Bachelor Thesis

To be decided

Author
Saip Can Hasbay

desired academic degree
Bachelor of Science (BSc)

Vienna, 2022

Studienkennzahl lt. Studienblatt:	UA 033 661
Fachrichtung:	Informatik - Medieninformatik
Betreuerin / Betreuer:	Projektass.(FWF) PhD David Hahn

Contents

1 Introduction **6**
- 1.1 Physical Based Rendering . . . 6
- 1.2 Ray Tracing . . . 6
- 1.3 The BSDF . . . 7
- 1.4 The Rendering Equation . . . 8
- 1.5 Physical Based Inverse Rendering . . . 9
- 1.6 The Differential Rendering Equation . . . 10
- 1.7 Contributions . . . 11

2 Related Work **12**
- 2.1 Physics-Based Differentiable Rendering: A Comprehensive Introduction [42] . . . 12
- 2.2 Monte Carlo Estimators for Differential Light Transport [40] . . . 12
- 2.3 Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering [28] . . . 13
- 2.4 Path Replay Backpropagation: Differentiating Light Paths using Constant Memory and Linear Time [31] . . . 13
- 2.5 Unbiased Inverse Volume Rendering with Differential Trackers [27] 13
- 2.6 Inverse Volume Rendering with Material Dictionaries [11] . . . 14
- 2.7 An Inverse Rendering Approach for Heterogeneous Translucent Materials [39] . . . 14
- 2.8 Reconstructing Translucent Objects Using Differentiable Rendering [10] . . . 14
- 2.9 NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis [24] . . . 14
- 2.10 ReLU Fields: The Little Non-linearity That Could [20] . . . 15
- 2.11 Instant Neural Graphics Primitives with a Multiresolution Hash Encoding [25] . . . 15
- 2.12 Extracting Triangular 3D Models, Materials, and Lighting From Images [26] . . . 15

3 Approach **16**
- 3.1 The Algorithm . . . 16
  - 3.1.1 Example: Algorithm in use . . . 17
- 3.2 Design Decisions . . . 18
  - 3.2.1 Showcase: *Material-Optimizer* . . . 21
- 3.3 System . . . 22
  - 3.3.1 Architecture . . . 22
  - 3.3.2 Input and Output . . . 24

4 Implementation **26**
- 4.1 Workflow: Scene file and reference image acquisition . . . 26
- 4.2 Program Details . . . 28
  - 4.2.1 ***Step 1: Load a scene file*** . . . 28

4.2.2 ***Step 2: Load reference images*** 28
4.2.3 ***Step 3: Choose scene parameters for the optimization*** 29
4.2.4 ***Step 4: Initialize Adam optimizer*** 30
4.2.5 ***Step 5: Choose an optimization function*** 30
4.2.6 ***Step 6: Choosing hyperparameters*** 31
4.2.7 ***Step 7: The optimization loop*** 31

5 Evaluation **33**
5.1 Synthetic Data 33
5.2 Real-World Data 37

6 Conlusion and Future Work **40**

Acknowledgments

The author would like to thank the following people who directly or indirectly influenced the completion of this project. First and foremost, special thanks to Projektass.(FWF) Ph.D. David Hahn and Ass.Prof. Dipl.-Ing. Dr.techn. Peter Ferschin for their support and suggestions during this project. Another special thanks to Prof. Torsten Möller, Ph.D., for his help during the topic discovery phase. This endeavor would not have been possible without the Mitsuba team at EPFL in Switzerland, which provided an incredible tool for the author and the whole computer graphics community. Thanks should also go to Siemens Mobility GmbH Austria, specifically to the project leads and members at project CARMEN-Kien Bui and project CIRCE-Csaba Kis for their understanding throughout this period. Words cannot express this author's gratitude to Robert Kollruss, who had been an excellent teacher and colleague. Lastly, but most importantly, the author would like to thank his family for the hardships they had to endure and for fanning the flames of education through these years for the love of their son and brother.

Abstract

In this project, we explore material reconstruction using the mitsuba renderer. For inverse rendering, we propose a mini-tool: the *material-optimizer*. *Material-optimizer* is a tool capable of reconstructing material properties by loading a scene description file and multiple images. In essence, the material optimizer implements a gradient-based optimization algorithm that each step improves the differentiable objective function to acquire the suitable material properties from a given reference image. The main focus of this project relies on translucent material reconstruction using synthetic and real-world data. We propose two ways to reconstruct translucent materials: first, with Principled BSDF, and second, with Rough/Thindielectric with a homogeneous volume interior. Lastly, to increase the reproducibility of the shown examples and help the future use of this tool, we also proposed a simple workflow to acquire the necessary scene and reference image files.

1 Introduction

In this section, we ~~are going to~~ present the main ideas for this thesis. Firstly, we will introduce a relatively well-known field of computer graphics: physically-based rendering. Following that, we will explore inverse rendering - a growing branch of computer graphics that is becoming extremely popular ~~every other day~~. Lastly, borrowing from the ideas we introduced, we will summarize this thesis's work and list our contributions.

1.1 Physical Based Rendering

The main goal of physically based rendering is to create imagery of how our eyes basically see the world. Moreover, our goal is to generate fascinating but often unnoticed visuals constrained by the physical rules we encounter daily using our computers. More formally, as stated in the bible of physically based rendering:

"At the highest level of abstraction, rendering is the process of converting a description of a three-dimensional scene into an image. Algorithms for animation, geometric modeling, texturing, and other areas of computer graphics all must pass their results through some sort of rendering process so that they can be made visible in an image." [29].

Photorealistic rendering is often easier said than done. However, with the research work accomplished in the last fifty years by the scientific community and industry, we can create imagery that one might argue is hard to separate from reality Figure 1.

Figure 1: The Grey & White Room by Wig42 [32]

1.2 Ray Tracing

Ray tracing is one of the most known methods to generate photorealistic images. As the name suggests, the idea is to send out rays from a pixel of a virtual camera and trace them throughout the scene. In their journey, most rays interact with the objects defined in the scene. Rays continue their journey by getting scattered from the objects defined in the scene. Eventually, each ray in the scene finishes

its journey by getting absorbed by the objects of the scene and providing color information for its original pixel location. The combination of these pixels makes the rendered image. In other words:

"Almost all photorealistic rendering systems are based on the ray-tracing algorithm. Ray tracing is actually a very simple algorithm; it is based on following the path of a ray of light through a scene as it interacts with and bounces off objects in an environment" [29].

It must be noted that both explanations are a highly simplified and shortened version of the ray tracing procedure. Still, in essence, this is how we create photorealistic images with computers. The process is also shown visually in Figure 2.

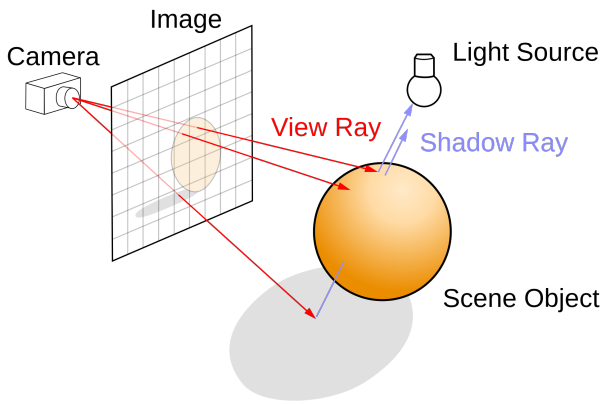

Figure 2: Ray Tracing Diagram by Henrik, CC BY-SA 4.0, via Wikimedia Commons [14]

1.3 The BSDF

The introduction of ray tracing brings another critical point that one must consider: how rays are scattered around in a scene. Since one of the things we underlined so far was physically based rendering, we must ensure that the rays we are tracing follow physical rules while they scatter around the scene.

For a moment, if we stop and observe our environment, we will see that light is scattered around from objects in our environment. Consequently, if we could model that interaction of light bouncing from one object to another in mathematical terms, then we could apply those rules to our ray tracing procedure. This abstraction of how rays are scattered around when they interact with objects in the scene is often modeled through the BSDF.

The BSDF, known as the bidirectional scattering distribution function, is a function that describes the material properties of an object in the scene. The BSDF is often also described as BSDF = BRDF + BTDF. The BRDF (bidirectional reflectance distribution function) describes an object's reflective

and the BTDF (bidirectional transmissive distribution function) transmissive material properties. In other words:

"Each object in the scene provides a material, which is a description of its appearance properties at each point on the surface. This description is given by the bidirectional reflectance distribution function (BRDF). This function tells us how much energy is reflected from an incoming direction ω_i to an outgoing direction ω_o. We will write the BRDF at p as $f_r(p, \omega_i, \omega_o)$." [29]

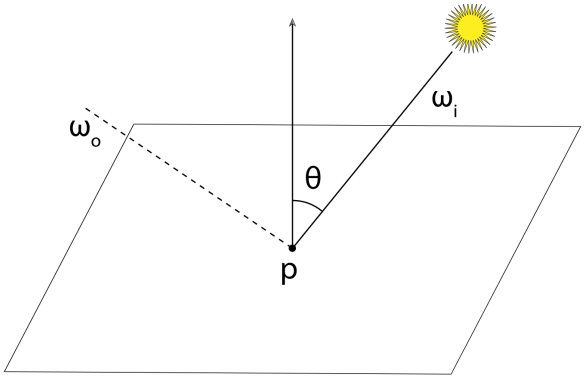

Figure 3: The Geometry of Surface Scattering, PBRT Chapter 1.2.5 Surface Scattering [29]

1.4 The Rendering Equation

The rendering equation [19], also mentioned as the light transport equation, is a recursive mathematical formulation of balanced light-energy distribution. It provides a mathematical model to compute the lighting at a surface.

In other words, it gives the total reflected radiance at a point on a surface in terms of emission from the surface, its BSDF, and the distribution of incident illumination arriving at the point. [29]

The detail that makes evaluating the LTE difficult is the fact that incident radiance at a point is affected by the geometry and scattering properties of all of the objects in the scene. For example, a bright light shining on a red object may cause a reddish tint on nearby objects in the scene, or glass may focus light into caustic patterns on a tabletop. Rendering algorithms that account for this complexity are often called global illumination algorithms, to differentiate them from local illumination algorithms that use only information about the local surface properties in their shading computations.[29]

The outgoing radiance L_o is described as the sum of the emitted radiance of the underlying surface and the reflected radiance [18]:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + L_r(p, \omega_o) \tag{1}$$

where $L_o(p, \omega_o)$ describes the outgoing radiance at position p in the direction of ω, and $L_e(p, \omega_o)$ describes the emitted and $L_r(p, \omega_o)$ reflected radiance at position p in the direction of ω.

