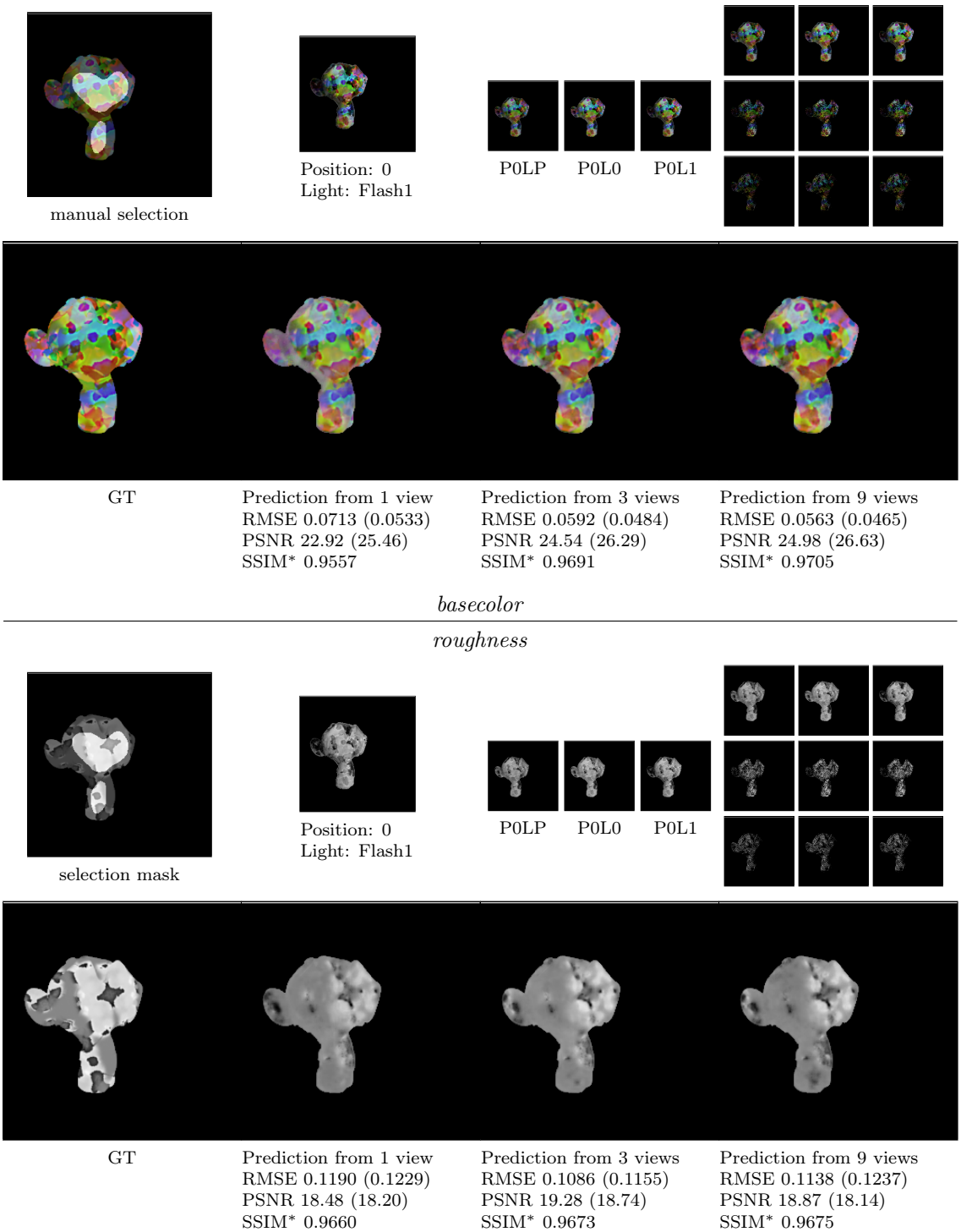


Figure 4.23: Predictions for the *basecolor*(top) and *roughness*(bottom): Ape multimat (Scene 4) of the evaluation-set. The main row shows the ground truth value (GT) and next to it, from left to right, predictions made with and increasing number of views as inputs. The prediction quality rises slightly with the number of used views. Error measures for the whole object (without background) are given under the respective predictions. The value in brackets shows the same measure, but only computed for a manually selected area. This area is highlighted in the top left image. On the right to this, the different views that were used as inputs are depicted (mapped to the first position). The mark * means: computed on the whole image, including the background.

4.7 Evaluation High Resolution

To show, that our network is able to handle arbitrary image resolutions, as intended, we rendered some scenes from the evaluation dataset again. This time with an increased resolution. We used 512×512 and 768×768 instead of the training-set resolution of 256×256. Further, we passed these renderings through our network. The results for the highest resolution for *roughness* and *basecolor* are shown in Figures 4.24 and 4.25. Inference did not take notably longer than with the original resolution and finished almost instantly again. To create these high-resolution outputs we could only use one view. Due to limitations in the VRam size of our device and the rather wasteful handling of it by our current implementation. This problem would be easy, but laborous to solve, by scheduling data transactions to the GPU more carefully. Due to time constraints we have not dealt with this yet. However, to come back to the results of the high-resolution dataset: Remember, that a view was defined as one distinct camera, light configuration. To visualize the view we also plotted the ambient lit image and the one with the additional artificial light source in the respective Figures in the bottom left corner. (For Figure 4.24 we used flash0 and for 4.25 the projector.) For the current evaluation, the ambient and the additionally lit render, are the only two "photographs", that are provided to the net. The other inputs, such as surface-normals and lightvectors, are not shown in these figures. What is shown in addition to the high-resolution predictions (at the top), are on the one hand, the ground truth values (in the bottom right) and on the other hand, also the predictions for inputs of size 256×256 (bottom right corner within the high-res predictions). For the *basecolor* in Figure 4.24, the predictions are good. Only the shiny golden areas caused some trouble to the net and lead to a slightly false color with some noise. More important is, that it is clearly visible, that the network successfully managed to upscale the predictions, without notably sacrificing quality of the predictions. It did not introduce blurred areas, as a naive method like *bi-cubic/-linear upscaling* would do. All edges, that are found in the low resolution predictions, maintain sharp in the big prediction. To back these visual findings with some data, we also computed the errors for this specific scene. They are as listed below in Table 4.4 for the different resolutions (* was computed on the whole image, with background. 512×512 is not plotted anywhere). The small predictions perform slightly better, but not significantly.

