MASTER'S THESIS 2020

DiPTeR: a Differentiable Procedural Texture Renderer

Sebastian Hegardt

ISSN 1650-2884
LU-CS-EX: 2020-61

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-61

DiPTeR: a Differentiable Procedural Texture Renderer

Sebastian Hegardt

DiPTeR: a Differentiable Procedural Texture Renderer

Sebastian Hegardt
dat15she@student.lu.se

December 13, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Elin Anna Topp, elin_anna.topp@cs.lth.se
Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

Procedural texturing in offline rendering is steadily gaining popularity as a more compact, flexible and resolution independent alternative to conventional bitmap texturing and is supported by many major 3D software suites such as Blender and 3ds Max. However, the complexity of procedural textures can grow to unmanageable proportions, often requiring users to manually control hundreds of parameters in convoluted node graphs to achieve realistic results. In this thesis we investigate an iterative solution to automatically estimate an optimal parameter set for dynamically created procedural textures in order to minimize the difference between the rendered texture and a user's target texture. Our approach uses the gradient descent algorithm, requiring a differentiable rendering process which necessitated the development of our own specialized differentiable procedural texture renderer. Results show that our approach can perform well for our test target textures, yielding visually similar procedurally rendered textures, but the outcome relies heavily on the choice and tuning of the loss function.

Keywords: Inverse graphics, differentiable rendering, procedural texturing, parameter estimation

Acknowledgements

First of all, I would like to extend my sincere thanks to professor Takeo Igarashi at the University of Tokyo, who accepted the role of supervisor for me and my project. I would also like to thank Seung-Tak Noh for his good advice during the inception of this project in Japan.

I'm very grateful to have had my Swedish supervisor Elin Anna Topp, who has been my support through this entire journey, and who graciously accepted the role of sole supervisor when my collaboration with Japan ended prematurely due to COVID-19. Without you, I would still be working on my project plan. Furthermore, I thank Michael Doggett for accepting the role as my examiner.

Special thanks goes to Ana-Lice Machado and my dear mother Maria Marforio, for devoting their valuable time to proofread my report and finally to Hampus Åström for tips and tricks on how to improve my results.

Contents

Chapter 1
Introduction

Two-dimensional images mapped to three-dimensional objects, referred to as *textures*, have long been used to efficiently add pre-calculated details to a 3D scene in computer graphics. In the past decade however, procedural textures have gained an increasing amount of traction as a serious contender to the traditional bitmap texture. The procedural approach presents a new way of generating textures mathematically with a high level of editability, arbitrary levels of detail, compactness and seamless transition between object faces, see Figure 1.1. With programs such as Blender or Substances Designer growing in popularity and being adopted by industry professionals, tools for creating procedural texture models using node graph editors are now more accessible than ever [1, 2].

Despite its very appealing advantages, procedural textures remain an advanced alternative to traditional texturing methods in offline rendering for mainly one reason alone; they are difficult to design in order to achieve desired results. While modern tools present users with a node editor that abstracts away the underlying functions, figuring out how to compose nodes and what values to assign the sometimes hundreds of parameters in order to achieve desired results, is very convoluted and time-consuming. Recent advances in texture synthesis using neural networks, e.g. *TileGAN*, presents an exemplar based alternative that can generate higher quality versions of user example textures, although not solving the problems with seams and lack of compactness of traditional textures [3]. Still, texture synthesis is an example of an *inverse* modeling process that is easy to operate for a novel user, especially as a plethora of bitmap images are already available for free on the web which can serve as a target for such an inverse problem.

This thesis focuses on a problem within *inverse procedural texture modeling*, a research field that aims to inverse the rendering process and estimate parameters of procedural texture models such that the rendered result matches an input image. However, due to the complexity of most modern render engines built without this functionality in mind, the relationship between procedural model parameters is non-linear and near impossible to differentiate. Additionally, developing a suitable loss function that can measure image similarity is a non-trivial task, due to the discrepancy between the way computers and humans perceive

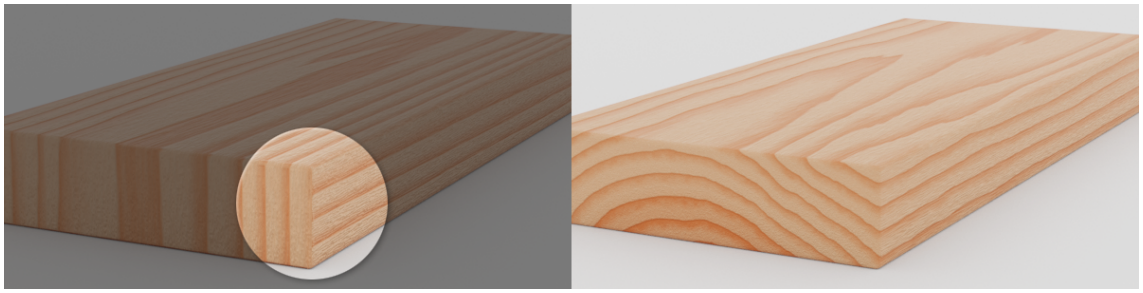


Figure 1.1: Left: Using a bitmap texture can result in visible seams between object faces as it is inherently difficult to naturally map a 2D texture onto a 3D object. Right: A similar texture rendered with a procedural model. As procedural textures are functions of an object's 3D coordinates, a seamless result is achieved [4].

images.

A few fully differentiable rendering engines have already been developed, such as *OpenDR* or *PyTorch3D*, but neither have direct support for procedural textures [5, 6]. Recently Guo et. al published a paper detailing a framework for procedural parameter estimation using Bayesian inference [7]. Their results are promising and the contribution of suitable loss functions serve as an inspiration for this thesis. However, their framework did not allow the user to compose their own procedural textures and only tested their algorithm on hard-coded procedural models. Intending to fill a gap in the field of inverse procedural texture modeling, we present DiPTeR, a Differentiable Procedural Texture Renderer that allows users to dynamically build fully differentiable procedural textures and automatically estimate its parameters using a target image.

This project was initially conducted in Japan at the University of Tokyo as part of the HCI for machine learning focused group *CREST*, led by Takeo Igarashi [8]. In April 2020 the threat of the COVID-19 crisis forced the university to close, and we had to continue the project in Lund, Sweden unfortunately without access to Lund University's premises or equipment.

1.1 Goal

To alleviate the difficulties of creating complex procedural texture models by hand, our goal is to develop a *Differentiable Procedural Texture Renderer* or DiPTeR, which can render procedural textures in OpenGL but also include a fully differentiable back-end renderer that mimics the rendering process of OpenGL. The back-end renderer should be written in a framework that allows for automatic differentiation, enabling us to create procedural functions and rendering procedures of arbitrary complexity while retaining differentiability. The framework should include a node editor, much like Blender and Substance Designer, that lets users easily build procedural texture models and observe their rendered results in real time. A user should also be able to present a target bitmap texture, and let the system automatically estimate the optimal parameter values of their procedural texture model. Automatically constructing the model itself is outside the scope of this project, and is an entirely different problem. Measuring the similarity between rendered texture and target texture is accomplished by means of

loss functions, and the framework should allow the user to easily select and implement their own loss functions. While the framework is intended to be operated through a graphical interface developed by us, it should be possible to run the entire texture matching algorithm through a non-graphical user interface.