More completely, the hemispherical form of the rendering equation is described as follows [29][6][18]:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) cos(\theta_i) d\omega_i \tag{2}$$

where as in Equation 1, $L_e(p, \omega_o)$ describes the emitted radiance at the surface location p and the reflected radiance is the integral over the hemisphere of incident light $L_i(p, \omega_i)$ weighted by the angle of incoming light $cos(\theta_i)$ (which can be also formulated as $\omega_i \cdot n$ where n is the surface normal, and consequently describes the weakening factor of outward irradiance due to incident angle, as the light flux is smeared across a surface whose area is larger than the projected area perpendicular to the ray) and the BSDF (material properties) $f_r(p, \omega_i, \omega_o)$ at the surface location p.

The formulation (however, in different mathematical notation) is also shown in Figure 4.

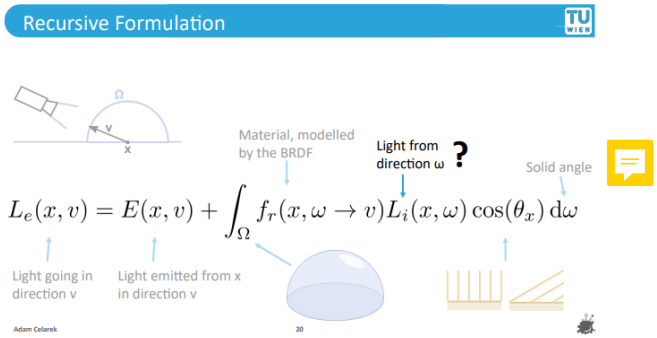

Figure 4: Rendering Equation (Recursive Formulation) by Adam Celarek, Lecture: VU Rendering TU Vienna [6]

For simplification reasons, we are going to omit the volumetric rendering equation. However, please refer to PBRT to understand the case of participating media in the scene. [29]

1.5 Physical Based Inverse Rendering

In physical-based rendering, our goal is to generate photorealistic images from a well-defined scene description. As mentioned previously, physical-based rendering is a relatively mature branch of computer graphics. In the last fifty years, the scientific community and industry have managed to solve many problems and will continue to understand the field of image synthesis.

On the other hand, inverse rendering - although not new - is a relatively young branch of computer graphics. However, the recent developments in computer graphics and hardware have awoken much interest and made the field of inverse rendering accessible.

As the name suggests, in inverse rendering, we have to think in the opposite manner. In inverse rendering, our task is to regain a scene description from an image. Intuitively, one would maybe think that it is not a complicated problem. For example, if we have an image of a red book, one would instinctively argue that the scene description should contain a book object with red material. However, we should remember that our goal is to describe things in a physically accurate manner. The red book in the image is actually a combination of red pixels, and there could be several reasons why the image contains red pixels. In the simplest case, the red pixel might result from the red-colored diffuse material. However, it might also be the result of an indirect global illumination effect, such as a reflection of a red object on the book. Figure 5 illustrates the complexity.

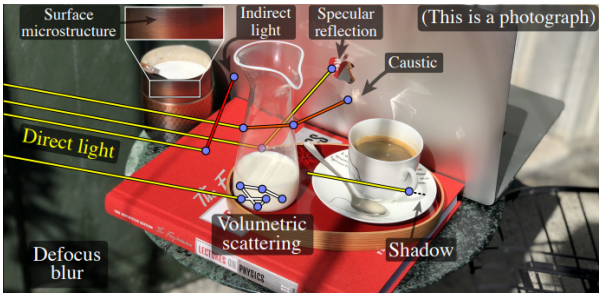

Figure 5: Image from the Introduction Physics-based differentiable rendering[42]

So far, we have described the rendering of an image as a function of $f(x) = y$, where y is the rendered image, and x is the scene description. In inverse rendering, we aim to invert this function and estimate x. This goal is often achieved by differentiating f(x), thus the rendering equation.

1.6 The Differential Rendering Equation

In this short section, we are going to present the differential rendering equation. Please refer to Physics-Based Differentiable Rendering: A Comprehensive Introduction for the complete derivation of the differential rendering equation.

We showed the rendering equation in 2. The differential rendering equation is another integral equation that can be solved with Reynold's transport theorem [33]. The differentiable rendering equation can be described as a combination of interior and boundary terms and is shown in Figure 6:

As previously in the case of the volumetric rendering equation, we omit the equation of Differentiable Rendering of Participating Media. For further under-

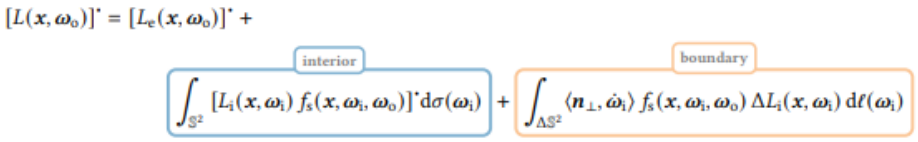

Figure 6: The Differentiable Rendering Equation from the Introduction Physics-based differentiable rendering [42]

standing, please refer to Physics-Based Differentiable Rendering: A Comprehensive Introduction and A Differential Theory of Radiative Transfer. [42][41]

To summarize, to accomplish the task of estimating scene parameters x from a given image y, we need to use a technique known as differentiable rendering. With this technique, we can estimate the physical attributes of a scene, e.g., material properties (BSDF attributes), lighting properties, and geometry of the objects. The differentiable rendered is used in the following manner:

"The function f is mathematically differentiated to obtain $\frac{dy}{dx}$, providing a first-order approximation of how a desired change in the output y (the rendering) can be achieved by changing the inputs x (the scene description). Together with a differentiable objective function $g(y)$ that quantifies the suitability of tentative scene parameters, a gradient-based optimization algorithm such as stochastic gradient descent or Adam can be used to find a sequence of scene parameters x0, x1, x2, etc., that successively improve the objective function." [30]

Figure 7 describes this procedure.

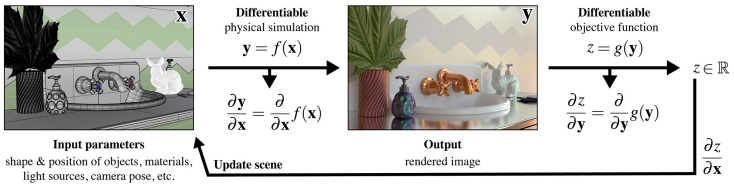

Figure 7: Inverse rendering [42][30]

1.7 Contributions

So far, we have accumulated background information from the field of computer graphics. Now that we can roughly understand how a computer can generate beautiful images and simultaneously reacquire scene descriptions from an image, we can explore what this thesis covers. Respectively, our contributions include the following:

- An optimization procedure to acquire scene parameters (i.e., material properties) from an image using the mitsuba renderer. Specifically, the

goal is to reconstruct translucent material properties from real and synthetic images. However, the procedure can additionally reconstruct opaque material properties.

- A mini GUI tool that incorporates the full power of the mitsuba renderer and various features to accomplish the task of inverse rendering.
- A flexible code base in python that makes use of the parallel computing power of NVIDIA GPUs to accomplish the task of inverse rendering.
- A workflow to increase the reproducibility of the shown examples and help the future use of the proposed tool.

2 Related Work

Inverse rendering is a young and developing branch of computer graphics. Nevertheless, the task of material or geometry reconstruction is not a new area of scientific research. In addition to the research work that has been accomplished in this field previously, the scientific community and the industry have observed exciting developments in the last couple of years. In this section, we are going to explore what has been accomplished previously.

2.1 Physics-Based Differentiable Rendering: A Comprehensive Introduction [42]

Physics-Based Differentiable Rendering: A Comprehensive Introduction was one of the lectures in SIGGRAPH 2020, which provided an in-depth introduction to the field of inverse rendering and, general-purpose physics-based differentiable rendering. This course also contains a discussion of many challenges that involve differentiable rendering. For example, the differentiation of many scene parameters (e.g., $10^6 - 10^{10}$) or boundaries of objects that introduce troublesome discontinuities during the computation of shadows and interreflections that lead to incorrect gradients. Lastly, the lecture also provides a simple differentiable renderer implementation in which the students can explore the practical challenges of differentiable rendering.

2.2 Monte Carlo Estimators for Differential Light Transport [40]

Nowadays, many physical-based renderer implementations use Monte Carlo techniques and require different design decisions, such as sampling the BSDF and light sources or combining them with multiple importance sampling. The task of inverse rendering involves differentiation of the rendering equation. Thus the complication of having various design choices in a renderer and how the rendering equation is implemented brings a subsequent number of questions for a differentiable renderer. For example: “Should we differentiate only the

estimator, or also the sampling technique? Should MIS be applied before or after differentiation? How should visibility-related discontinuities be handled when millions of parameters are differentiated simultaneously?". In this paper, Zeltner et al. analyze the vast space cases in differentiable rendering and mathematically classify various estimators for differential light transport.

2.3 Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering [28]

Physical-based differentiable rendering executes a differentiable simulation of light transport and scattering effects to estimate partial derivatives relating scene parameters to pixels in the rendered image. Often this task is achieved using reverse mode differentiation that computes all requested scene parameter derivatives at once. The extreme amount of free variables in reverse mode differentiation usually requires extensive detailed transcripts of the computation that is subsequently replayed to back-propagate derivatives to the scene parameters. This need often results in large memory consumption. Thus the task of inverse rendering is usually limited to a simple scene with low resolutions. In this paper, Nimier-David et al. introduce a new approach that does not require highly detailed and numerous transcripts, significantly improving the scalability and efficiency of inverse rendering.

2.4 Path Replay Backpropagation: Differentiating Light Paths using Constant Memory and Linear Time [31]

Physical-based differentiable rendering has become the primary method to solve the tasks in inverse rendering. Reverse mode differentiation is the go-to method that physical-based differentiable renderers use to obtain scene parameter gradients. Prior to this work, many techniques used statistically biased methods or suffered from high memory and computation time. This work extends the work of Radiative Backpropogation [28], in which Vicini et al. propose a new backpropagation algorithm that only requires constant memory and linear computation time. Additionally, this method provides a method to handle highly specular materials such as smooth dielectrics and conductors, which mitsuba 3 [16]; thus, this thesis relies upon.

2.5 Unbiased Inverse Volume Rendering with Differential Trackers [27]

In inverse rendering, volumetric representations are commonly used since volumetric models have simple parameterization, are smoothly varying, and transparently handle topology changes. One of the challenges is estimating the volume's appearance concerning its scattering and absorption parameters. In this paper, Nimier-David et al. show that a naive approach results in biased and high-variance gradients. Nimier-David et al. propose a new sampling strategy

to unbiasedly estimate the volume's appearance. Moreover, the new approach yields low-variance gradients and runs in linear time.

2.6 Inverse Volume Rendering with Material Dictionaries [11]

This paper is one of the initial inverse rendering research regarding the reconstruction of heterogeneous translucent materials. Gkioulekas et al. introduce an optimization framework to measure the bulk scattering properties of homogeneous materials. Similar to this thesis work, the optimization framework incorporates stochastic gradient descent with Monte Carlo rendering. However, it additionally uses material dictionaries to invert the radiative transfer equation.