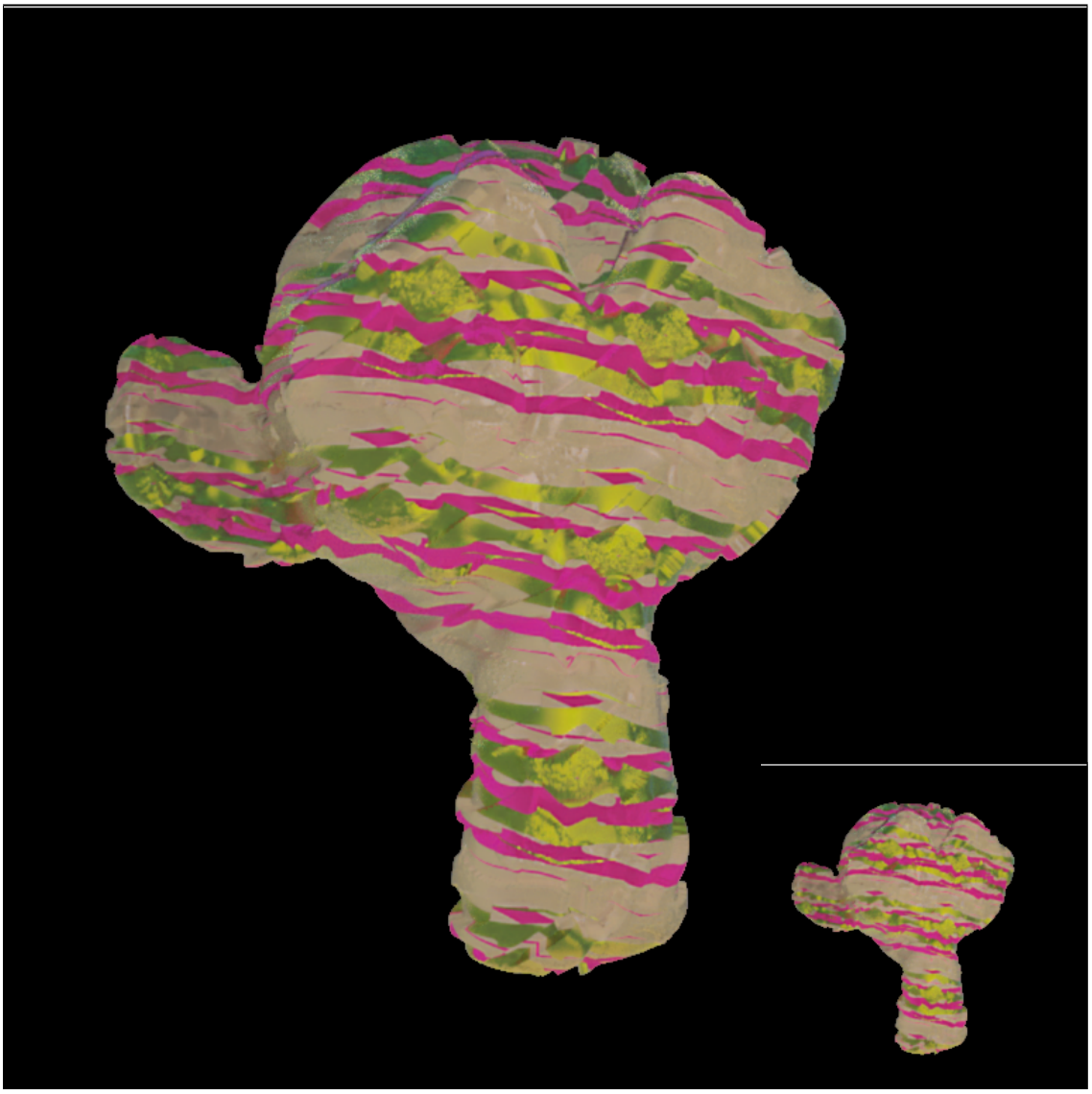
resolution	RMSE	PSNR	SSIM*
768×768	0.0720	22.85	0.9378
512×512	0.0698	23.11	0.9349
256×256	0.0632	23.98	0.9399

Table 4.4: Ape 3mat (Scene 5) *basecolor*: errors for different in- and output-sizes.

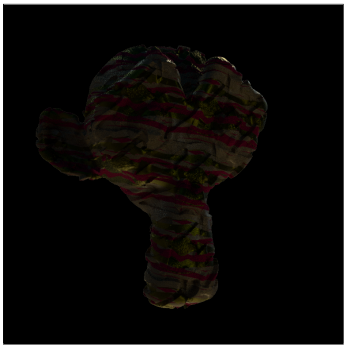
resolution	RMSE	PSNR	SSIM*
768×768	0.1183	18.53	0.9459
512×512	0.1164	18.67	0.9396
256×256	0.1086	19.28	0.9332

Table 4.5: Ape 3mat (Scene 5) *roughness*: errors for different in- and output-sizes.