Through evaluation of our system, we would like to answer the following research questions: Is it possible to build a differentiable rendering system in PyTorch? Is it efficient enough to render in real time? Is it possible to mimic the intricacies of procedural texturing of OpenGL in PyTorch, including the use of noise? Can we achieve an acceptable parameter estimation for a complex procedural texture on a timescale of minutes? Can we find a loss function that performs well for many different types of procedural textures?

1.2 Methodology

This project was initialized by conducting a general literature review where we researched current and past methods of inverse rendering, specifically pertaining to procedural textures. Most papers pursued a neural network based solution which inspired our initial research plan to train a GAN on a dataset of procedurally rendered textures in order to perform regression on the procedural model parameters.

During the first seven weeks in Tokyo we developed an add-on for Blender to automatically render datasets to train on, as well as conducting some testing of our initial solution. Eventually we realized that this approach does not perform well on complex procedural textures and is not flexible enough. We then entered our second literature study phase and designed a new solution relying on explicit inverse rendering by using a differentiable renderer.

Our main development phase was then started and our solution implemented in Python, which was chosen for its excellent support for machine learning as well as fast and simple development. The implementation consists of three main parts, the differentiable renderer including shaders and parameter estimation implemented using the machine learning library PyTorch, the non-differentiable OpenGL rendering integrated in Python by the package Glumpy and lastly a graphical user interface implemented using PyQt5 [9, 10, 11].

Finally, our system was evaluated by designing three test shader models of increasing complexity and running parameter estimation on them for all combinations of optimizers and loss functions that we support, measuring the loss value at each iteration. Furthermore, rendering performance of our back-end render engine was evaluated as well as the time required to complete an iteration of gradient descent for each loss function.

1.3 Contributions

In this thesis we outline a novel approach to the under-researched problem of inverse procedural texture modeling by applying reliable techniques such as stochastic gradient descent. Specifically, our principal contributions are:

- Implementing a fully differentiable, dynamic procedural texture renderer. Procedural textures can be dynamically created during run-time and rendered in real-time while maintaining differentiability.

- Pairing this differentiable renderer with OpenGL, a popular and modern computer graphics API, in such a way that users do not have to be aware that two systems are used in tandem. This allows for fast and familiar forward rendering in OpenGL as well as parameter estimation using our differentiable renderer, therefore enabling both rendering system to be exported to other projects separately.
- Using proven techniques for optimization such as stochastic gradient descent for procedural texture parameter estimation. To the best of our knowledge this has not been done before.
- Developing a graphical user interface, including a node editor, enabling users to build composite procedural texture models.
- Developing a GLSL code parser framework to enable GLSL shader composition and compilation at run-time as well as code reuse through a custom preprocessor directive.
- Evaluating a number of different loss functions and optimizers for parameter estimation.

1.4 Related Work

This thesis primarily covers two topics; Inverse Rendering and Procedural Textures. Neither subject is new and both are well researched, but few papers attempt to combine the two. Many approaches to inverse rendering use neural networks, with good results for their somewhat narrow use cases. This thesis was born from inspiration of this method, but it was soon discovered that it is not a viable solution for our use case, where we allow users to freely modify and compose procedural textures, resulting in near endless pattern combinations. Training a neural network to perform well on such data would require an immense dataset to train on, as well as an extremely complex model. Instead, we chose a differentiable rendering approach using gradient descent to estimate procedural texture parameters, which to the best of our knowledge, has not been done before. This section analyzes a number of papers and frameworks whose solutions have directly affected this thesis.

1.4.1 Inverse Procedural Texture Modeling

A relatively low amount of research has been conducted within *inverse procedural texture modeling*, a narrow subfield of computer science, which has nonetheless produced two recent papers that lay the foundation for this thesis. In 2019, Hu et. al. published *A Novel Framework For Inverse Procedural Texture Modeling* in which they describe a framework for procedural texture model acquisition as well as parameter estimation for said model [12]. A K-means algorithm is used to find the most suitable procedural texture model among a library of predefined models for a user's input texture. Each procedural texture has an associated Convolutional Neural Network model, used to solve the regression problem of finding an optimal parameter set to the procedural texture so that it renders an image similar to the user's input image. This approach is unfortunately not very flexible, as the procedural textures have to be predefined and training a neural network on each texture requires a considerable amount of effort

and time. Using CNNs for regression also proved difficult for complex procedural texture models, and so neural networks were abandoned for this thesis altogether.

A much different approach was formulated in the work by Guo et. al. in 2019, where their solution involves Bayesian inference and sampling of the space of plausible parameters to find an optimal parameter set for a chosen procedural texture [7]. Similar to their work, we also implemented a solution in a differentiable framework, however, our system does not require any sampling techniques. Furthermore, they contribute by developing smart loss functions that do not depend on textures being pixel-wise aligned, some of which have been implemented in this thesis. Finally, unlike us they do not provide an interface to create procedural textures, and only offer hard-coded procedural textures.

1.4.2 Differentiable Rendering

An important foundation that allows our approach to work is the concept of differentiable rendering, that is, the ability to differentiate the entire rendering process, and thus obtain gradients for it. A few notable examples of this have been an inspiration when we developed our own differentiable renderer. Perhaps most notable is the very recently released PyTorch3D, a rendering framework developed by the team behind *PyTorch* which was in fact used to implement our differentiable renderer [6, 9]. PyTorch3D was unfortunately released without our knowledge around the same time as this thesis was started and shares many concepts, although their approach is more general and lack any advanced procedural texturing support. Earlier approaches like the OpenDR framework is closely related to our project, but with a different focus [5]. OpenDR is a framework where the entire forward rendering process is differentiable, enabling the user to automatically compute scene parameters such as vertex positions, camera parameters and vertex colors. Our solution does not focus on the 3D model or camera parameters, but on per-pixel procedural colors and not only per-vertex colors. As much of our algorithm relies on writing differentiable shaders, we must not fail to mention an attempt to do just that, directly implemented in the HLSL shader language [13]. This project was led by Microsoft Research and although their use case was more aimed towards calculating efficient normals, tangents and derivatives of sub-routines, it remains an interesting idea that would have been too complex to implement in this thesis.

1.5 Report Outline

The next chapter explains some of the theoretical background needed to understand this thesis. At the end of the chapter in section 2.6, some other similar forms of inverse graphics and related works are explained. Next, chapter 3 outlines the method used in this project, explaining the different parts of DiPTeR, including the graphical interface, as well as putting the theory into context. Chapter 4 then explains some of the implementation details on how our PyTorch shaders mimic OpenGL shaders and some details on the OpenGL rendering pipeline. In chapter 5 we explain the test setup and evaluation of our parameter estimation method on a number of test shaders and discuss the results. Finally, we draw a brief conclusion on the rendering and parameter estimation capabilities of DiPTeR, as well as suggest what could be done in the future to further improve our method.