2.7 An Inverse Rendering Approach for Heterogeneous Translucent Materials [39]

This paper proposes another inverse rendering approach for heterogeneous translucent materials from a single input photograph. The proposed method can obtain the material distribution and estimate heterogeneous material parameters to render images similar to the provided reference image. This work is similar to this thesis because both pieces can try to reconstruct translucent materials from images. Additionally, as one method in this thesis, this paper achieves the task of reconstruction using volumetric data.

2.8 Reconstructing Translucent Objects Using Differentiable Rendering [10]

This relatively recent paper introduces another approach to reconstructing translucent objects using differentiable rendering. In this paper, Deng et al. extend previous methods using bidirectional scattering-surface reflectance distribution function (BSSRDF) for translucent materials. Moreover, to handle the noise introduced by the BSSRDF integral, Deng et al. propose a dual-buffer method for evaluating the loss during optimization. In this thesis, we make use of this proposed dual-buffer method and observe faster and correct convergence during the optimization procedure.

2.9 NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis [24]

This groundbreaking paper, published in 2020, proposes a new approach to reconstructing novel views of complex scenes by optimizing an underlying continuous volumetric scene function using a sparse set of input views. The algorithm offers to represent a scene using a fully connected neural network, where the input is a 5D coordinate (spatial location x, y, z) and viewing direction (θ, ϕ) and

whose output is the volume density and view-dependent emitted radiance at that spatial location. Since volume rendering is differentiable, the optimization only requires one input: a set of images with known camera poses. Mildenhall et al. describe the procedure to reconstruct geometry and appearance.

2.10 ReLU Fields: The Little Non-linearity That Could [20]

Currently, many methods make use of the power of multi-layer perceptrons (MLPs) to model complex spatially-varying functions such as images and 3D scenes. However, the use of MLPs often comes at the cost of long training and inference times. On the contrary, voxel grid representations of scenes can give fast training and inference times; however, it is not able to reach the quality of MLPs without significant memory consumption. In this paper, Karnewar et al. propose a method that allows fixed non-linearity (ReLU) on interpolated grid values that represents complex signals such as images or 3D scenes on regularly sampled grid vertices.

2.11 Instant Neural Graphics Primitives with a Multiresolution Hash Encoding [25]

In 2020 although NeRFs [24] provided an extremely innovative approach, its most significant problem was the costly training time of the neural network to reconstruct a novel view of complex scenes. In this paper, Mueller et al. extend prior work in NeRFs by reducing the costly training time with a versatile new input encoding scheme that allows the use of smaller neural networks. Consequently, Mueller et al. significantly reduce memory access operations during optimization. Although, in essence, NeRF is a different approach to how in this thesis, we deal with material reconstruction, with this award-winning SIGGRAPH 2022 paper, Mueller et al. make NeRFs accessible and provide another method for reconstructing views.

2.12 Extracting Triangular 3D Models, Materials, and Lighting From Images [26]

As in instant neural graphics [25], another fascinating paper by NVIDIA that is published in 2022 presents an efficient method for joint optimization of topology, materials, and lighting from multi-view image observations. However, in this work, Munkberg et al. provide an approach where the output is triangle meshes with spatially-varying materials and environment lighting instead of 3D representation encoded in neural networks. This paper also indicates an approach where constructed volumetric representation by NeRFs can be extracted in 3D models to use in 3D engines. The suggested pipeline by Munkberg et al. is NeRF [24] $\rightarrow$ Marching Cubes [22] $\rightarrow$ Differentiable renderer. Thus, this paper connects the two worlds of NeRFs and differentiable rendering. Consequently,

one can argue that this thesis work can be a part of the suggested pipeline to reconstruct geometry and materials.

3 Approach

Now that we have basic background knowledge about rendering and inverse rendering, we are going to introduce *material-optimizer*: the mini tool that is used in this work to reconstruct BSDF (material) properties from images. To understand how this task is achieved, we will first discuss the cooking recipe (or, more fancier: *the algorithm*). Secondly, we will examine the design decisions behind *material-optimizer*. And lastly, from a more computer science point of view, we are going to analyze the architecture of *material-optimizer*.

3.1 The Algorithm

As we previously discussed and visualized in Figure 7, we need a differentiable renderer to achieve the task of inverse rendering. *Material-optimizer* entirely makes use of mitsuba 3 differentiable rendering capability [16].

We mathematically note the image generation procedure (i.e., rendering) as a function $f(x)$, where $f(x)$ converts the scene description x into an image y. To acquire a first-order approximation of how a desired change in the image can be achieved by changing the scene description, we differentiate the $f(x)$ and obtain $\frac{d\mathbf{y}}{d\mathbf{x}}$. We then use a gradient-based optimization algorithm Adam [21] that sequentially improves the differentiable objective function $g(y)$ to acquire the suitable material properties from a given reference image.

With mitsuba 3 (i.e., also Dr.Jit [17]) under its hood, the *material-optimizer* follows the subsequent procedure to reconstruct material properties from a given reference image:

1. Load an initial scene file that contains the scene description (e.g., object locations, camera locations, material properties of objects). Default: Cornell box scene [34].
2. Load reference images.
3. Choose parameter/s (such as the color of an object in the scene) that will be successively optimized to recover the look in the reference image.
4. Initialize the Adam optimizer with a learning rate and assign chosen parameters from Step 3. Default learning rate: 0.03.
5. Pick an objective function g(y). Default: Mean square error (or also known as L2 error) between the current image and the reference image.
6. (Optional) Choose hyper-parameters to control the optimization loop. Default: iteration count = 100, minimum error = 0.001, samples per pixel = 4.

7. Start the optimization procedure and stop if the defined minimum error or iteration count is achieved.

   (a) For each camera pose/location (i.e. loaded reference image):

      i. From the current camera pose, perform a differentiable rendering of the scene from Step 1.

      ii. Evaluate the objective function from Step 5 with the rendered image from Step 7(a)i and corresponding loaded reference image.

      iii. Backpropogate the rendering process using Dr.Jit [17].

      iv. Take a ~~gradient descent~~ step with the optimizer from Step 4.

      v. Ensure legal values for the optimized parameters from Steps 3 and 7(a)iv

      vi. Update the scene with the optimized parameters

The above-described procedure is not a new invention but has often been used in different works [10][11][28][30][31]. Moreover, the above-described *cooking recipe* is generalized for simplification. For example, the *material-optimizer* contains additional steps for restarting the optimization, keeping track of loss, and displaying the results. But, in essence, the above-described procedure is the main idea for material optimization. Overall this procedure is relatively simple to understand, powerful, and flexible in the sense that we may use different BSDF models (diffuse, dielectric, principled, etc.) and integrators (direct illumination integrator, path-tracer, volumetric path, etc.). For the intrinsic details of the procedure, we invite the reader to explore the provided implementation.

3.1.1 Example: Algorithm in use

In this short section, we show a simple example to understand the provided algorithm in more detail.

1. We load a modified version of the Cornell box scene (we simply change the beige/white colored walls into blue). The initial look of the scene can be found in Figure 8.

2. We load the original render of the Cornell box scene (i.e. only one reference image and camera location). The original render is also shown in Figure 8.

3. We pick white-colored walls as parameters to optimize (more precisely, mitsuba provides the name of the assigned BSDF parameter as *'white.reflectance.value'*).

4. We choose the default learning rate (0.03) for the Adam optimizer. The initialization and assignment of the chosen parameter happen automatically by the *material-optimizer*.

5. We pick the default optimization function (i.e., mean squared error).

6. We choose default hyper-parameters.

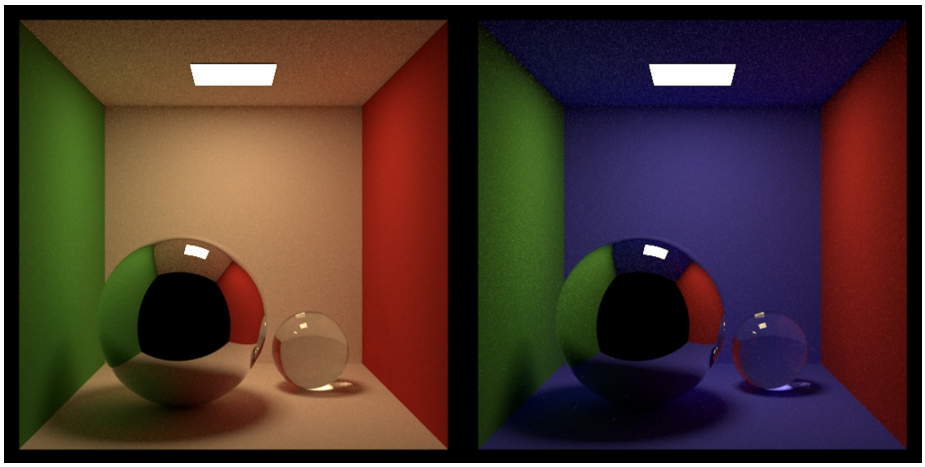

Figure 8: Left: Reference image. Right: Render of the initial state of the scene.

7. We start the optimization, and the material optimizer completes the steps defined accordingly (Step 7):

The result is shown in Figure 9.

3.2 Design Decisions

In this section, we are going to discuss the design decisions behind the *material-optimizer*. *Material-optimizer* basically wraps mitsuba 3 and provides a graphical user interface to accomplish the task of inverse rendering for provided scenes and images.

Although there are different ways to reconstruct geometry and materials [24][26], our goal is to reconstruct materials - specifically translucent materials - using a physically-based differentiable renderer. As we discussed, much research has been done using mitsuba 3 [16]. Consequently, mitsuba 3 provides an optimal engine to accomplish the task of inverse rendering.

In the previous section, we discussed the algorithm behind the *material-optimizer*. As noted in that section, the provided algorithm is a general idea that could be used in distinct differentiable renderers. Since the provided procedure is quite generic and mitsuba 3 supplies the users with different capabilities (such as the use of various BSDFs, integrators, samplers, etc.), it is a perfect match for a mini tool that accomplishes the task of material reconstruction. Additionally, from a performance side of view, mitsuba 3 provides users with a GPU (specifically NVIDIA GPU's using the CUDA API) implementation, which allows for accomplishing the expansive task of inverse rendering in a more acceptable time period.