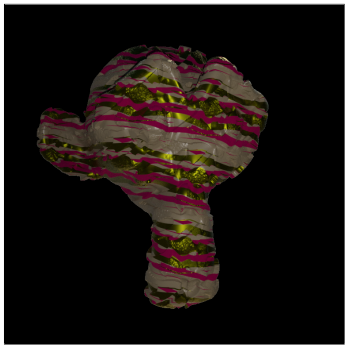
Similar findings show for *roughness* in Figure 4.24 and Table 4.5. The errors do not change that much and the predictions conform to the ground truth (of the respective resolution). Regardless of the slightly worse performance compared to *basecolor*. The network predicted *roughness* well in most areas. The pink area was correctly detected as the roughest. Likewise the beige area got properly recognized as the shiniest. Only the golden area caused some notable problems. Here the predicted *roughness*-value is slightly too high. The shadowed area at the ear also caused some difficulties for the network and contributed to the errors. When more views with sufficient illumination are used, the problem disappears.



Predicted 768 × 768: *basecolor* Predicted 256 × 256: *basecolor*



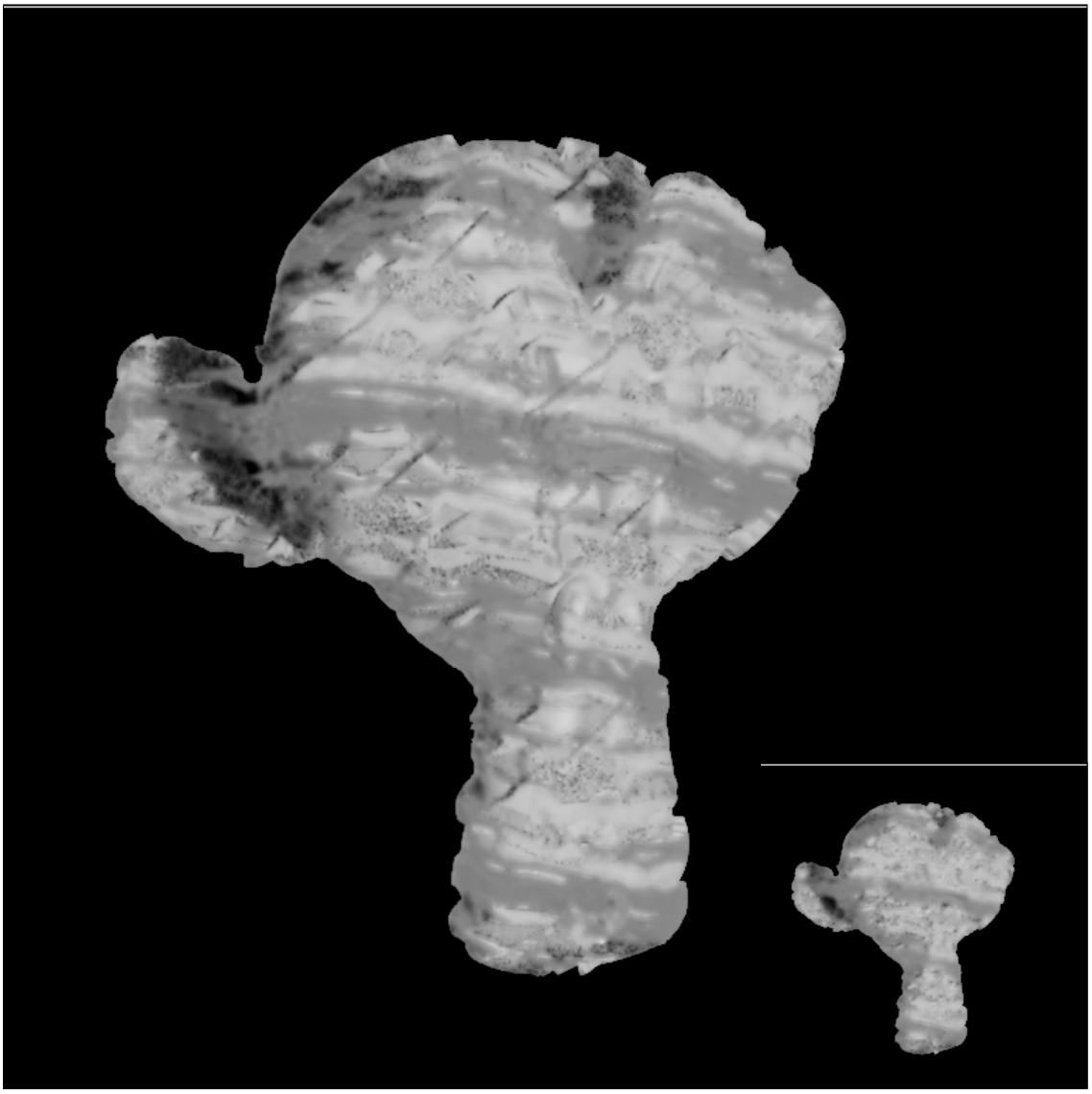
Input: ambient



Input: projector

GT: *basecolor*

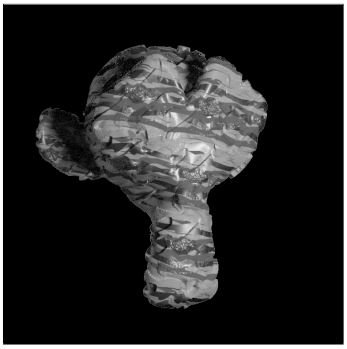
Figure 4.24: Ape 3mat (Scene 5) *basecolor*: Predictions (top) from different resolutions, both from one view. Inputs (bottom left) and ground truth values (bottom right).