Chapter 2
Background

This chapter explains concepts and mathematical theory used in this thesis. First, some mathematical background on procedural texturing in general is presented, followed by the basics of our central algorithm gradient descent, used for parameter estimation, in sections 2.2 and 2.3. In section 2.4 the concepts behind automatic differentiation that enables us to develop a differentiable renderer are described. Finally, some related work from different sub-fields of inverse rendering are presented and discussed, illustrating why their solutions were not used in this project.

2.1 Procedural Textures

Texturing is a method of adding more detail and realism to a 3D object, traditionally by projecting a 2D bitmap texture onto its surface. This is a rather straightforward process, but presents a number of practical problems that are difficult to overcome. Firstly, a bitmap texture has a fixed amount of detail; a pixel resolution. The resolution of an image can normally not be scaled up, although new research has made progress in that aspect much thanks to the advances in neural networks [14, 15]. However, even if the resolution is sufficient, bitmap textures are still inflexible as it is difficult to modify the underlying patterns without negatively affecting its overall appearance. Furthermore, bitmap textures require a considerable amount of storage space, especially at high resolutions. Another hurdle that is often overlooked, is the inherent problem of smoothly projecting 2D texture coordinates $p_{2D} = (u, v)$ onto points on a 3D object's surface $p_{3D} = (x, y, z)$. This mapping process is a function from two-dimensional coordinates in a texture matrix to three-dimensional points on an object's surface and is often referred to as *UV mapping*, see equation 2.1, and can give rise to visible seams on the edges between object surfaces as demonstrated in Figure 1.1.

$$f(u, v) : \mathbb{N}^2 \mapsto \mathbb{R}^3 \tag{2.1}$$

Procedural textures, often referred to as *shaders* in computer graphics, are created using

mathematical functions that take the surface coordinates of the 3D object as input, as well as a set of n parameters $\theta \in \mathbb{R}^n$ describing its appearance, and returns a color for the point $p_{3D} = (x, y, z)$ see equation 2.2. The parameters θ can be adjusted by the user to change the underlying patterns and colors, making them much more flexible than the static bitmap approach. They are however slower than conventional bitmap textures as the mathematical functions have to be called thousands of times, once for each pixel, whereas for bitmap textures these values are pre-computed and read into memory. Procedural texturing is therefore mostly used in offline rendering as it introduces too much overhead in real-time rendering.

$$f(x, y, z, \theta) : \mathbb{R}^3, \mathbb{R}^n \mapsto \mathbb{R}^3 \tag{2.2}$$

2.1.1 Function Composition

As procedural textures are mathematical functions, their versatility can be extended using *function composition*, a basic mathematical operation taking two functions $f(x)$ and $g(x)$ as input, producing a new function $h(x) = f(g(x))$. While the final output of the procedural texture function should be the color of a point on a target object's surface, any output of any function $g(x)$ can be used as input to a procedural function $f(x)$. To understand how powerful function composition is when used with procedural textures, let $f(\vec{v}, \vec{w}, a)$ be a function of two vectors and a scalar mix factor that returns a weighted sum of both vectors, essentially mixing them. Let $g(x, y, z, a)$ be a function that returns an arbitrary sinusoidal pattern that depends on the input coordinates (x, y, z) as well as a scaling factor a and finally, let $r(x, y, z, \vec{\theta})$ be the main procedural texture function that is a composition of f and g. We also define a red color vector and a green color vector, $c_r = (1.0, 0.0, 0.0)$ and $c_g = (0.0, 0.3, 0.0)$ where the elements represent the red, green and blue channels. The functions are defined in equation 2.3 where all operators are applied element-wise.

$$\begin{aligned} f(\vec{v}, \vec{w}, a) &= \vec{v} \cdot (1 - a) + \vec{w} \cdot a \\ g(x, y, z, a) &= \sin(a \cdot x) \cdot \sin(a \cdot y) \cdot \sin(a \cdot z) \\ r(x, y, z, \vec{\theta}) &= f(c_r, c_g, g(x, y, z, \theta_0)) \end{aligned} \tag{2.3}$$

We can now texture a plane by coloring each pixel with coordinates (x, y, z), where $z = 1$ is constant, with the output of our procedural function r, and achieve the result in Figure 2.1, where parameter θ_0 is set to 20. Essentially, this procedural strawberry pattern is achieved by mixing the red and green color in function f using the return value of the sinusoidal function g as a factor. Note that the coordinates $0.0 \leq x, y, z \leq 1.0$ are normalized, meaning the pattern is independent of the size of the rendered plane. Refer to appendix A.1 for an online version.

2.1.2 Procedural Noise

The real world is full of seemingly unpredictable patterns from the way trees branch out to the uneven surface of a rock. To mimic these patterns in procedural texturing, a type of controlled randomness called noise is extensively used. Noise is deterministic, which means that for the same parameters, a noise function will always generate the same value. Noise also has the property that it is smooth, so that local changes are small and gradual but on a global

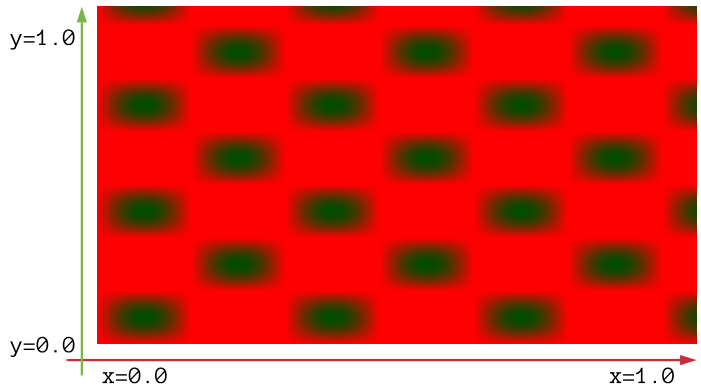


Figure 2.1: Simple procedural strawberry pattern created from composing two functions. The axes show the normalized plane coordinates.

scale, it looks random. The most famous type of noise is *Perlin noise*, an award winning noise algorithm developed by Ken Perlin in order to create more realistic textures for the movie Tron (1982), formerly described in his paper in 1985 [16]. The way this and many other noise algorithms generate a pseudorandom, deterministic number from a coordinate (x, y) is by taking the fractional part of a sinusoid that is multiplied by any large number, like shown in equation 2.4. This results in a well behaved function that returns a number between 0 and 1 that is perceived as random for a relatively large change in coordinate (x, y), but is deterministic.

$$rand(x, y) = \text{fract}(\sin(123.23x + 923.1y) \cdot 465123.97) \tag{2.4}$$

To create smooth 2D noise, Perlin had the idea of smoothly interpolating between four corners of a square, with a pseudorandom value assigned to each corner. If we texture a plane with normalized coordinates using this technique, it will mostly look like a gradient. However, if we simply scale up the coordinates, effectively scaling down the pattern, it will be pseudorandomly repeated across the plane. The pattern will still look rather blocky, and to create an appealing noise pattern, we can utilize what is called *fractal brownian motion*, where any number of octaves of this sinusoidal noise is added with diminishing scale and amplitude. The progress towards a deterministic cloud pattern is shown in Figure 2.2.