In this work, our primary goal is to find a way to reconstruct material properties. Consequently, the implementation and results of the provided procedure

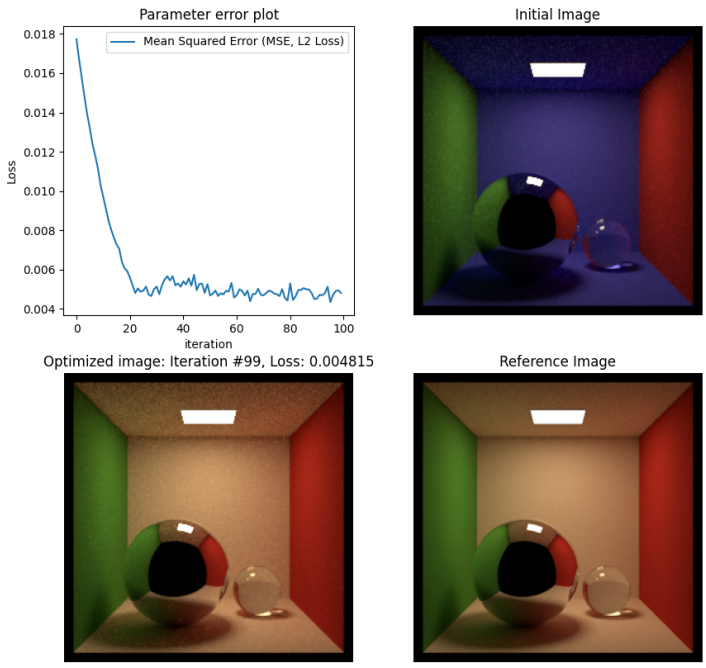

Figure 9: Top left: Parameter error plot (i.e. loss during optimization). Top right: Render of the initial state of the scene. Bottom left: Render of the optimized scene. Bottom right: Reference image.

have the highest priority. Although mitsuba 3 provides users with the right components, we quickly realized that we needed a more specific tool that would allow a professional user the task of inverse rendering. In other words, the *material-optimizer* is built for academic purposes, where the provided flexible algorithm could be executed with two-three clicks.

The task of inverse rendering could be accomplished only using mitsuba 3, however, quite often only by modifying or using multiple scripts. Since mitsuba 3 provides many components (such as multiple BSDFs, and integrators), every time the user wants to reconstruct a material from an image, the user needs to modify one of the scripts that would be necessary for the use case. For example, suppose the user wants to reconstruct the material properties of a translucent light object from a reference photograph. In that case, the user needs to pick the suitable integrator (e.g., whether to use a differentiable volumetric path tracer or simple path tracer), BSDFs, and correct resolutions for the scene.

After all these steps, the user needs to modify the script/s such that the task is accomplished. Finally, the user would get a result in a terminal window, which might be entirely incorrect; in that case, the user may need to repeat the process.

With *material-optimizer* we streamline this process into one mini tool with a graphical user interface. In its simplest form, the tool user only needs to load a mitsuba 3 scene and a reference image and pick the proper material parameters to execute the material reconstruction procedure provided in the algorithm section. After loading a scene file, the *material-optimizer* picks the qualified integrators, scene resolution, and hyperparameters and presents supported differentiable scene parameters for the optimization procedure. Suppose the material reconstruction procedure is ended with incorrect results; the user may restart the optimization procedure by changing the parameters (e.g., hyperparameters, optimized BSDF parameters, learning rate for the optimization) on the GUI rather than modifying a code base. *Material-optimizer* is shown in Figure 10.

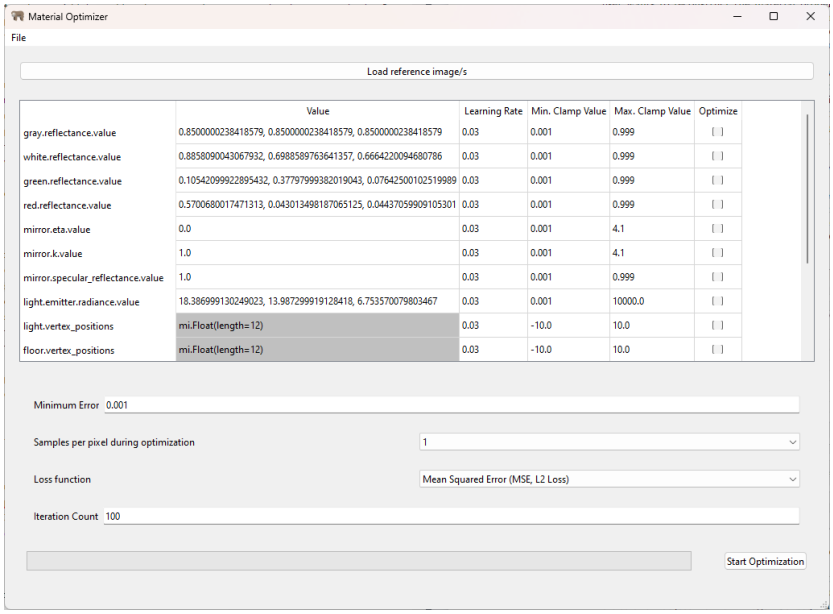

Figure 10: The look of *material-optimizer*.

As we noted previously, we expect the use of this tool for academic purposes, thus to examine whether an educated user could reconstruct the material properties of an object from images. Consequently, the appearance of the GUI is constructed with simplicity in mind rather than elegancy. The main idea behind the user interface of the *material-optimizer* is its simplicity in the code base and presentation.

3.2.1 Showcase: *Material-Optimizer*

In this short section, we follow the steps presented in the 3.1 and show the appropriate actions to take to get a similar result as in Figure 9.

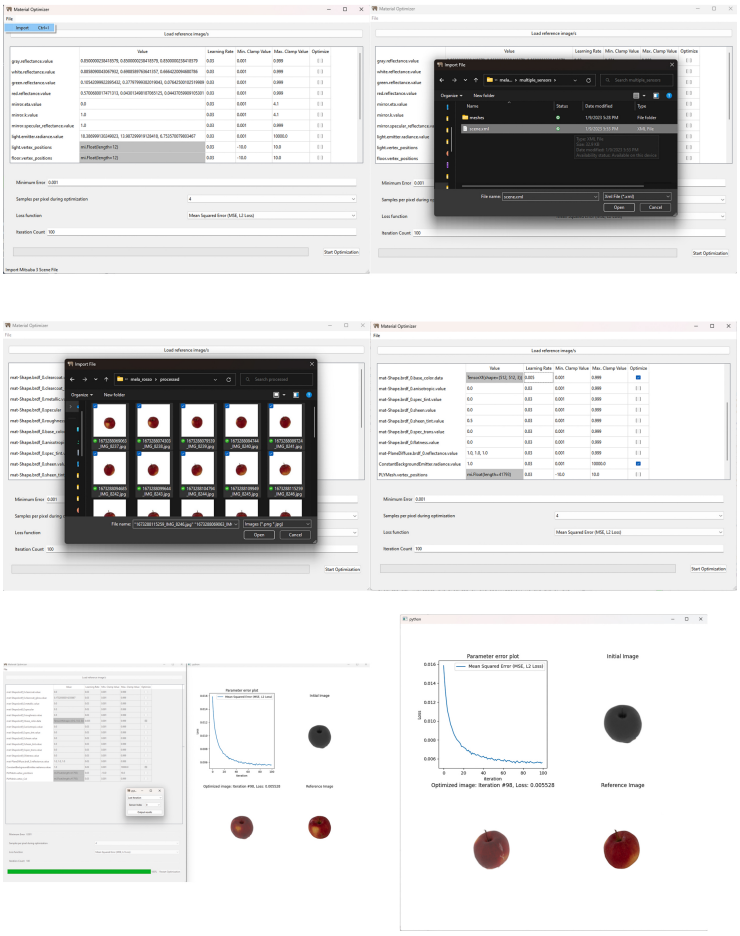

Figure 11: Configuration steps in *material-optimizer*. Top left/right: Importing a scene (XML) file. Middle left: Importing multiple reference images. Middle right: Configuration for the optimization. Bottom left: The look after the optimization is finished. Bottom right: Optimization results from another camera view.

As seen in the top left corner of Figure 11, we can import mitsuba scene files by clicking *'File'* and then *'Import'* buttons in the top left corner of the material-optimizer. Consequently, a pop-up window opens, and the user can select the scene file as in the top right corner Figure 11. In this example, we choose a scene file that contains an apple object with multiple sensors (specifically

the description contains 10 camera poses). Subsequently, the user can load a reference image by clicking the *'Load reference image/s'* button as in the middle left image in Figure 11. With this action, a pop-up window opens, and then the user can select reference images. In this example, we choose a real-life - however processed (background-removal) - images (specifically 10) of an apple. After loading a reference image, the user can select the scene parameters to optimize and modify hyperparameters. In this case, we pick the bitmap texture of the apple (represented as $mat.Shape.brdf\_0base\_color.data$) and constant emitter radiance value (represented as $ConstantBackgroundEmitter.radiance.value$) as scene parameters for the optimization, as in the middle right corner of 11. Next, we modify the learning rate of the selected bitmap texture parameter to an arbitrary value of 0.005. We leave the hyperparameters set as default values. As a last step, the user can start the optimization procedure by clicking the *'Start Optimization'* button, which is shown in the bottom left corner of Figure 11. While the optimization continues, the progress bar indicates the progress of the optimization. At the end of the optimization, material-optimizer provides a plot, as in Figure 8. Depending on the results, the user can switch between the scene state in the last iteration or the minimum optimization error by selecting the corresponding dropdown menu option as in the bottom left image of Figure 11. Moreover, if multiple reference images are provided, the user may switch between different camera poses by selecting the index value of the sensor. The result of this action is shown in the bottom right corner of Figure 11

3.3 System

In this section, we are going to discuss the system decisions behind the *material-optimizer*. First, we will introduce the architectural design choices and, afterward, the input and output of the system.

3.3.1 Architecture

Since our goal in this project primarily relies on material reconstruction, the system's architecture took less priority during the development. However, since we wanted to have a reusable and flexible system, we made sure that the components of the system were well divided. Consequently, the actions taken by the user are processed and visualized separately. Additionally, whenever a new finding emerged, the flexible design allowed easy changes without large refactorings.

Material-optimizer implements a well know Model-View-Controller pattern [36]. As the name indicates, the Model-View-Controller pattern divides the system into three components. The Model is the system's central component and contains the system's business logic. The Model is independent of the user interface, and its primary goal is to process and manage data. The View can be seen as the user interface of the system. The View visually represents information about the Model. As the name suggests, the Controller controls

the interaction between the Model and View. Its main goal is to convert user input for the Model and Controller.

Figure 12: The UML package (left) and class (right) diagram of *material-optimizer*

The UML diagram is shown in Figure 12. In this project, the Model (*MaterialOptimizerModel*) contains the appropriate logic to accomplish the task of inverse rendering. For example, the main tasks of *MaterialOptimizerModel* are:

- Implement the optimization procedure.
- Load, process, and prepare scene and reference image.
- Find appropriate integrator.
- Evaluate the objective function.
- Communicate with mitsuba 3 and Dr.Jit.

The View of *material-optimizer* contains and implements the right visual components and containers (i.e., PyQt6 components [37]) to visualize the information that comes from the *MaterialOptimizerModel*. For example, *MaterialOptimizerView* provides a table to visualize available scene parameters for the optimization. It furthermore provides text input and dropdown menus to modify the hyperparameters for the optimization. These components can be seen in Figure 10. The Controller of the *material-optimizer* is essentially the coordinator between *MaterialOptimizerModel* and *MaterialOptimizerView*. For example, if a user clicks the 'Start Optimization' button shown in Figure 10, the *MaterialOptimizerController* ensures that the modified hyperparameters and scene parameters by the user are converted such that the *MaterialOptimizerModel* can make use of them for the optimization.

3.3.2 Input and Output

In this section we are going to discuss the input and output of the *material-optimizer*. In this project, we can abstract the inputs into a mitsuba scene and a reference image. The system's output can be summarized into a file/s containing the results from the optimization procedure.