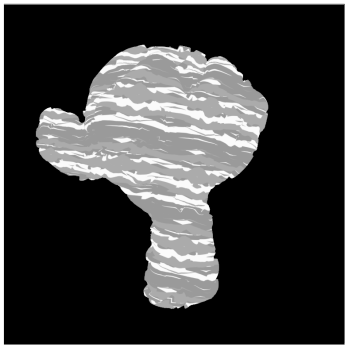
Predicted 768 × 768: *roughness*

Predicted 256 × 256: *roughness*

Input: ambient (luminance)



Input: flash1 (luminance sqrt diff)



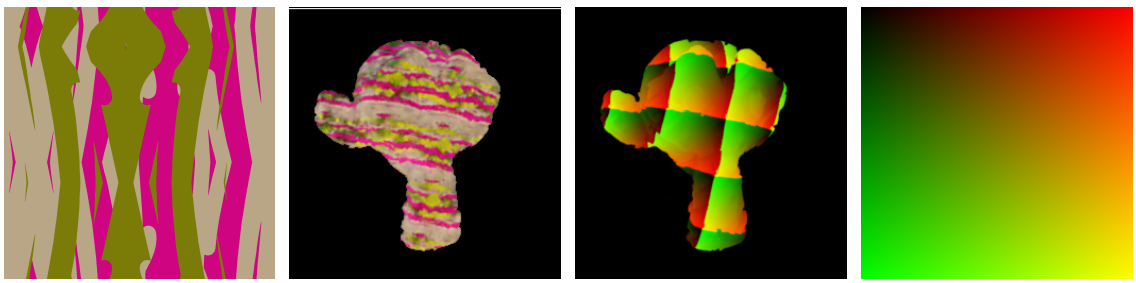
GT: *roughness*

Figure 4.25: Ape 3mat (Scene 5) *roughness*: Predictions (top) from different resolutions, both from one view. Inputs (bottom left) and ground truth values (bottom right).

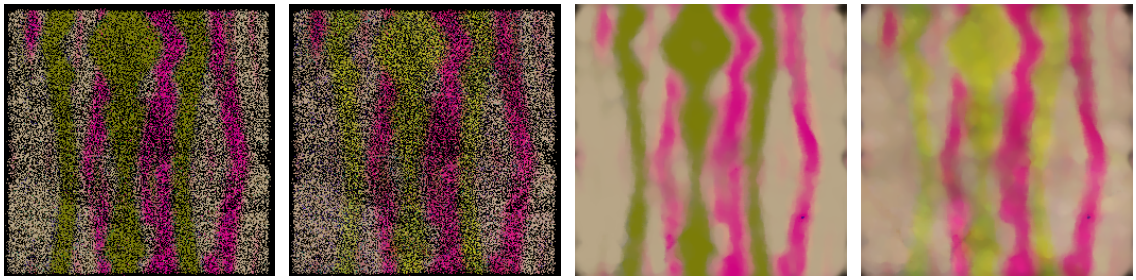
4.8 Re-render

The original goal, and the reason, why we determined BSDF parameters in the first place, was to re-render a scanned scene under new views. Finally, to make this possible, it is necessary to map our predicted parameters back onto the object. Or more practically, back to the original texture, which is then mapped with a uv-map onto the object by the rendering engine itself. Note that with respect to a real scan (as we use it for inputs to our network), there is no such thing as an original texture and an original uv-map. For scans, both are usually generated after the computation of the geometry and texture (or better the BRDF/BSDF parameters instead). For this reason, we explicitly do not want to call this mapping-process part of our contribution.

However, for the creation of the datasets, it was necessary to define uv-maps (which we did once for each geometry with Blender). Further we can render the object with its texture coordinates, like we did for all other ground truth parameters. This provides us with a way to map our predictions back to the object. In short the idea is as follows: For each pixel of the predicted BSDF parameters, look up its position in the original texture map, by consulting the render of the texture coordinates. Store the pixel's value in a new image at the looked up position. This new image is the predicted texture, that can be used to create re-renderings. Figure 4.26 illustrates the idea in more detail. For rendering a scene, a randomly generated texture (Figure 4.26a) is used, which in this case defines the base color. From the renderings and the artificial scanner-data predictions of the *basecolor* are made (Figure 4.26b). Moreover, we also rendered the scene (Figure 4.26c), with the texture coordinates (Figure 4.26d) instead of its material. Depending on the (random) scale of the uv-map it can happen, that some positions on the object's surface use the same texture-values. In the provided example this happens multiple times[2]. This repetition can be seen well in Figure 4.26c. Anyways, these rendered texture-coordinates can now be used to map the predicted *basecolor* back in to the space of the original texture. If multiple samples were mapped to the same position, we used their mean instead. The result of this mapping is shown in Figure 4.26f. For comparison we also mapped the render of the ground truth *basecolor*



(a) basecolor texture (for rendering)
(b) basecolor (prediction from view)
(c) texture coordinates from view
(d) texture coordinates



(e) GT mapped back
(f) P mapped back
(g) GT mapped, postproc.
(h) P mapped, postproc.