2.2 Gradient

A concept commonly used throughout this thesis is the *gradient* of a function, which is a generalization of the ordinary derivative. Indeed, the gradient of a function $f(x)$ of a single variable is equal to the derivative of f with respect to x, denoted $\frac{df}{dx}$. For a multivariate function $f(x_1, x_2, \ldots, x_n)$ however, the gradient of f, denoted ∇f, is defined as the vector of partial derivatives of f with respect to each argument, as shown in equation 2.5.

$$\nabla f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \ldots & \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{2.5}$$

The derivative famously describes a function's rate of change, or direction of greatest

Figure 2.2: Left: Perlin noise interpolated between four corners of a plane. Middle: The normalized coordinates of this plane can be scaled up, in this example five times, to add more detail. Right: Finally, to achieve a cloud pattern, octaves of the noise is summed using different scale values and amplitudes.

increase. Similarly, for a multivariate function f of n variables, the gradient is an n dimensional vector in the direction of greatest increase, also called the direction of *steepest ascent*. This stems from the common depiction of multivariate functions as surface plots with the gradient pointing in the direction where the surface has the steepest upward slope. In minimization problems, the gradient can be used to find the direction of *steepest descent*, which is simply defined as the negative of the gradient [17].

2.3 Stochastic Gradient Descent

The gradient descent algorithm is one of the cornerstones of this project and is in fact a fairly simple algorithm for iteratively optimizing a, most often, multivariate function. It is a popular algorithm in machine learning where it is used to minimize the loss of a neural network model as measured by a loss function. It works by iteratively finding the direction of steepest descent of the loss function defined as the negative of a function's gradient, as explained in section 2.2, eventually leading to a minimal loss value. Finding the gradient of a complex function can be done automatically and reliably if defined in an automatic differentiation framework as explained in section 2.4.

There are a few major versions of the gradient descent algorithm, the most commonly used in machine learning being mini-batch gradient descent, where the gradient is calculated as an average of multiple samples in order to complete a single step or iteration. Stochastic gradient descent on the other hand, performs one step for each sample, which is used in our implementation as we only have a single sample (a user's input image). The general algorithm of gradient descent is presented in pseudo-code in Algorithm 1. Starting with an arbitrary parameter vector $\theta \in \mathbb{R}^n$ which contains all n parameters needed to define an output procedural texture, as well as a loss function $\mathcal{L}$ with $\nabla\mathcal{L}(\theta)$ denoting the gradient of $\mathcal{L}$ with respect to each parameter in θ, we can update our parameters in each iteration t as shown in equation 2.6

$$\begin{aligned} v_t &= \alpha \cdot \nabla\mathcal{L}(\theta_t) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{2.6}$$

The variable $\alpha \in [0, 1]$ is called the learning rate and controls how fast θ will converge toward the optimal parameter set θ^* that minimizes our loss function $\mathcal{L}$. The strategy for updating the parameters can be refined by using a different *Optimizer*, see section 2.3.1. As we will see in section 2.3.2, loss functions generally don't depend directly on the parameters in θ, but instead on two values, a target and a sample which are generated using θ.

Result: Optimal set of parameters θ^* that minimizes loss function $\mathcal{L}(\hat{X}, X)$
Input : The loss goal l_{goal}, maximum number of iterations i_{max}, and a target target.

Randomly or strategically initialize θ;
iter $\leftarrow$ 0;
while iter $< i_{max}$ **do**
  sample $\leftarrow$ `Compute(`θ`)`;
  loss $\leftarrow$ $\mathcal{L}$(sample, target) ;
  if loss $\leq l_{goal}$ **then**
    return θ;
  end
  Backpropagate loss to find the gradient $\nabla\mathcal{L}$ w.r.t θ;
  $\theta \leftarrow$`Optimize(`θ, $\nabla\mathcal{L}$`)` ; `/* Update` θ `according to optimizer */`
  iter $\leftarrow$ iter +1;
end
return θ;

Algorithm 1: Stochastic Gradient Descent Algorithm

2.3.1 Optimizers

Optimizers are algorithms describing a strategy for gradient descent to reliably reach the most optimal value of the loss function in the least number of steps possible. This is achieved by controlling the updating of the parameters in such a way that the largest steps possible are taken towards the optimal loss value, without overshooting it or getting stuck in local minima. An analogy often used to describe an optimizer's progress is that of a ball rolling towards the lowest point on a surface (the loss function) in the direction of the downward slope (steepest descent). Given a loss function $\mathcal{L}$, the most basic optimizer updates each parameter in θ according to equation 2.6 shown earlier. Additionally, for each iteration t, the learning rate is usually updated by multiplying it with a decay term δ so that it diminishes over time. This is to prevent it from overshooting the minimum by forcing it to take smaller steps the closer it gets to the minimum. The drawback of this simple solution is that it is not very adaptive, and uses the same learning rate for all parameters, and will thus give us problems when the derivative differs between parameters. To overcome this, optimizers usually implement something called *momentum*, which adds a fraction $0 \leq \beta \leq 1$ of past update vectors to the current update vector v_t. The update steps now become:

$$\begin{aligned} v_t &= \beta v_{t-1} + \alpha \cdot \nabla\mathcal{L}(\theta) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{2.7}$$

This is effectively an exponentially weighted moving average, and for parameters where the gradient is oscillating, this addition will average out those oscillations. However, each

parameter in θ still gets updated using the same learning rate α. To correct this, an additional term dubbed *second moment* is calculated, which is a squared weighted average of the past gradients, popularized by *Adagrad* [18] and refined in *Adadelta* [19]. The combination of momentum (first moment) and second moment is used in *Adam* or *Adaptive Moment Estimation*, one of the most popular and successful optimization algorithms used in machine learning [20]. The first and second moments, m_t and v_t, for an iteration t are calculated in equation 2.8.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) \cdot \nabla \mathcal{L}(\theta) \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) \cdot \nabla \mathcal{L}(\theta)^2 \end{aligned} \tag{2.8}$$

The two parameters $0 \leq \beta_1, \beta_2 \leq 1$ are used to tune how fast the influence of previous iterations should decay, and is recommended by the authors to be set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The second moment, which is a running average of the square of the gradients, controls how much influence the learning rate α has on each parameter in θ, as shown in equation 2.9.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t} + \epsilon} m_t \tag{2.9}$$

The ϵ is a small corrective term to prevent division by zero errors. Note that all variables, except ϵ and α are vectors of size n, the number of parameters in θ and all operations are performed element-wise. Before the first iteration, m_t and v_t are initialized to all zeros, which introduces a bias towards zero for the early iterations. To prevent this, m_t and v_t in equation 2.9 are replaced with their bias corrected versions, $\hat{m}_t$ and $\hat{v}_t$, which are calculated in equation 2.10.

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.10}$$

As the iteration counter t increases, the denominator $1-\beta^t$ converges towards 1 and will thus have a negligible effect on the moments in later iterations. Most other optimizers build on these concepts.