More specifically, when we explore the mitsuba input scene, we discover that the syntax must follow an XML file structure and the semantics of a scene description (e.g., objects, light positions, material properties, integrator, etc.) [4]. The Model of the system makes use of mitsuba 3 traversing a scene file capability and prepares the appropriate configuration for the system.

Correspondingly the provided reference image/s plays a vital role in the system. *Material-optimizer* uses reference image/s as the goal for the optimization. Thus, the reference image that corresponds to the camera pose will be used when evaluating the objective function. For example, if the mean squared error is being used during the optimization, the objective function can be evaluated as the mean value of $\sqrt{Image_{ref} - Image_{current}}$. *Material-optimizer* allows two formats for reference images: PNG and JPEG. However, mitsuba 3 provides a Bitmap implementation; thus, this constraint can be relaxed into different image formats in the future.

When we deep dive into the provided output forms in the *material-optimizer*, we find four formats in use. The JSON file format can be seen as the default output format in the *material-optimizer*. The outputted JSON file contains multiple key-value pairs of optimized scene parameters whenever the user clicks the 'Output' button after the optimization. For example, if the user wanted to optimize the color value (RGB color value [38]) of the walls in a scene - as shown in Example 3.1.1 - the results are:

```
{
    "gray.reflectance.value": "[[...]]",
    "white.reflectance.value": "[[0.94, 0.73, 0.68]]",
    "green.reflectance.value": "[[...]]",
    "red.reflectance.value": "[[...]]",
    "mirror.eta.value": [...],
    "mirror.k.value": [...],
    "mirror.specular_reflectance.value": [...],
    "light.emitter.radiance.value": "[[...]]",
    "light.vertex_positions": [...],
    "floor.vertex_positions": [...],
    "ceiling.vertex_positions": [...],
    "back.vertex_positions": [...],
    "greenwall.vertex_positions": [...],
    "redwall.vertex_positions": [...]
}
```

The other three types of output formats provided by the *material-optimizer* correspond to more specific scene parameters. These formats are Volume (*.vol*) [5], PNG, and NumPy Array (*.npy*) [13]. For example, as we showed in Example 3.2.1, mitsuba 3 (i.e., *material-optimizer*) provides a mechanism to optimize the bitmap texture of objects. In that case, the provided output will be an image (i.e., a PNG file). The resulting bitmap-texture output from Example 3.2.1 is shown in Figure 13. The output formats and their description is shown in Table 1.

Format	Description
`JSON`	Contains optimized scene parameters (default choice).
`PNG`	Contains optimized bitmap texture data.
`Volume (.vol)`	Contains volumetric data [5].
`NumPy Array (.npy)`	Contains multidimensional floating point arrays (e.g. vertex colors) [13].

Table 1: Provided output formats in *material-optimizer*.

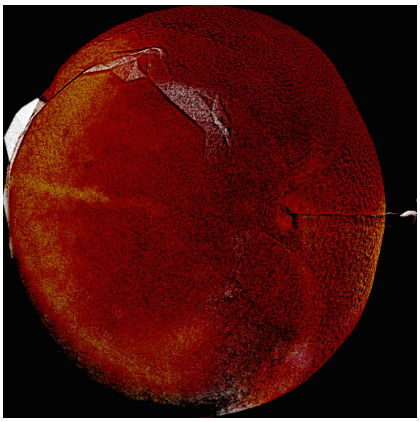

Figure 13: The outputted PNG file after the optimization as showed in Example 3.2.1.

4 Implementation

In this section, we are going to discuss the implementation details of the *material-optimizer*. More specifically, first, we are going to show the workflow details. Then, we will provide the code used by the *material-optimizer*. The *material-optimizer*'s full implementation and build instructions will be provided on the GitHub/GitLab page.

4.1 Workflow: Scene file and reference image acquisition

Before we show the code details behind the *material-optimizer*, we will show how to acquire prerequisite scene files and reference images. Although this step is not part of the *material-optimizer*, it is important to offer a simple workflow to be able to reproduce optimization tasks achieved in this paper.

Depending on the task at hand, such as material reconstruction from real-world objects or synthetic data, the user must find reference images for optimization. This task is relatively simple if synthetic data is in use. In the most simple case, the user can find an image of a rendered scene and use it in *material-optimizer*. Alternatively, the user may construct a scene in a 3D computer graphics software tool such as Blender [9] and use the rendered image in the material optimizer. In this workflow, we suggest using Blender since mitsuba 3 provides an additional Blender Add-on that is able to export constructed scene file as a Mitsuba XML file [7], which then *material-optimizer* can use.

In the case of real-world images, the task can get seriously complicated, especially highly accurate results are expected. The user must control the lighting conditions of the environment where the reference object is imaged. Additionally, the camera position and other objects in the image also play a critical role in the use of real-world data. In this workflow, for real-world data examples, we suggest the use of a photogrammetry tool such as Metashape [35] to reconstruct the geometry of the shape that will be used in the scene file and refers to the object in the reference image. The geometry reconstruction step can also be achieved with different NeRF procedures [24][25][20][23]. However, one needs to ensure that if the reconstructed geometry is in volumetric representation, the representation must be converted into the mesh representation, especially BSDF reconstruction will be the main interest in the further steps. Geometry reconstruction on its own is a fascinating and challenging research topic, and we will omit the details in this project.

In this project, our workflow for real-world images - unless specified otherwise - is the following [1] (e.g. this procedure is used reconstruct the geometry of the apple in Example 3.2.1):

1. Take multiple photos of an object.
2. Load and align images in Metashape [35].
3. Build Mesh in Metashape.

[1] This procedure might have difficulties with translucent/transparent objects.

4. Export Mesh and Camera Locations in Metashape (Suggested Format: X3D).

5. Import the X3D file from Step 4 in Blender [9] (i.e., Blender scene is acquired).

As in the synthetic image usage, once we have a scene in Blender, we may use the Mitsuba Blender Add-on to export it as a Mitsuba XML scene to use in the *material-optimizer*. In the delivered GitLab/GitHub repository, we provide the user with the necessary reference images and metashape files to reproduce the Example 3.2.1.

The above-described workflow is one of the many solutions to acquire a mitsuba scene file. Consequently, the *material-optimizer* is only interested in a mitsuba scene file and reference images and does not involve in the suggested workflow. However, since the *material-optimizer* supports vertex position optimization, one can argue that the *material-optimizer* could provide the following procedure to reconstruct geometry and materials:

1. Construct a simple mitsuba scene with a light source and a sphere (or any other simple object) with multiple vertex positions [2].

2. Import the scene from Step 1 in the *material-optimizer*.

3. Load reference images of an object.

4. Select vertex positions of the sphere from Step 1 for the optimization procedure.

5. Pick appropriate optimization parameters (e.g., learning rate, min., and max. clamp values) and hyperparameters.

6. Start the optimization procedure as in Step 7.

7. Export results (i.e., vertex pos. and consequently reconstructed mesh).

8. Use the reconstructed mesh from Step 7 in a mitsuba scene file.

9. Use the scene file from Step 7 to reconstruct material properties, thus follow the steps provided in Section 3.1.

One should note that mesh reconstruction is not the primary target of the *material-optimizer*. The above-described procedure 4.1 may provide unsuccessful results since geometry reconstruction might require additional constraints not implemented in the *material-optimizer*.

[2] The number vertex positions might play a critical role in this procedure.

4.2 Program Details

In this section, we are going to show the implementation details behind the material-optimizer. Since, in this section, it is not feasible to examine the full implementation, we are going to discover code details behind the algorithm of the material-optimizer presented in Section 3.1. As mentioned previously, for the full implementation, we invite the reader to explore the provided GitHub/GitLab repository.

4.2.1 *Step 1: Load a scene file*

Whenever the user clicks the 'Import File' button, the View (MaterialOptimizerView) notifies (or triggers) the Controller (MaterialOptimizerController) that the user wants to load a scene file. The Controller then calls the following method:

```
1  def loadMitsubaScene(self):
2      try:
3          fileName = self.view.showFileDialog(XML_FILE_FILTER_STRING)
4          self.model.loadMitsubaScene(fileName)
5          self.model.resetSensorToReferenceImageDict()
6          self.model.setSceneParams(self.model.scene)
7          self.model.setInitialSceneParams(self.model.sceneParams)
8          self.model.setDefaultOptimizationParams(
9              self.model.initialSceneParams)
10         self.updateTable(self.model.initialSceneParams,
11             self.model.optimizationParams)
12     except Exception as err:
13         self.model.loadMitsubaScene()
14         self.view.showInfoMessageBox(...)
```

The *loadMitsubaScene(self)* method can be summarized in the following way: first, the Controller instructs the View to show a Window so the user can pick an existing scene file (*line 3*). Then, the Controller passes the selected scene file to the Model. The Model parses the XML file and selects an Integrator depending on the patterns found (*line 4*). For example, if the file contains a scene parameter that might introduce discontinuities, then the material-optimizer selects the *'prb_reparam'* integrator [2]. Next, the Controller instructs the Model to reset the data structure that contains a list of key-value pairs, where the key refers to sensors (i.e., camera positions) and the values corresponding reference image (*line 5*). In *line 6*, the Model saves the scene parameters. Next, the Model creates a new data structure that contains the scene parameters supported in the material-optimizer (*line 7*). In *line 8*, the Model sets the default optimization parameters (e.g., the learning rate). Lastly, the Controller passes the new scene parameters and instructs the View to update its table (*line 10*). If an error occurs during these steps, the Controller instructs the View to show the affiliated error message (*line 16*). Additionally, in that case, the Controller instructs the Model the load the default mitsuba scene (i.e. the Cornell Box Scene, *line 15*).

4.2.2 *Step 2: Load reference images*

Every time the user clicks the 'Load reference image/s' button, the Controller is triggered by the View for the user to select reference images. The implemen-

tation of Step 2 is done in the following manner:

```
1  def loadReferenceImages(self):
2      try:
3          refImgFileNames = self.view.showFileDialog(
4              IMAGES_FILE_FILTER_STRING, isMultipleSelectionOk=True
5          )
6          # check length of reference images is equal to # of sensors
7          if len(refImgFileNames) != len(self.model.scene.sensors()):
8              raise RuntimeError(...)
9          readImgs = [
10             self.model.readImage(refImgFileName)
11             for refImgFileName in refImgFileNames
12         ]
13         # Create a dictionary where key: sensor, value: refImage
14         self.model.setSensorToReferenceImageDict(readImgs)
15         self.view.showInfoMessageBox(...)
16     except Exception as err:
17         logging.error(...)
18         self.view.showInfoMessageBox(...)
```

As in the file import case, the View shows the user a dialog box to choose reference images (*line 3*). Then, the material-optimizer checks that the number of picked images equals the available sensors loaded in the scene (*line 7*). If not, an error is displayed to the user (*line 8*). In the next step, from the selected files, the Model constructs Bitmap Objects according to the specified scene resolution in the loaded scene file; however, it considers the image aspect ratio [3] (*line 9*). Next, from the constructed Bitmap objects, the Model assembles a new data structure that contains a list of key-value pairs, where the key refers to sensors (i.e., camera positions) and the values corresponding to a reference image (*line 14*). In the last step, the Controller instructs the View to inform the user that the reference images are successfully loaded (*line 15*). Similar to the *loadMitsubaScene(self)* method, the View displays the corresponding error message to the user in case of failure.