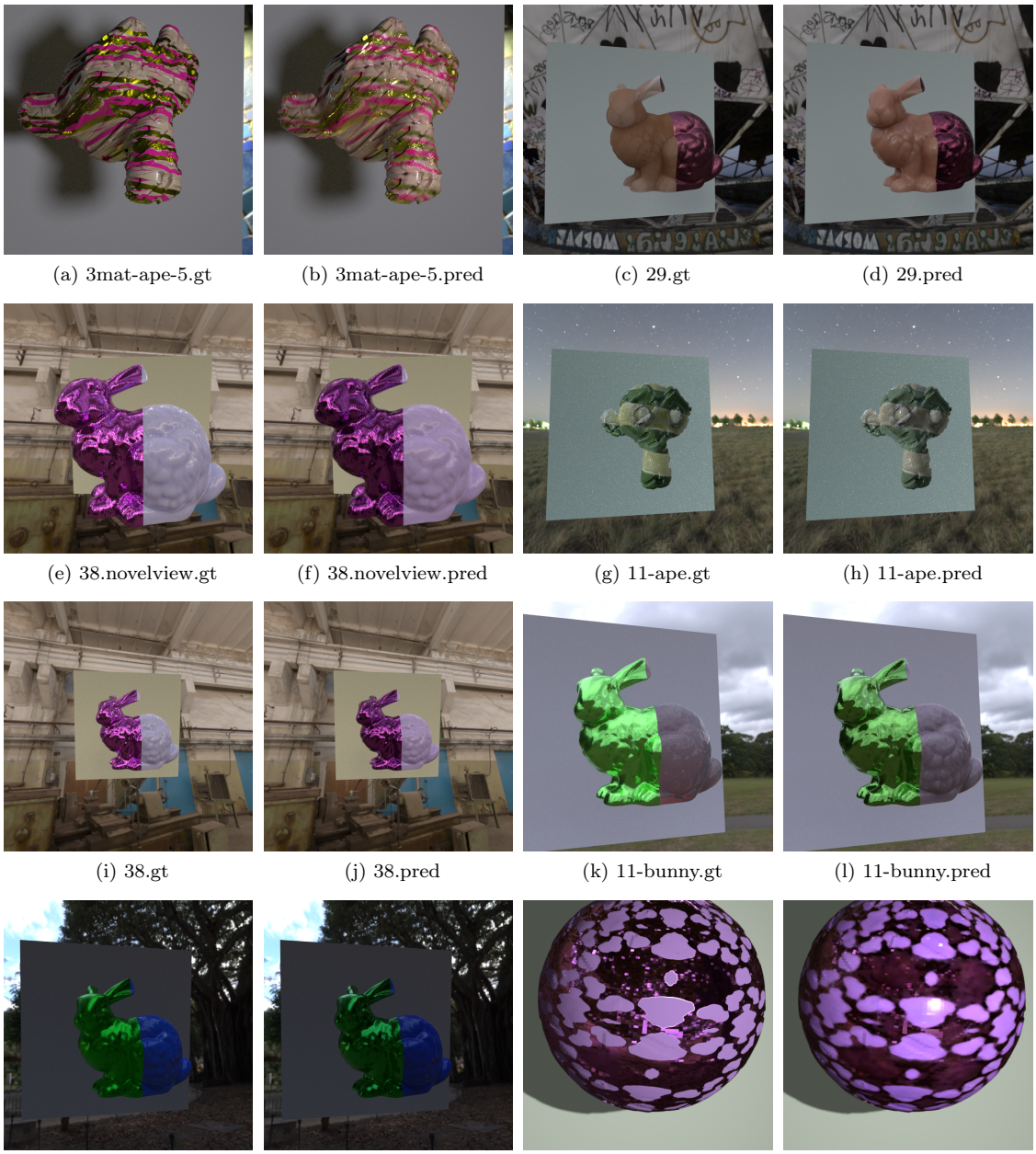
Figure 4.26: A depiction of the different steps in mapping our predictions (P) or ground truth values (GT) back on the object. Visualized with the *basecolor* prediction.

[2]This is an extreme example. The majority of the scenes in the datasets use a less repetitive mapping.

(which is not depicted, but would look like Figure 4.26b, just with the GT values instead) in the same way. This returns Figure 4.26e. Ideally, it should look like Figure 4.26a again. However, it is clear, that this is not the case. The texture is only sparsely sampled. This has two reasons. The first one is not that problematic in the depicted scene, but exists for most other scenes. It is the fact, that the camera can not observe all used texture-pixels from one position. We can overcome this partially, by using all available positions, to recreate the texture. This is in fact, what we did for Figure 4.26f and 4.26e. For values, that were mapped to the same texture-locations, we instead used the mean again. In most cases, this method helps to obtain a more complete texture. But due to the way mapping works[3], the texture is still not complete in the general case. If these textures would be used for a re-rendering it would look inherently wrong. To improve it, the holes need to be filled. A very simple method to do this quickly is, to repeatedly apply dilation and blurring (to avoid hard edges from dilation). Moreover, overwriting the modified image with the valid pixels again, helps to maintain these values. The so modified result is shown in Figure 4.26h. It resembles the ground truth *basecolor* texture, but still has notable deviations from it. This will show in the rendering later. To visualize, that this is not a problem of our predictions, but rather the mapping procedure, we mapped and post-processed the ground truth parameters in the same manner (Figure 4.26g). This is supposed to be the exact same image as 4.26a, but clearly is not. Moreover, we only have 8 bits available to encode the texture coordinates in the rendered images. This demands textures of no more than 256×256 pixels resolution. This means the created textures are a lot blurrier than the original one. This adds to the previous problem. The textures created in the explained manner, suffice to roughly visualize the predictions on the 3D objects. But we feel, that this very limited mapping does not do justice to our predictions. A more sophisticated mapping would be necessary for good re-renderings. However, this is out of scope for this work. For the sake of making proper mappings (and thereby re-renderings) possible, we created yet another dataset. Here we used two random materials, with the definition, that each of them takes the horizontal half of the texture map. This allows us to compute mean values of the left and right side after mapping (as explained before) to define the materials. Strictly, this is no longer a prediction of spatially varying materials. Rather, this can be seen as predicting spatially constant clusters of material, which makes the overall task easier. However, we used this strategy in addition to have the possibility to perform re-renderings without mappings that degrade the quality of our predictions. In Figures 4.27, 4.28 and 4.29, we show some re-renderings of the predicted materials using half of the texture for each of the two materials. In the same figures, we additionally show the re-renderings of some truly spatially varying materials where the mapping worked reasonably well.

Figure 4.27 shows some scenes rendered with environments and camera configurations that were included in the prediction inputs. Despite the problems, mentioned for the spatially varying parameters (Figure 4.27b and 4.27p), all re-renderings of predicted BSDF parameters look very similar to the renderings with the ground truth parameters. Only small errors are visible. For example, besides a well predicted *subsurface* value in Figure 4.27d, the re-render does not look perfect, due to a wrong *roughness*. Even better are the predictions in Figure 4.27f. Differences are barely visible, even under a novel view (Figure 4.27f). The remaining scenes all worked quite well and likewise contain only small visible errors. We will discuss them with the following figures, for which we use different environments in addition. The sequential inference of the BSDF parameters in these scenes took about 30 seconds for each (on a NVIDIA GTX 1080ti, with 9 views per scene). If necessary, this could be speed up further by parallelization, in compliance with the dependence graph (Figure 3.14). Inference of a single BSDF parameter, based on multiple views (like in Section 4.5), takes about $0.5 - 4$ seconds.