2.3.2 Loss Functions

Gradient descent is commonly used to minimize a function that measures the difference between a generated sample $\hat{X}$, our prediction, and a ground truth X, the target, as a single real number. Such a function is often referred to as a *loss function* or *cost function* as it represents a penalty that we desire to be as small as possible. In our case, we can use it to measure the difference between a user input texture image, our target, and an image that our system has generated. A loss function should conventionally output a single scalar value, where larger values represent a bigger difference and a value of zero represents two identical images, at least as measured by the loss function. In theory, an image loss function could have any input, but the convention is shown in equation 2.11, where $\mathcal{L}$ denotes a loss function that maps an input of two image matrices $\hat{X}$ and X to a real scalar value. The matrices are real valued and

in *column-first* order which is why the width is specified first in $\mathbb{R}^{w\times h\times 3}$, where w and h is the width and height of a rendered image with three color channels.

$$\begin{gathered}\mathcal{L} : \hat{X}, X \mapsto \mathbb{R} \\ \text{where } \hat{X}, X \in \mathbb{R}^{w\times h\times 3} \text{ and } w, h \in \mathbb{N}\end{gathered} \tag{2.11}$$

The choice of loss function fully depends on what is being measured. Loss functions for images are a special case and often it is desirable to design it in a way that coincides with humans' perceived difference of two images, which can be difficult as the human visual system is very complex. Often simple functions like Mean Squared Error, explained below, are very sensitive to spatial changes in data, which is why more advanced functions are often required.

Mean Squared Error

Mean Squared Error is a popular and simple loss function that, when used on images, measures the squared difference of each pixel, sums them and returns the mean of the sum as shown in equation 2.12. This kind of loss function is useful if the goal is to reproduce the target image down to the pixel. This is however, not a good representation of how a human would judge the similarity of two images. If target image X and predicted image $\hat{X}$ are identical, except each pixel in $\hat{X}$ is shifted one column to the right, it would still be very hard to distinguish any difference between the two for a human (given a reasonable resolution). However, depending on the amount of noise in the image, most pixels are no longer identical and a large difference is measured. This also applies to any scaling, rotation or sheering of images and thus MSE has a strong spatial dependency on the compared images. Especially for procedural texturing, where high frequency features and patterns are common, we need to look for more sophisticated means of measuring differences between images.

$$\mathcal{L}_{MSE}(\hat{X}, X) = \frac{1}{w \cdot h \cdot 3} \sum_{i,j,c} (\hat{X}_{i,j,c} - X_{i,j,c})^2 \tag{2.12}$$

Neural Feature Loss

Developing a reliable method of measuring differences in images is important in inverse graphics as well as machine learning and directly affects its performance. In a way, our algorithm is only as good as our loss function which defines an upper bound on accuracy. In the paper by Guo et. al [7], several different loss functions are discussed, reaching the conclusion that a function utilizing features extracted by a neural network such as VGG19 performed best [21], a method directly adapted from a paper by Gatys et. al. [22]. Hu et. al. also used a version of this as the loss function for their neural network model, but calculated histograms on the features instead [12].

A convolutional neural network like VGG19 contains a series of sequential layers through which a generated image $\hat{X}$ and a target image X are passed. The activations in the layers form so called *feature maps*, a kind of filtered image. Each layer l has N_l filters, each producing one feature map of size M_l when vectorized, which are stored in a feature matrix $F^l \in \mathbb{R}^{N_l \times M_l}$. What we end up with is a set of these feature matrices for each of the two input images. These matrices are flattened to vectors and the dot product is utilized to measure their similarity.

The dot product of two vectors is given by the equation 2.13, where a small angle θ indicates similar vectors. Thus a small value of the dot product between two features maps of the same layer l, implies similar features.

$$\vec{v} \cdot \vec{w} = |v|\,|w| \cos(\theta) = v_1 w_1 + v_2 w_2 + \ldots v_{M_l} w_{M_l} \tag{2.13}$$

Flattening feature matrices into vectors and computing the dot product between them effectively removes the spatial information in the features. The feature correlations of layer l are given by the so called *Gram matrix* $G^l \in \mathbb{R}^{N_l \times N_l}$, produced by the dot product between the vectors in the two feature matrices as shown in equation 2.14.

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk} \tag{2.14}$$

Features can be extracted from any number of layers and collected as a set of Gram matrices $\{G^1, G^2, \ldots, G^L\}$. A weighted sum squared error of the set of gram matrices is used to calculate the final loss like shown in equation 2.15, where weights w_l are set per layer.

$$\mathcal{L}(\hat{X}, X) = \sum_{l=0}^{L} \underbrace{\frac{w_l}{4N_l^2 M_l^2} \sum_{i,j} \left(G^l_{ij} - \hat{G}^l_{ij}\right)^2}_{\text{weighted contribution per layer } l} \tag{2.15}$$

This type of loss function typically perform well for images and correspond to the human visual system, but is slowed down substantially if layers are selected far from the input, as the images have to be passed through each preceding layer. Using a pre-trained neural network such as VGG19 means we can skip the time consuming and difficult work of training on millions of images, but comes with the disadvantage of requiring input of a specific resolution of 224×224 pixels.

2.4 Automatic Differentiation

The rendering process needs to be differentiable in order to be inverted, and is a major reason why inverse rendering is a difficult problem. Why differentiability is a requirement relates to our iterative optimization Algorithm *Gradient Descent*, explained in section 2.3, which uses the gradient of a loss function to find the direction where it decreases the most.

The rendering process can be seen as one, very convoluted function of many parameters that completely describe a 3D scene, for which the derivatives with respect to each independent parameter needs to be calculated. This is completely infeasible in almost all modern renderers like those utilized by *Blender* or *3Ds Max*, but can be achieved if fully implemented in a framework that supports *automatic differentiation*.

Automatic Differentiation, usually abbreviated AD or *autograd*, is a method of automatically calculating gradients of functions, specifically functions defined in a programming language. Different libraries implement AD in different ways, but most work by breaking down complex functions into primitive expressions for which the derivative is known and all other, more complex functions, must be created using these building blocks. The order that these primitives are applied to variables is mapped in a *Computational Graph*, which is the main feature of the AD algorithm.

2.4.1 Chain Rule

As function composition is a natural part of mathematics and any programming language, an automatic differentiation framework needs to be able to handle this. To differentiate a composite function, the *chain rule* is used, which is shown in equation 2.16. This rule states that the derivative of a composite function $h(x) = f(g(x))$ is equal to the product of the derivative of the outer function f' and inner function g'. The chain rule is central to the automatic differentiation algorithm which, in a way, is nothing more than a way of recording operations used on a variable and correctly applying the chain rule.

$$\frac{df(g(x))}{dx} = \frac{df}{dg} \times \frac{dg}{dx} \tag{2.16}$$

2.4.2 Computational Graph

There exists a myriad of different automatic differentiation packages today, using different techniques. Common among them is using building blocks of functions with known derivatives as well as recording the order and settings with which these functions are applied to variables in order to correctly apply the chain rule. Often they require the user to define variables using a custom data structure that stores the state and data of the variable, often referred to as a Tensor as they support data of any dimension, be it scalars, vectors or matrices. Popular packages today, like Google's Jax, TensorFlow, or Facebook's PyTorch record computations in a data structure called a computational graph, a directed acyclic graph with nodes corresponding to variables and applied functions.

```
x = Tensor(1.5)
y = Tensor(4.0)
z = mul(x, y)  # x*y = Tensor(6.0)
h = pow(z, 3.0)  # z^3 = Tensor(216.0)
dh_dx = gradient(h, x)  # Tensor(432.0)
```

Code 2.4.1: Generic example of operations applied to variables and the gradient being calculated.