4.2.3 *Step 3: Choose scene parameters for the optimization*

The material-optimizer allows the users to modify the initial value of loaded scene parameters if the scene parameter is an RGB or simple floating point number. In this section, we omit to show the details of this functionality; however, the reader could find the implementation in the *onSceneParamChanged()* method. Similarly, the reader may refer to the *onOptimizationParamChanged()* method to find the implementation details behind setting optimization values, such as the learning rate of a specific scene parameter for the optimization.

Whenever the user clicks the 'Start Optimization' button, the Controller calls the *optimizeMaterials()* method, which is shown here:

[3] Material-optimizer uses 256 as the fix height value. The new resolution is computed as $(256 \times ImageAspectRatio, 256)$. The main reason behind this resolution specificity is the Author's limited CUDA Memory.

```
1  def optimizeMaterials(self):
2      # Precondition: (1) reference image/s is/are loaded, and
3      #               (2) at least one checked scene parameter
4      checkedRows = self.getCheckedRows()
5      if (self.model.sensorToReferenceImageDict is None
6          or len(checkedRows) <= 0):
7          self.view.showInfoMessageBox(...)
8          return
9
10     ... # for now we omit the details, please refer to next steps
```

First, the *material-optimizer* gathers the checked scene parameters (*line 3*) and then controls that at least one reference image is loaded and one scene parameter is selected (*line 5*). If this is not the case, the *material-optimizer* informs the user that the optimization cannot begin. In the following sections, we will continue to discover the *optimizeMaterials()* method.

4.2.4 *Step 4: Initialize Adam optimizer*

If the preconditions are met, the Controller instructs the Model to prepare for optimization.

```
1  def optimizeMaterials(self):
2      # Precondition: (1) reference image/s is/are loaded, and
3      #               (2) at least one checked scene parameter
4      # The implementation is as in the previous sections
5      ...
6      # Omitting View dependent details...
7      opts, initImg = self.model.prepareOptimization(checkedRows)
8      # Omitting View dependent details...
9      # initiate the optimization loop
10     lossHist, sceneParamsHist = self.model.optimizationLoop(
11         opts, lambda x: self.view.progressBar.setValue(x)
12     )
13     # Omitting View dependent details...
```

The Model prepares for the optimization by receiving selected scene parameters and, from them, initializes the Adam optimizer. Subsequently, from the initial scene parameters the Model makes makes the initial render of the loaded scene (*line 7*). The details can be found in the *prepareOptimization()* method.

4.2.5 *Step 5: Choose an optimization function*

As mentioned in Section 3.1 at Step 5, the user is always provided with the default optimization function mean squared error. The implementation looks like this:

```
1  def mse(self, image, refImage):
2      """L2 Loss: Mean Squared Error"""
3      return dr.mean(dr.sqr(refImage - image))
```

However, as shown in Figure 10, the user is provided with a dropdown menu where the following optimization functions can be chosen: mean squared error, brightness independent mean squared error, dual buffer method [10], mean absolute error, and mean bias error. In the following, we show the dual buffer method implementation (the other optimization function can be found in the *MaterialOptimizerModel*):

```
    def dualBufferError(self, image, image2, refImage):
        """
        Loss Function mentioned in: Reconstructing Translucent Objects Using
     Differentiable Rendering, Deng et al.
        ...
        """
        return dr.mean((image - refImage) * (image2 - refImage))
```

Whenever the user selects one of the available optimization functions from the dropdown menu, the *onLossFunction()* method is called, which saves the loss function in the Model for the optimization. We omit the details; the interested reader may find the details in the implementation.

4.2.6 *Step 6: Choosing hyperparameters*

We omit the details of the implementation. However, the reader may think that the implementation has many similarities to the *loadReferenceImages()* method, in the sense that whenever an action in the View occurs, then the Controller is triggered, thus Model is updated with the appropriate values. The interested reader may find the implementation details in the *onMinErrLineChanged(), onIterationCountLineChanged(), onSamplesPerPixelChanged()* methods.

4.2.7 *Step 7: The optimization loop*

As mentioned previously, whenever the user clicks the 'Start Optimization' button, the following method is called by the Controller:

```
def optimizeMaterials(self):
    # Precondition: (1) reference image/s is/are loaded, and
    #               (2) at least one checked scene parameter
    # The implementation is as in the previous sections
    ...
    lossHist, sceneParamsHist = self.model.optimizationLoop(
        opts, lambda x: self.view.progressBar.setValue(x)
    )
    # Omitting View dependent details...
```

If the preconditions are met, the Controller instructs the Model to prepare for optimization. Next, the Model is called for the optimization loop, which the implementation is shown here:

```
1  def optimizationLoop(self, opts: list, setProgressValue: callable = None):
2      lossHist = []; sceneParamsHist = []
3      for it in range(self.iterationCount):
4          if setProgressValue is not None:
5              setProgressValue(int(it / self.iterationCount * 100))
6          total_loss = 0.0
7          for sensorIdx, sensor in enumerate(self.scene.sensors()):
8              loss = self.computeLoss(sensor=sensor, seed=it)
9              # Backpropagate through the rendering process
10             dr.backward(loss)
11             self.updateAfterStep(opts, self.sceneParams)
12             total_loss += loss[0]
13
14         # update loss and scene parameter histograms
15         sceneParamsHist.append(
16             self.createSubsetSceneParams(
17                 self.sceneParams,
18                 SUPPORTED_MITSUBA_PARAMETER_PATTERNS,
19             )
20         )
21         lossHist.append(total_loss)
22         if total_loss < self.minError:
23             break
24     return lossHist, sceneParamsHist
```

The *optimizationLoop()* method receives two input, the *opts* refers to a list of Adam optimizers that we initialized previously and a function *setProgressValue* that indirectly updates the optimization progress.

The output of this function is two lists *lossHist* contains the loss during optimization, and *sceneParamsHist* contains scene parameters at each iteration step - they are initialized to empty lists at the beginning of the loop (*line 2*).

The optimization loop begins with the selected amount of iterations counts (*line 3*). Then the progress bar is updated with the *setProgressValue* (*line 5*).

Subsequently, the following loop begins for each available camera position (*line 7*). For each defined camera pose, the Model computes the loss with the chosen loss function (*line 8*), where the optimization function receives two inputs, firstly the noisy differentiable rendering of the scene and, secondly, the reference image. Next, Dr.Jit backpropogates through the rendering process with the computed loss (*line 10*). Note that Dr.Jit backward propagates gradients from the provided differentiable Dr.Jit Array (i.e., the computed loss) [4]. Next, the optimizer takes a gradient descent step and we update the scene state to the new optimized values (*line 11*). Additionally, during these steps, we keep track of the loss (*line 12*).

After completing the loop for each camera pose, we update the loss and scene parameter histograms (*line 14, 21*). Later, we stop the optimization if the total loss is lower than the previously defined minimum error (*line 22*). The optimization repeats these steps until either minimum error is reached or the amount of iteration equals the previously defined iteration count. In the end, the method returns the two lists: *lossHist*, and *sceneParamsHist* (*line 24*).

[4] Reminder: The differentiable render and backpropagation steps are handled by Mitsuba and Dr.Jit. [17, 16]

5 Evaluation

In this section, we are going to examine the results that we gathered from the *material-optimizer* [5]. This section will be divided into two parts. The first part will contain the outcomes of using synthetic data. Furthermore, in the second part, we are going to discuss the results that originate from real-world data. In both sections, we will try to focus on translucent material reconstruction. However, we will also show some results from opaque materials.

Testing material reconstruction with synthetic data is much more straightforward than real-world data since a scene description is already given or at least could be constructed digitally without taking into account specific real-world details. But, more importantly, it often does not require further modeling. In the case of real-world data, scene reconstruction from the real world is substantially more complicated. It requires a controlled environment where all the scene parameters that might affect the resulting image - such as light position and object geometry - are known or at least approximated accurately.

5.1 Synthetic Data

In this section, we examine material reconstruction from synthetic data. More specifically, the reference image is rendered after a mitsuba scene that is constructed digitally. The reference mitsuba scene is manipulated such that the render of the scene results in the initial image as in Figure 9.

In Figure 14, two plots of the material reconstruction of a bunny are shown. As in the previous cases, the top left corner of each figure shows the parameter error plot (i.e. loss during optimization), where the x-axis displays the iteration count, and the y-axis shows the loss during optimization. The top right corner shows the initial state, the bottom left the optimized image, and lastly, the bottom right corner shows the reference image.

The bunny object that is shown in Figure 14 is assigned to a Principled BSDF [3][6]. The only difference between the left and right plots is the loss function used during optimization. As seen on the left, the Dual Buffer Method by Deng et al. delivers results with a lesser error rate [10]. Please note that the optimization result with the minimum error rate is shown in both optimizations. The details of the material optimization procedure - such as the optimized scene parameters - can be found in Table 2. The difference in samples per pixel during iteration is mainly how the Dual Buffer Method by Deng et al. 2022 is introduced. For further details, please refer to the corresponding paper.

Another material reconstruction example is shown in Figure 15. However, in this example, the material optimization procedure is applied to another shape that blends two BSDFs that are defined in mitsuba, namely Bump map BSDF adapter and Principled BSDF [1]. Additionally, the base color value of the object

[5] Please note that unless specified otherwise we are going to use the default hyperparameters and optimization parameters.

[6] The Principled BSDF on Mitsuba is based on [8, 15]

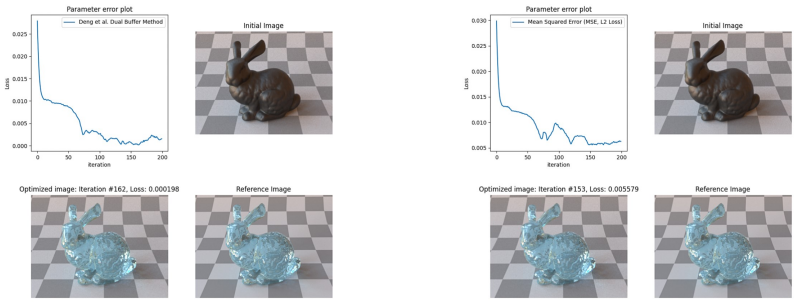

Figure 14: Material optimization results of a translucent bunny object with Principled BSDF. Left: The optimization results with the Dual Buffer Method [10]. Right: The optimization results with the with mean squared error (L2 Error).