[3]The pixels in the camera usually never refer to an exact pixel in the texture. Rather, they show an interpolated value. Now, if the values from the camera pixels are mapped back into the texture, they will usually never map to an exact pixel. Nearby texture pixels must be calculated from these mapped values. The strategy described before works indirectly like *nearest neighbor*. Therefore, some pixels of the newly created texture remain undefined.



(a) 3mat-ape-5.gt (b) 3mat-ape-5.pred (c) 29.gt (d) 29.pred

(e) 38.novelview.gt (f) 38.novelview.pred (g) 11-ape.gt (h) 11-ape.pred

(i) 38.gt (j) 38.pred (k) 11-bunny.gt (l) 11-bunny.pred

(m) 10.gt (n) 10.pred (o) sphere-30.gt (p) sphere-30.pred

Figure 4.27: Re-renders in their original environments from a view that was provided for inference (except 4.27e and 4.27f). (gt) shows renderings with the ground truth BSDF parameters and (pred) shows renders with the predicted parameters.

To show that our predictions generalize well, we also do some re-renderings with environments that differ from those in the input. For this purpose, a completely uniform environment is usually well suited, since it avoids influences from the surroundings that might not be assignable for human perception (for example color variations, caused by different light sources). To render the scenes in the following, we mostly used an entirely black environment. In Figure 4.28 we rendered some scenes under novel lights. The first one shows good results for both materials, besides slight aberration of the *basecolor*. The second scene also has a slightly too dark *basecolor* for the left side. On the right side the *subsurface* strenght was predicted very well, but *roughness* is a bit to low. This is why it has some reflection highlights and does not look quite as soft as the original material. The last scene shows a dark purple metal on the right. Besides it is not really visible in the dark environment, it got predicted well. The left side is made of a lighter blue material. Transmission was detected by our net, but assumed slightly to weak. Therefore, especially the pictures 50 and 57 (where the light is behind the object) are too dark. For the same reason in picture 78 the right side of the bunny's head is notably shadowed in the prediction, whereas in the original the light passes nearly effortless through it.

Like in the previously explained Figure, we created some further re-renderings in Figure 4.29. However, here we fixated the light and only moved the camera. For the first two scenes we used a completely dark environment again. For the third scene, however, we used an environment map again, to better visualize the highly reflective material. Note that this is a different environment-map, than the one used for capturing the data and predicting the BSDF parameters. The first scene is the one with the previously shown spatially varying material. The yellowish tint of the golden part is slightly too strong. But the main problem is, as discussed before, the mapping of the predictions back on the object. The second row shows a re-render of a material like the one used for the bunnies before. Two materials each spatially constant and defined as one half of the original textures. The render only looks more complicated than a simple halved texture, because of the uv-mapping (that is different to the bunny[4]). Nevertheless, this is only deceptive. The predictions for this scene are good. Especially the *subsurface* of the light green part was recognized. Only the saturation of the *basecolor* is slightly too low. The last scene shows a sphere with a spatially varying material again. The original scene contains a very rough light purple and a very shiny dark purple material. Both got recognized as such by our network, but the absolute values are not perfect. Especially the rough light purple material is a bit too shiny, which leads to highlights, that are not observable in the original scene. The absolute *roughness* value for the other material is better. There the *basecolor* is not predicted so well. It has a clearly wrong hue and also shows strong spatial variations (probably caused by reflections contained in the input images). Due to the high gloss in this area, the problems with the base color are hardly noticeable.

[4]For the bunny we just frontal-projected the texture on the object.

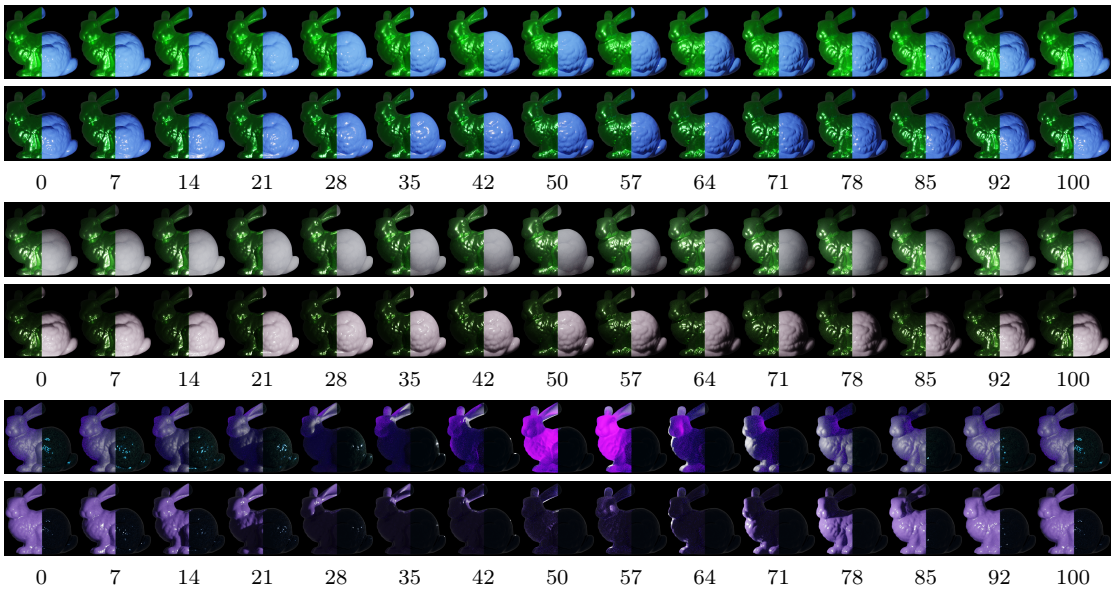


Figure 4.28: Renderings with a novel light configuration. The environment is completely black and only a single light, that circles around the object is used. For the upper two scenes the light starts at the left side and circles counterclockwise around the view-axis. In the scene in the third row the light cycles around the normal of the surface, on which the bunny is sitting on. It starts on the left side again. Then it moves between the object and the camera to the right side. Finally it travels behind the object, leftwards to its starting position. The upper rows (for each scene) show the renderings with ground truth BSDF parameters, the lower rows the renderings with the predicted parameters. The numbers under each scene indicate the progress of the light movement.