Code 2.4.1 shows a general example of operations performed on two variables and the gradient being calculated. The additional intermediate variable z is not strictly needed, but allows for a more informative computational graph. In the code example, the `gradient` function calculates $\frac{\partial h}{\partial x}$, the derivative of h with respect to x. We can do this mathematically by using the chain rule and applying standard derivatives, as shown in equation 2.17 where we have assigned shorthand functions $m(x, y) = x \cdot y$ and $p(x, y) = x^y$ to represent the multiplication and the power operations respectively, making $h(x, y) = p(m(x, y), 3.0)$ a composite function of p and m.

$$\frac{\partial h}{\partial x} = \frac{\partial p}{\partial m} \cdot \frac{\partial m}{\partial x} = \frac{\partial}{\partial m} m^3 \cdot \frac{\partial}{\partial x}(x \cdot y) = 3m^2 \cdot y = 3 \cdot (x \cdot y)^2 \cdot y = 3 \cdot (1.5 \cdot 4)^2 \cdot 4 = 432 \tag{2.17}$$

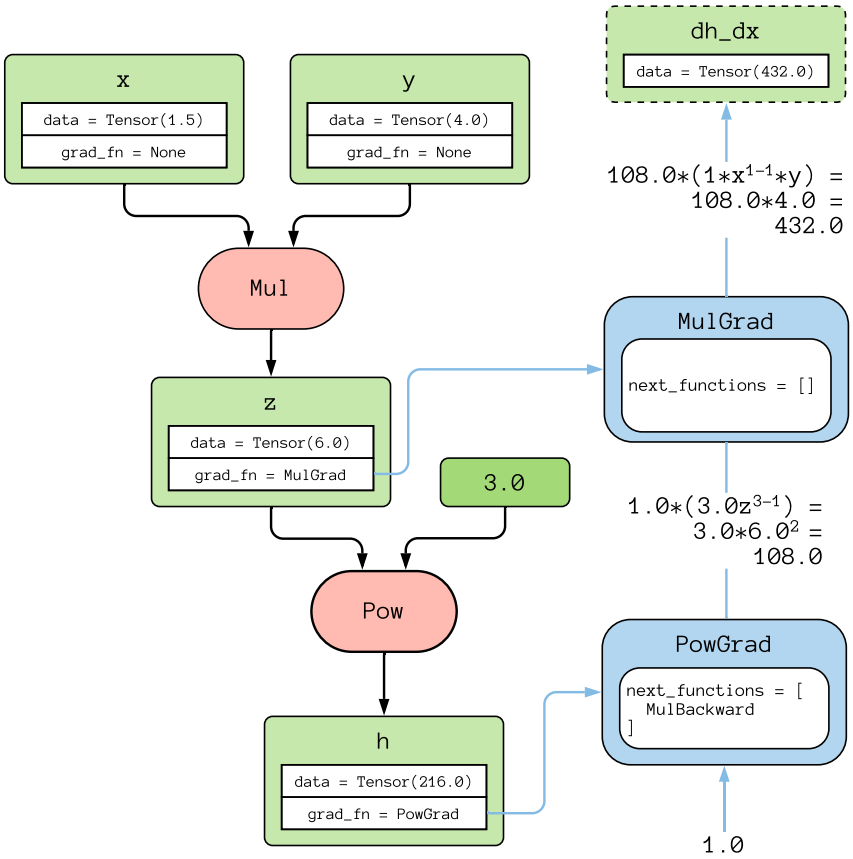


Figure 2.3: The computational graph created from the operations in Figure 2.4.1. Green nodes represent variables, red nodes operations and blue nodes the backwards operations that calculate gradients.

In Figure 2.3, the computational graph that is created as a result of the operations in 2.4.1 is shown. Green nodes represent variables, red nodes represent operations and blue nodes represent the differential version of the operator that calculates the gradient. The variable nodes have a `data` attribute to track their tensor value and a `grad_fn` attribute to track the gradient version of the function that created it. The two variables x and y are connected to the `Mul` operator node, which outputs a new variable z. The new variable z, as well as a scalar constant, are connected to the `Pow` operator which finally results in a new variable h. When the gradient is requested the flow backward commences, starting from the target node h. Because h was created from function `Pow`, the next gradient function to evaluate is `PowGrad` which has internally registered that z and the constant scalar 3.0 were used as input to `Pow`. There exists no outer function to evaluate for this gradient, so a value of 1.0 is used to multiply with the inner gradient according to the chain rule. The output of `PowGrad` is the gradient of z, which is sent to the next gradient function node `MulGrad`. The inner derivative $\frac{\partial z}{\partial x}$, where $z = x \cdot y$ is evaluated to $1.0 \cdot x^{1-1}y = y$, using common derivation rules, and is multiplied by the outer gradient from `PowGrad` according to the chain rule. The returned result 432 is the gradient of h with respect to x. This shows how the computational graph is a powerful tool to calculate gradients of arbitrarily complicated functions with respect to any input variable.

2.5 Interpolation

Interpolation is the mathematical process of estimating new data points withing a range of known data points. It is popularly used in computer graphics because interpolating data is often computationally cheaper than explicitly calculating the new data. Interpolation is a popular tool in animation where an artist can specify key points and let an algorithm smoothly estimate an object's movement in between these points using interpolation. Another popular use case, and one very relevant to this project, is the interpolation of pixel data between known vertex data. In computer graphics, vertices are almost always clustered in groups of three to form triangles and interpolation is performed between these three vertices as in Figure 2.4. Such a triangle (p_a, p_b, p_c) is defined by its three vertices p_a, p_b, p_c and any point p inside this triangle is expressed by equation 2.18.

$$p = a \cdot p_a + b \cdot p_b + c \cdot p_c \tag{2.18}$$

The set (a, b, c) contains the so called *barycentric coordinates* of point p with the restrictions that $a + b + c = 1$ as well as $0 \leq a, b, c \leq 1$. The *barycentric coordinates* act as weights when calculating each vertex's influence on the interpolated value for p and as such, should all sum to 1, as there can not be more than 100% total influence at any point in the triangle. From this stems that if a point p is equal to any of the vertices, then the weight for that vertex is 1.0 and 0.0 for all other. Let A denote the area of the triangle (p_a, p_b, p_c), and A_a denote the area of the sub-triangle opposite point p_a, then the barycentric coordinate $a = \frac{A_a}{A}$, thus, the barycentric coordinates are essentially the normalized areas of each sub-triangle, as seen in Figure 2.4b. Using the equation in 2.18, we can calculate any attribute for the fragment shown in Figure 2.4b by interpolating attributes from p_a,p_b and p_c. The trigonometry required to actually calculate the areas of the triangles is not relevant to understanding interpolation at this point, so we can assume plausible values for a, b and c.