Parameters	Reference	Initial	Result (left)	Result (right)
`roughness`	0.01	0.5	0.017	0.01
`base_color`	$0.41, 0.82, 0.99$	$0.1, 0.1, 0.1$	$0.41, 0.83, 0.99$	$0.41, 0.81, 0.99$
`spec_trans`	0.9	0.02	0.90	0.91
`eta`	1.49	1.54	1.49	1.48
`Loss function`	-	-	Dual Buffer Method	Mean Squared Error
`Samples per pixel`	-	-	8	16
`Iteration (min. error)`	-	-	162	153
`Loss`	-	-	0.000198	0.005579

Table 2: Results from Figure 14

is defined as a bitmap texture to show the capability of reconstructing materials of translucent objects with varying color values using Principled BSDF.

Table 3 shows the corresponding results. Please note that for conciseness, Figure 15 and Table 3 show only the results from the Dual Buffer Method by Deng et al. 2022.

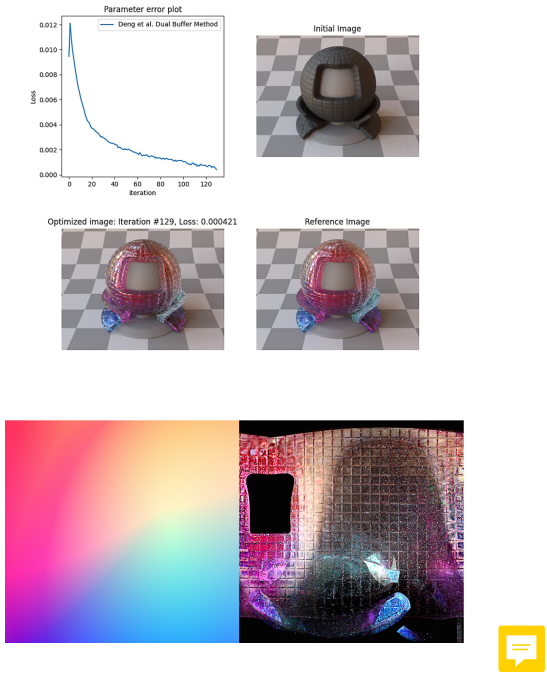

Figure 15: Top: Material optimization result of a translucent object with blended BSDF (Bump map and Principled) Left Bottom: Reference bitmap texture [12]. Right: Optimized bitmap texture.

Parameters	Reference	Initial	Result
`roughness`	0.001	0.7	0.01
`base_color`	Texture	Empty (Black) Bitmap Texture	Reconstructed Bitmap Texture
`spec_trans`	1.0	0.1	0.99
`eta`	1.49	1.64	1.50
`Samples per pixel`	-	-	8
`Iteration (min. error)`	-	-	129
`Loss`	-	-	0.000421

Table 3: Results from Figure 15

Similar to the previous examples, the material reconstruction of a bunny object is shown in Figure 16. However, in the following examples the bunny object is assigned to a Roughdielectric BSDF with a volume (participating media: medium) inside [1]. Consequently, these examples use a differentiable volumetric path tracer (*prbvolpath*) and reconstruct volumetric material properties [5, 2]. Thus, the *material-optimizer* provides two *possible* ways to reconstruct translucent objects: one via reconstructing Principled BSDF properties and the other with (Rough/Thin)Dielectric BSDF, including a participating media interior.

Note that a constant RGB value is being optimized in Figure 16 to represent the albedo value of the object. The results are shown in Table 4.

Parameters	Reference	Initial	Result
`alpha`	0.01	0.98	0.001
`albedo`	$0.412, 0.824, 0.999$	$0.01, 0.01, 0.01$	$0.53, 0.87, 0.99$
`sigma_t`	0.4	0.99	0.63
`eta`	1.49	1.55	1.49
`Learning rate: eta`	-	-	0.003
`Max. clamp value: eta`	-	-	1.55
`Samples per pixel`	-	-	8
`Iteration (min. error)`	-	-	113
`Loss`	-	-	0.001205

Table 4: Results from Figure 16

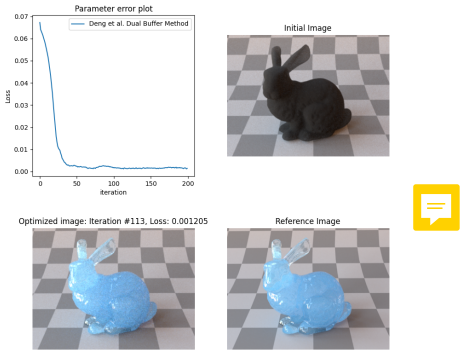

Figure 16: Material optimization result of a translucent bunny with Roughdielectric BSDF and homogeneous volume.

Another example is shown in Figure 17. However, this time the material of the bunny object is represented with Roughdielectric BSDF, and homogeneous *varying* volume albedo. This is similar to the difference between the Figure 14 and 15, where in Figure 14 a constant RGB value and in Figure 15, a bitmap texture represented the object's color value. The results are shown in Table 5.

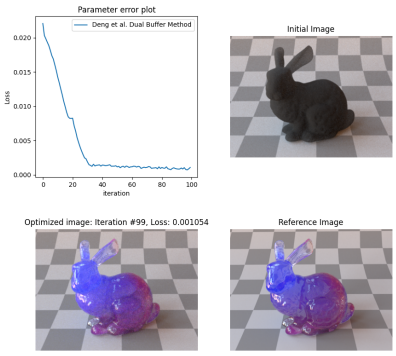

Figure 17: Material optimization result of a translucent bunny with Roughdielectric BSDF, homogeneous varying volume.

Parameters	Reference	Initial	Result
`alpha`	0.01	0.98	0.001
`albedo`	Volume (16x16x16x3)	Empty Volume	Reconstructed Volume
`sigma_t`	0.4	0.99	0.64
`eta`	1.49	1.55	1.48
`Learning rate: eta`	-	-	0.01
`Max. clamp value: eta`	-	-	1.55
`Samples per pixel`	-	-	4
`Iteration (min. error)`	-	-	100
`Loss`	-	-	0.001054

Table 5: Results from Figure 17

5.2 Real-World Data

In this section, we examine material reconstruction from real-world data. In Section 4.1, we proposed a workflow for real-world data acquisition. However, the **translucent material reconstruction** results we will present in this section do not follow the previously mentioned workflow. Instead, we approximated the real-world environment in Blender and then exported it using Mitsuba Blender Add-On for further use [7].

Figure 18 shows the material reconstruction of a translucent algae material. In the initial state, the object is assigned to the Principled BSDF. As in the

[7] More specifically, Prof. Peter Ferscin (TU Vienna) and his former masters' student Cheng Shi provided us with a material scanner, where we had the chance to take sample reference images. Consequently, we also used the 3D model - that is developed by Cheng Shi - of the material scanner in Blender to approximate the sample reference images that we took.

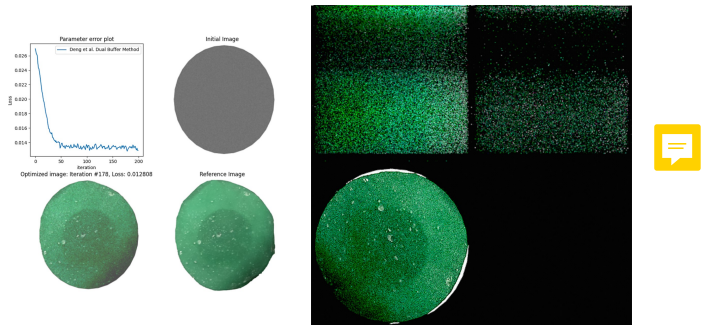

Figure 18: Left: Material optimization result of a translucent green algae object with Principled BSDF. Right: Optimized bitmap texture.

previous cases, the plots we provide have no changes. The top left shows the loss during optimization, the top right the initial state, the bottom left the optimized scene state, and the top right the reference image. The results from Figure 18 are shown in Table 6.

Parameters	Initial	Result
`roughness`	0.5	0.69
`base_color`	Empty (Black) Bitmap Texture	Reconstructed Bitmap Texture
`spec_trans`	0.0	0.51
`Min/Max clamp value: roughness`	-	$(0.3, 0.7)$
`Min/Max clamp value: spec_trans`	-	$(0.4, 0.8)$
`Iteration (min. error)`	-	178
`Loss`	-	0.012808

Table 6: Results from Figure 18

Another material reconstruction example is shown in Figure 19. Table 7 shows the corresponding results. The interesting thing to note in this example is the selected *'spec_trans'* scene parameter [3], which after 300 iterations, remained at 0.6, the same value as the defined minimum clamp value.

Parameters	Initial	Result
`roughness`	0.5	0.31
`base_color`	Empty (Black) Bitmap Texture	Reconstructed Bitmap Texture
`spec_trans`	0.0	0.6
`Max. clamp value: roughness`	-	0.5
`Min. clamp value: spec_trans`	-	0.6
`Iteration (min. error)`	-	299
`Loss`	-	0.006886

Table 7: Results from Figure 19

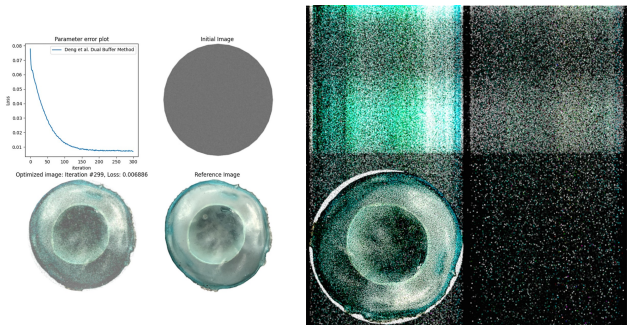

Figure 19: Left: Material optimization result of a translucent blue algae object with Principled BSDF. Right: Optimized bitmap texture.

Similar to the example in Section 3.2.1, we show another material reconstruction illustration in Figure 20. As in example in Section 3.2.1, multiple reference images are used to reconstruct this model. Moreover, the example follow the workflow described in Section 4.1. Although there are many similarities to the example shown in Section 3.2.1, this time, rather than optimizing a bitmap texture, we optimize the mesh attribute texture (precisely vertex colors) of the model. The results are shown in Table 8.

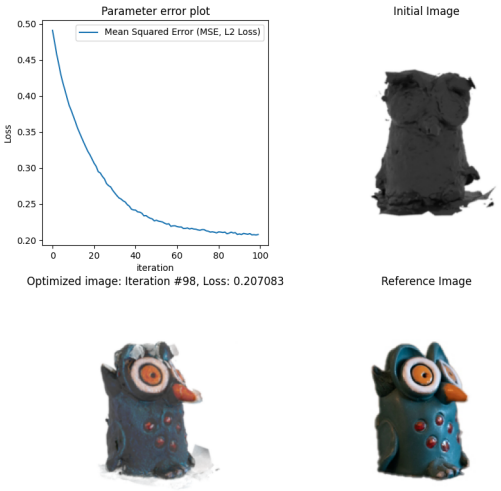

Figure 20: Material optimization result of mini bird statue object with Principled BSDF.