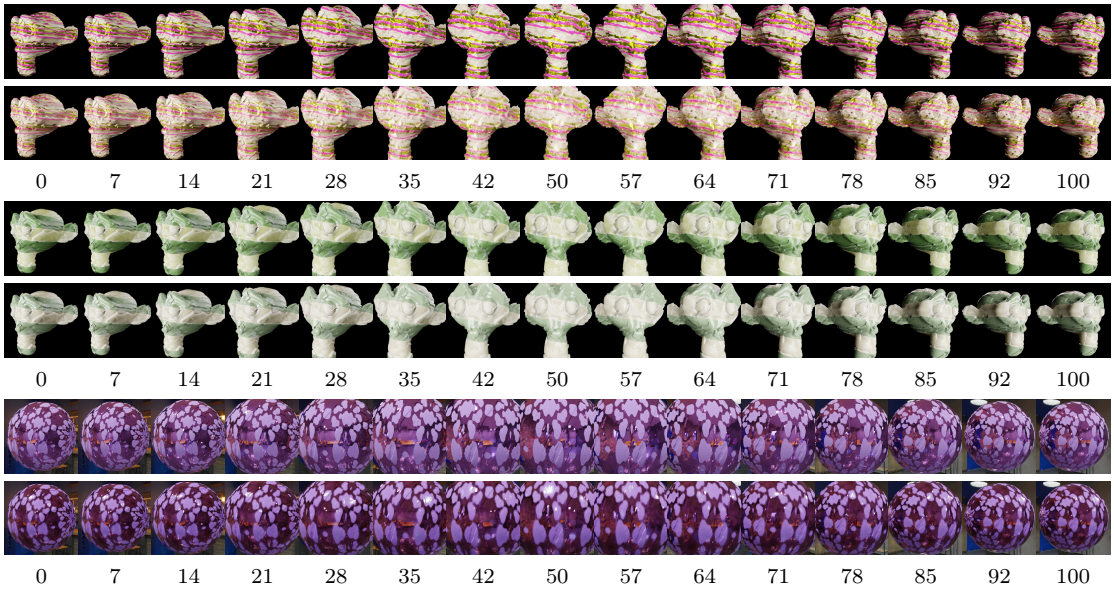


Figure 4.29: Renderings with a novel views. The lights are static. For the first two scenes two point-lights are used, one in the back and another in the front. For the last scene a novel environment map is used for rendering. In all three scenes the camera moves the same way: In a straight line in front of the object from right to left, tracking it. The upper rows (for each scene) show the renderings with ground truth BSDF parameters, the lower rows the renderings with the predicted parameters. The numbers indicate the progress of the camera movement.

Chapter 5

Conclusions

With this work we gave a brief introduction to *bidirectional reflectance distribution functions* (BRDFs) and its superset the *bidirectional scattering distribution functions* (BSDFs). We put a special focus on the Disney*/Principled BSDF*. BSDFs in general are used to model visual material properties of objects in computer graphics. The *Disney BSDF* is of interest to our work, because it is a state of the art material descriptor, that allows modeling an especially wide variety of different light transport properties. For the field of 3D scanning, the automatic inference of BRDF/BSDF hyper-parameters is of great interest. That is, because for a scene a BSDF with properly chosen parmeters, allows the realistic re-rendering of the scanned geometry. This is especially the case for spatially varying parameters. Besides an introduction to BSDFs, we also gave a short overview of state of the art methods, to infer these parameters. We pointed out, that the various approaches differ in some aspects. This is, because each of them is a trade-off in terms of prediction quality, supported material complexity and runtime, tailored to a specific use case. The main contribution of this work is the proposal of a novel deep learning based architecture to infer a selected set of the *Disney BSDF* parameters. For this, we described our own use case and related it to the approaches, that we described before. We use 3D scanning data and some special images, captured with a *structured light* or *stereo system* for inference. Or method is flexible in the sense, that parameters that are not of interest/are known beforehand could be left. This saves computation time and eases acquisition for some parameters. For capturing, we need some supporting tools in addition to the scanner. These are an uncalibrated diffuse white material exemplar and an uncalibrated backlight (which qualifies for the former as well) for *transmission*. Moreover, we need calibrated light sources and an (un-)calibrated projector for *subsurface*, the latter can be used again for the former. In terms of runtime our method is slightly slower than most neural net based approaches, and it also takes longer to obtain the input data. Mainly, because it utilizes more information as input. Still, it is notably faster than high precision approaches like *gonioreflectometers* or [21]. We do not support *anisotropic* materials (which some newer approaches do), nonetheless our architecture is able to describe a notably wide range of materials. We showed that especially the inference of *subsurface* and *transmission* is possible as well with the described approach. None of the other methods from the overview did that. Besides that, we described a very flexible method. Once trained, it allows arbitrary and varying image sizes and produces spatially varying predictions of BSDF parameters for actual three dimensional objects. All three trades are not that common in existing methods – especially not their combination and especially not in the fast neural network approaches. The outputs are (to our perception) the visually most important *Disney BSDF* parameters: *roughness*, *metallic*, *transmission*, *subsurface* and *mixed subsurface basecolor* (that can be used for both *subsurfacecolor* and *basecolor* in re-renderings). The architecture also dynamically processes a variable number of given views as input and benefits from additional ones. The prediction in the network works as an iterative refinement of the initial predictions – going down the pyramid of inputs (from the smallest to the largest resolution). As a result, our network can be defined with a small set of reusable weights. Moreover, for the same reason, the prediction of a single pixel has the perceptual

field containing the entire input, despite variable input sizes. Our overall architecture is defined by five such nets (with some dependencies) – one for each BSDF parameter. The second major part of our contribution is the creation of a novel dataset that can be used to train the described architecture (and others). This dataset contains many different materials. Some of them with large spatially constant regions with highly correlated BSDF parameters, as often observed on the surfaces of real objects. Others with rather random materials and uncorrelated parameters, to provide many different samples for training. Moreover, we ensured an exhaustive coverage of the parameter space with some special sets of manually predefined values. We also used these types of datasets, to emphasize the expose of some networks to specific material types. In the last part, we showed that our architecture is capable of learning what is desired from our dataset. Moreover, we evaluated performance using common error metrics. Further, we demonstrated visually, that while our predictions are not perfect for all scenes, they are generally good and can reliably describe the material properties of the scanned objects in novel views.