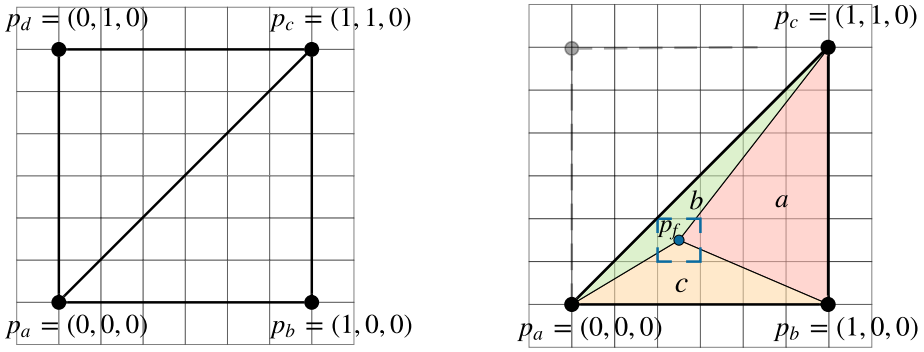
$$\text{let } a = 0.6, b = 0.15 \text{ and } c = 0.25$$
$$a + b + c = 0.6 + 0.15 + 0.25 = 1$$

We verify that all weights sum to one and lie in the interval $[0, 1]$. It is now straight forward to find the interpolated value of any attribute, for example the coordinate, of the fragment f, calculated from its center point p_f as seen in equation 2.19.

$$\begin{aligned} p_f &= 0.6 \cdot (0,0,0) + 0.15 \cdot (1,0,0) + 0.25 \cdot (1,1,0) \\ &= (0 + 0.15 + 0.25, 0 + 0 + 0.25, 0 + 0 + 0) \\ &= (0.4, 0.25, 0) \end{aligned} \tag{2.19}$$

2.6 Inverse Graphics

Inverse graphics is a broad subject that encompasses much more than just 2D texture acquisition. Many techniques for solving these problems have been developed, some closely related to this project and others completely different but with a similar goal. We have already discussed a few forms of inverse graphics and even tested some during this project but in order



(a) A plane in 3D space with each vertex's coordinate displayed.

(b) Interpolation of a point p_f inside a triangle in the plane.

Figure 2.4: Interpolation of vertex coordinates between the three vertices of a triangle for a fragment at point p_f inside the triangle. A weight $w \in \{a, b, c\}$ is equal to the normalized area of the triangle opposite the vertex p_w.

to gain a more complete understanding of our work, some of these will be reviewed from an inverse procedural texturing perspective.

2.6.1 Inverse Graphics using Neural Networks

A popular approach to the inverse graphics problem is to use neural networks. Kulkarni et. al propose using a modified Variational Auto Encoder to learn disentangled 3D transformation properties (such as rotation about an axis, or the azimuth of the light source) of a 3D scene [23]. A *disentangled* representation means that each latent variable z_i in the VAE represents a distinct and isolated transformation of the 3D scene. This is achieved, in part, by training on batches of images where all but one parameter are kept constant. A similar approach is used in a paper by Mahendran et. al, although using a Convolutional Neural Network [24] while a third solution is to use a Generative Adversarial Network that imitates a graphics renderer, which Shi et. al recently proposed to retrieve parameters for 3D faces from 2D images [25]. Neither of these solutions specialize in inverse rendering of textures, but instead focus on a small subset of possible 3D scene parameters or in the case of Shi et. al. a pre-defined set of 3D face parameters. This approach can be adapted for simple pre-defined procedural textures, but is difficult to train on more complex examples. Neural networks also require a lot of preparation, such as generating training data which in turn requires pre-designed texture models and are therefore not a flexible solution.

2.6.2 SVBRDF Aquisition

SVBRDF acquisition is a popular subdivision of inverse graphics for textures that aims to decompose 2D image textures into spatially-varying bi-directional reflectance distribution functions maps (SVBRDF for short), a type of image data that describes a texture's roughness, specularity or 3D displacement (normal map) etc. Many papers have in recent years used

different neural network models or image analysis to achieve this [26, 27, 28, 29]. However, instead of finding these maps directly, our thesis focuses on finding the parameters to a procedural texture model that would allows us to theoretically generate these maps if necessary, although we do not consider any 3D displacement or light interaction in DiPTeR.

2.6.3 Texture Synthesis

Texture synthesis is a very active research area where a sample image is used to generate a larger texture with the same pattern and style. Virtually all texture synthesis solutions work with bitmap textures, and thus act as an alternative solution to procedural textures when the goal is to generate larger textures from a sample, preserving its underlying appearance. Solving the inverse problem is not very common, nor is it often needed as it is usually easier to acquire a small texture example of a desired larger version. None the less, Wei et. al. managed to develop a framework for inverse texture synthesis, and argued that it is useful for acquiring a sample from inhomogenous textures, as it can otherwise be difficult to find a local sample that is representable [30]. Some early solutions to texture synthesis relied on feature extraction and optimization between input and output texture [31], while others utilized a patch based method where a new image is created by stitching together patches of the original sample [32]. As the popularity of neural networks grew, the field of texture synthesis found itself moving away from somewhat manual image analysis solutions and started utilizing CNNs and later GANs. Early on, Gatys et. al. proposed a method for texture synthesis based on extracting features from different levels of a neural network [22] and recent papers have found that GANs can be used to create high quality results of impressive resolutions. Notably, Früstück et. al. created a framework that supports multiple example images as input to produce a large-scale texture with very little boundary artefacts [3]. While texture synthesis can be used to quickly generate high quality output, they are still bound to the pixel space and will thus consume large amounts of storage space and lack the flexibility of procedural textures, much like SVBRDF maps.

Chapter 3
Method

As explained in the background section, it is nearly impossible to differentiate the rendering process of a system not built for that purpose, but is a requirement when performing parameter estimation using gradient descent. To solve this, we propose building a back-end renderer in a framework that is automatically differentiable in addition to the normal forward rendering system. If this system is carefully setup to mimic the main forward rendering process, finding optimal parameters to the back-end rendering function will simultaneously solve the problem of finding parameters to the forward rendering function. In the overview of our system shown in Figure 3.1, these two parts are displayed; the forward rendering framework and a differentiable version used in parameter estimation. The user-facing forward rendering is performed with OpenGL, a popular API for rendering 2D and 3D graphics while the back-end renderer is implemented in Python using PyTorch. To estimate parameters of the functions constituting a procedural texture, the iterative optimization algorithm *Stochastic Gradient Descent* is used. As explained in section 2.3 on SGD, this algorithm relies on being able to find gradients of a loss function which, in our case, would measure the visual difference between a user's input target image and the image rendered from our back-end renderer. To assist users in designing procedural texture models, we have implemented a graphical node editor, where each node keeps a reference to both an OpenGL shader, and an equivalent shader implemented in PyTorch. The resulting node graph is a tree-like data structure that visualizes data flowing from node outputs into node inputs of connected nodes. In this chapter, each part of the system will be explained and motivated in detail, ending with a presentation of our graphical user interface used to control DiPTeR.