Parameters	Initial	Result
`roughness`	0.5	0.56
`vertex_color`	Black Vertex Colors	Reconstructed Vertex Colors
`specular`	0.5	0.8
`Min/Max clamp value: roughness`	-	$(0.3, 0.8)$
`Min/Max clamp value: specular`	-	$(0.3, 0.8)$
`Iteration (min. error)`	-	100
`Samples per pixel`	-	2
`Loss`	-	0.207083

Table 8: Results from Figure 20

6 Conlusion and Future Work

In this project, we explored material reconstruction using the mitsuba (differentiable) renderer. For the task of inverse rendering, we proposed a mini-tool: the material optimizer.

Material-optimizer is a tool capable of reconstructing material properties by loading a scene description file and multiple images. The proposed tool automatically detects the most suitable integrator and offers a list of differentiable scene parameters for optimization. In its ~~most~~ simplest form, *material-optimizer* lets the user start the optimization with three to four clicks. In case of unsuccessful optimization attempts, the *material-optimizer* lets the user restart the optimization without any code modifications. In satisfactory attempts, the user of the tool is able to output the optimized scene parameters in supported file formats.

Moreover, we also outlined a well-known procedure in Section 3.1 and integrated it into the *material-optimizer*. In essence, the material optimizer implements a gradient-based optimization algorithm that each step improves the differentiable objective function to acquire the suitable material properties from a given reference image.

To increase the reproducibility of the shown examples and help the future use of this tool, we also proposed a simple workflow to acquire the necessary scene and reference image files. We also implied that the *material-optimizer* is capable of geometry reconstruction and proposed a naive approach to combine geometry and material reconstruction. However, we also indicated that geometry reconstruction might not be achievable that naively and mentioned that the procedure might require additional constraints.

The main focus of this project relied on translucent material reconstruction using synthetic and real-world data. We proposed two ways to reconstruct translucent materials: first, with the use of Principled BSDF, and second, with Rough/Thindielectric with a homogeneous volume interior. Nonetheless, we likewise discovered that opaque material reconstruction is also possible and, in most cases, easier, such as diffuse materials.

One limitation we observe in our work is the requirement of the CUDA API (i.e., NVIDIA GPUs) for the *material-optimizer*. However, since mitsuba 3 supports LLVM variants, we hope to integrate the capability of LLVM (i.e., the use of multiple CPUs) in the *material-optimizer* in the near future.

Another limitation we examine in our work is the mitsuba scene file specificity. One possible and relatively easy solution to discard this constraint in the future would be to allow importing standardized scene description formats such as X3D and Universal Scene Description.

Regarding translucent material reconstruction, the main limitation we observe in our work is the complexity of the scene description file and reference image acquisition. More specifically, as we discussed previously, the nature of real-world data acquisitions can be extremely complex since the measurement process contains many parameters, such as the requirement of a controlled environment and the necessity of geometry reconstruction prior to material reconstruction. In the future, we hope to explore these nitty-gritty details for more scientific and exact measurements.

A philosophic limitation we see in our work is the nature of a renderer. At the end of the day, a differentiable renderer would be only capable of what the original renderer is capable of. More specifically, an inverse renderer can only reconstruct something from the real world if it can already simulate them through the steps of modeling and shading. For example, if the renderer only supports diffuse materials, then the differentiable renderer would only be capable of reconstructing objects with diffuse material. Although this might not sound trivial, real-world examples may include complex scenarios that current rendering engines are incapable of. Nonetheless, with the use of scientific methods, we believe that the computer graphics community and the industry will continue to research and find explanations for the questions to which we are currently unable to find solutions.

References

[1] 3, M. Mitsuba 3: Bsdfs. https://mitsuba.readthedocs.io/en/stable/src/generated/plugins_bsdfs.html#, 2023. [Online; accessed 13-January-2023].

[2] 3, M. Mitsuba 3: Integrators. https://mitsuba.readthedocs.io/en/stable/src/generated/plugins_integrators.html, 2023. [Online; accessed 11-January-2023].

[3] 3, M. Mitsuba 3: Principled bsdf. https://mitsuba.readthedocs.io/en/stable/src/generated/plugins_bsdfs.html#the-principled-bsdf-principled, 2023. [Online; accessed 12-January-2023].

[4] 3, M. Mitsuba 3: Scene xml file format. https://mitsuba.readthedocs.io/en/stable/src/key_topics/scene_format.html, 2023. [Online; accessed 06-January-2023].

[5] 3, M. Mitsuba 3: Volumes. https://mitsuba.readthedocs.io/en/stable/src/generated/plugins_volumes.html#, 2023. [Online; accessed 06-January-2023].

[6] Adam Celarek. This diagram illustrates the rendering equation (recursive formulation). https://www.cg.tuwien.ac.at/sites/default/files/course/4854/attachments/05_the_rendering_equation.pdf, 2022. [Online; accessed December 22, 2022. Description:This diagram illustrates the rendering equation (recursive formulation). Source: VU Rendering 2022, TU Vienna. Author: Adam Celarek].

[7] <baptiste.nicolet@epfl.ch>, B. N. Mitsuba blender add-on. https://github.com/mitsuba-renderer/mitsuba-blender, 2022.

[8] Burley, B. Physically-based shading at disney.

[9] Community, B. O. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[10] Deng, X., Luan, F., Walter, B., Bala, K., and Marschner, S. Reconstructing translucent objects using differentiable rendering. In *ACM SIGGRAPH 2022 Conference Proceedings* (New York, NY, USA, 2022), SIGGRAPH '22, Association for Computing Machinery.

[11] Gkioulekas, I., Zhao, S., Bala, K., Zickler, T., and Levin, A. Inverse volume rendering with material dictionaries. *ACM Trans. Graph. 32*, 6 (nov 2013).

[12] Gradienta. Unsplash: Gradienta. https://unsplash.com/backgrounds/colors/gradient?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText, 2022. [Online; accessed January 13, 2023.].

[13] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature 585*, 7825 (Sept. 2020), 357–362.

[14] Henrik. This diagram illustrates the ray tracing algorithm for rendering an image. `https://upload.wikimedia.org/wikipedia/commons/8/83/Ray_trace_diagram.svg`, 2008. [Online; accessed December 21, 2022. Description: This diagram illustrates the ray tracing algorithm for rendering an image.Date: 12 April 2008.

[15] Hill, S., McAuley, S., Burley, B., Chan, D., Fascione, L., Iwanicki, M., Hoffman, N., Jakob, W., Neubelt, D., Pesce, A., and Pettineo, M. Physically based shading in theory and practice. In *ACM SIGGRAPH 2015 Courses* (New York, NY, USA, 2015), SIGGRAPH '15, Association for Computing Machinery.

[16] Jakob, W., Speierer, S., Roussel, N., Nimier-David, M., Vicini, D., Zeltner, T., Nicolet, B., Crespo, M., Leroy, V., and Zhang, Z. Mitsuba 3 renderer, 2022. https://mitsuba-renderer.org.

[17] Jakob, W., Speierer, S., Roussel, N., and Vicini, D. Dr.jit: A just-in-time compiler for differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH) 41*, 4 (July 2022).

[18] Jarosz, W. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.

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

[20] Karnewar, A., Ritschel, T., Wang, O., and Mitra, N. Relu fields: The little non-linearity that could. In *ACM SIGGRAPH 2022 Conference Proceedings* (New York, NY, USA, 2022), SIGGRAPH '22, Association for Computing Machinery.

[21] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization, 2014.

[22] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (aug 1987), 163–169.

[23] Matthew Tancik*, Ethan Weber*, E. N. Nerfstudio: A framework for neural radiance field development, 2022.

[24] Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM 65*, 1 (dec 2021), 99–106.

[25] Müller, T., Evans, A., Schied, C., and Keller, A. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph. 41*, 4 (jul 2022).

[26] Munkberg, J., Hasselgren, J., Shen, T., Gao, J., Chen, W., Evans, A., Mueller, T., and Fidler, S. Extracting Triangular 3D Models, Materials, and Lighting From Images. *arXiv:2111.12503* (2021).

[27] Nimier-David, M., Müller, T., Keller, A., and Jakob, W. Unbiased inverse volume rendering with differential trackers. *ACM Trans. Graph. 41*, 4 (jul 2022).

[28] Nimier-David, M., Speierer, S., Ruiz, B., and Jakob, W. Radiative backpropagation: An adjoint method for lightning-fast differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH) 39*, 4 (July 2020).

[29] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation (3rd ed.)*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Oct. 2016.

[30] the docs, M. . R. Procedure of differentiable rendering. `https://mitsuba.readthedocs.io/en/stable/_images/autodiff_figure.jpg`, 2022. [Online; accessed December 22, 2022.].

[31] Vicini, D., Speierer, S., and Jakob, W. Path replay backpropagation: Differentiating light paths using constant memory and linear time. *Transactions on Graphics (Proceedings of SIGGRAPH) 40*, 4 (Aug. 2021), 108:1–108:14.

[32] Wig42. The grey & white room. `https://mitsuba.readthedocs.io/en/stable/_images/living-room.png`, 2014. [Online; accessed December 21, 2022. This is the Mitsuba version of 'The Grey & White Room' by Wig42, downloaded from https://benedikt-bitterli.me/resources/ The original file may be obtained here: http://www.blendswap.com/blends/view/75795 This scene was released under a CC-BY license. It may be copied, modified and used commercially without permission, as long as: Appropriate credit is given to the original author. A link to the license is provided. For more information about the license, please see https://creativecommons.org/licenses/by/3.0/].

[33] Wikipedia. Reynolds transport theorem — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Reynolds%20transport%20theorem&oldid=1111091325`, 2022. [Online; accessed 22-December-2022].

[34] Wikipedia. Cornell box — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cornell%20box&oldid=970092624, 2023. [Online; accessed 02-January-2023].

[35] Wikipedia. Metashape — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Metashape&oldid=1020580802, 2023. [Online; accessed 10-January-2023].

[36] Wikipedia. Model–view–controller — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=1122651234, 2023. [Online; accessed 06-January-2023].

[37] Wikipedia. PyQt — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=PyQt&oldid=1120752768, 2023. [Online; accessed 06-January-2023].

[38] Wikipedia. RGB color model — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=RGB%20color%20model&oldid=1128337273, 2023. [Online; accessed 06-January-2023].

[39] Yang, J., and Xiao, S. An inverse rendering approach for heterogeneous translucent materials. In *Proceedings of the 15th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry - Volume 1* (New York, NY, USA, 2016), VRCAI '16, Association for Computing Machinery, p. 79–88.

[40] Zeltner, T., Speierer, S., Georgiev, I., and Jakob, W. Monte carlo estimators for differential light transport. *ACM Trans. Graph. 40*, 4 (jul 2021).

[41] Zhang, C., Wu, L., Zheng, C., Gkioulekas, I., Ramamoorthi, R., and Zhao, S. A differential theory of radiative transfer. *ACM Trans. Graph. 38*, 6 (nov 2019).

[42] Zhao, S., Jakob, W., and Li, T.-M. Physics-based differentiable rendering: From theory to implementation. In *ACM SIGGRAPH 2020 Courses* (New York, NY, USA, 2020), SIGGRAPH '20, Association for Computing Machinery.