Hereby I declare that I have self-dependently composed the Master Thesis at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

Kaiserslautern, January 2021

Lauritz Feick

Bibliography

[1] Autodesk: List of ior values. https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/3DSMax/files/GUID-CCD9B76C-9AC6-46E6-8B9C-E367CFC0FDAF-htm.html. Online; accessed 6-January-2021. 8

[2] Blender. https://www.blender.org/. Online; accessed 1-December-2020. 16, 17

[3] Blender code: Principled bsdf. https://github.com/blender/blender/blob/master/intern/cycles/kernel/shaders/node_principled_bsdf.osl. Online; accessed 1-December-2020. 13, 42

[4] Blender documentation: Principled bsdf. https://docs.blender.org/manual/en/latest/render/shader_nodes/shader. Online; accessed 6-January-2021. 5, 6, 7

[5] Bsdf composition. https://upload.wikimedia.org/wikipedia/en/d/d8/BSDF05_800.png. Online; accessed 1-December-2020. 3

[6] Miika Aittala, Timo Aila, and Jaakko Lehtinen. Reflectance modeling by neural texture synthesis. *ACM Transactions on Graphics (TOG)*, 35(4):65, 2016. 9, 10, 15

[7] Michael Ashikmin, Simon Premože, and Peter Shirley. A microfacet-based brdf generator. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 65–74, 2000. 10

[8] FO Bartell, EL Dereniak, and WL Wolfe. The theory and measurement of bidirectional reflectance distribution function (brdf) and bidirectional transmittance distribution function (btdf). In *Radiation scattering in optical systems*, volume 257, pages 154–160. International Society for Optics and Photonics, 1981. 4

[9] James F Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, 1977. 5

[10] Brent Burley. Extending the disney brdf to a bsdf with integrated subsurface scattering. *Physically Based Shading in Theory and Practice'SIGGRAPH Course*, 2015. 5, 8

[11] Brent Burley and Walt Disney Animation Studios. Physically-based shading at disney. In *ACM SIGGRAPH*, volume 2012, pages 1–7. vol. 2012, 2012. 4, 5

[12] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017. 37

[13] Robert L Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG)*, 1(1):7–24, 1982. 5

[14] Valentin Deschaintre, Miika Aittala, Fredo Durand, George Drettakis, and Adrien Bousseau. Single-image svbrdf capture with a rendering-aware deep network. *ACM Transactions on Graphics (TOG)*, 37(4):128, 2018. 9, 10, 11, 15, 34, 41

[15] Valentin Deschaintre, Miika Aittala, Frédo Durand, George Drettakis, and Adrien Bousseau. Flexible svbrdf capture with a multi-image deep network. In *Computer Graphics Forum*, volume 38, pages 1–13. Wiley Online Library, 2019. 9, 10, 11, 41, 54

[16] Bernardt Duvenhage, Kadi Bouatouch, and Derrick G Kourie. Numerical verification of bidirectional reflectance distribution functions for physical plausibility. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, pages 200–208, 2013. 4

[17] C. Garth. Lecture notes: Computer graphics, 2017. 4

[18] Andreas Mischok Greg Zaal, Sergej Majboroda. hdrihaven.com. `https://hdrihaven.com/hdris`. Online; accessed 22-December-2020. 18

[19] Duck Bong Kim, Kang Su Park, Kang Yeon Kim, Myoung Kook Seo, and Kwan-Heng Lee. High-dynamic-range camera-based bidirectional reflectance distribution function measurement system for isotropic materials. *Optical Engineering*, 48(9):093601, 2009. 3

[20] Stanford University Computer Graphics Laboratory. The stanford 3d scanning repository. `graphics.stanford.edu/pub/3Dscanrep`. Online; accessed 28-December-2020. 21

[21] Sebastian Merzbach, Max Hermann, Martin Rump, and Reinhard Klein. Learned fitting of spatially varying brdfs. In *Computer Graphics Forum*, volume 38, pages 193–205. Wiley Online Library, 2019. 9, 12, 14, 34, 69

[22] Rosana Montes Soldado, Carlos Ureña Almagro, et al. An overview of brdf models. 2012. 4, 5

[23] Fred E Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied optics*, 4(7):767–775, 1965. 3, 4

[24] Peiran Ren, Jiaping Wang, John Snyder, Xin Tong, and Baining Guo. Pocket reflectometry. In *ACM Transactions on Graphics (TOG)*, volume 30, page 45. ACM, 2011. 9, 10

[25] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. 34

[26] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*, 2016. 35, 36, 37

[27] Rob Tuytel. texturehaven.com. `https://texturehaven.com/textures/`. Online; accessed 22-December-2020. 29

[28] Raquel Vidaurre, Dan Casas, Elena Garces, and Jorge Lopez-Moreno. Brdf estimation of complex materials with nested learning. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1347–1356. IEEE, 2019. 10, 12

[29] Gregory J Ward. Measuring and modeling anisotropic reflection. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 265–272, 1992. 5

[30] Chris Wynn. An introduction to brdf-based lighting. *Nvidia Corporation*, 2000. 4