3.1 Node Graph

Function composition in procedural texturing systems is often illustrated as a node graph, where nodes represent a procedural function. More specifically, this graph is a *directed acyclic graph*, where data flows from the leaf nodes up to the root, the *Material Output Node*, so called

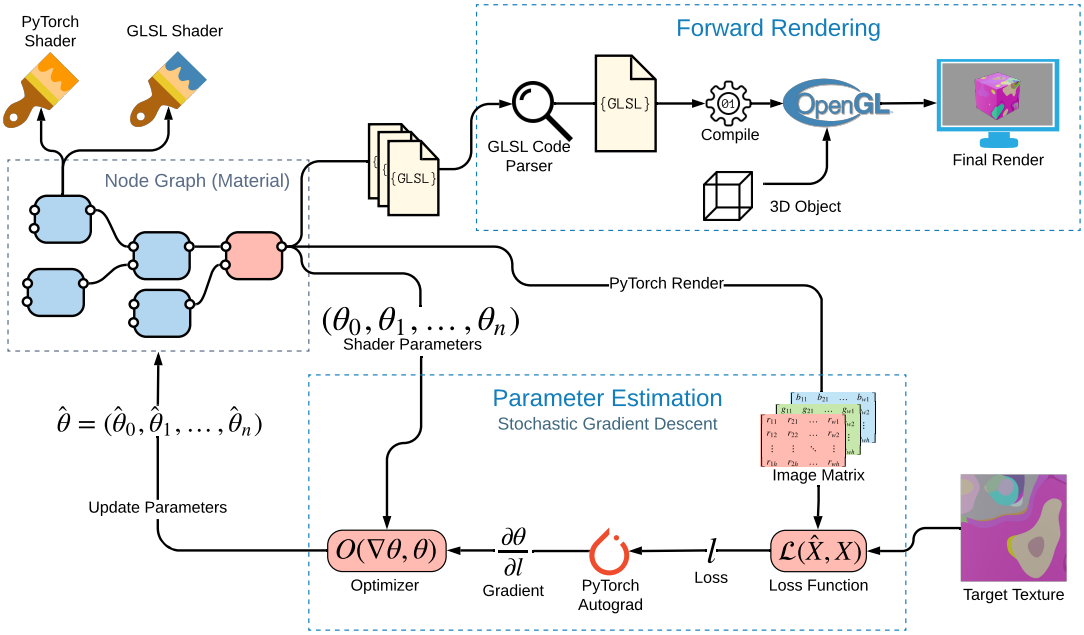


Figure 3.1: Overview of our forward and inverse rendering framework. A user starts by designing a composite shader using the node editor interface. Each node keeps a reference to two versions of the same shader, one for rendering and one for parameter estimation. The GLSL code of each node is parsed, assembled and compiled and sent to OpenGL for rendering. The optional parameter estimation is run in a loop where each iteration, a new rendered image and rendering parameters are used to estimate a better set of parameters, striving to reduce the loss.

because the resulting procedural texture is also referred to as a *material*. The graph is *directed* because data can only flow in one direction, from an output socket into an input socket, and *acyclic* because any nodes connected in a loop would give rise to infinite recursion. An example of such a node graph is illustrated in Figure 3.2. The root node, a Material Output Node, is a specialized node in DiPTeR for a specific shader called the *material output shader*. This shader acts as the `main` function in many programming languages, the entry point of execution, as explained in section 4.1, and rendering in Python must be initialized from this node. The function accepts one input that the user can not manually set, a generated pixel color which must be fed from another node.

The number and type of input sockets of a node are entirely dictated by the shader function it represents and if not connected to another node, its value can be set by the user. The only exception being the coordinates of the rendered object which is handled internally and is not user controllable. In Figure 3.2 the different functions the nodes represent are printed inside them, where some nodes represent the same function like f_1 and f_2. The resulting composite procedural texture function m is shown in Equation 3.1, omitting the fragment position arguments.

$$m = h_1(g_1(), f_2(f_1([0.1, 1.0, 0.8], 10.5), 8.3), g_2()) \tag{3.1}$$

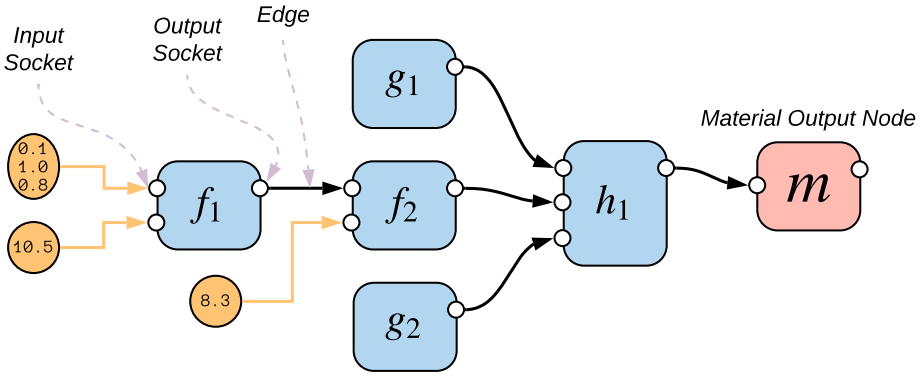


Figure 3.2: An example procedural texture model represented as a node graph where yellow nodes represent user controlled input. The material output node is the root of the graph, marked in red.

3.2 Forward Rendering

To render procedural textures onto objects we chose OpenGL, a popular and mature graphics API that is fairly easy to set up and integrate into our project. OpenGL comes with all the tools for rendering textures, procedural or not, onto both 2D and 3D objects and is highly optimized for this task. A reasonable question to ask at this point however, is why we even need a separate forward rendering setup at all if our differentiable renderer is designed to render the same result, and the reason is twofold; speed and portability. First of all, our back-end implementation is nowhere near as fast as OpenGL. Even if PyTorch has support for GPU acceleration, the overhead of differentiability and the fact that it is not specifically optimized for graphical calculations makes it too slow for real time rendering of anything but the simplest procedural textures, but performs well enough for parameter estimation. In the near future however, this might change as the team behind PyTorch is developing their own differentiable 3D renderer [6]. Lastly, the procedural textures built in our system can be exported and used in any other system that supports shaders written in GLSL. The Python code in our back end is unique to our system, and can not easily be incorporated elsewhere.

3.2.1 OpenGL Shader

Procedural textures are rendered in OpenGL using small programs called *shaders*, written in the purpose-built language *OpenGL Shading Language* or GLSL. Details on OpenGL's rendering pipeline, where different types of shaders are explained, can be found in section 4.3.1. DiPTeR mainly deals with fragment shaders which are applied to each fragment or pixel on an object, coloring them according to a function implemented in GLSL. The normalized coordinate of the point on the surface of the 3D object that is covered by the 2D fragment is an important input parameter to the fragment shader. It is calculated by the vertex shader, then interpolated and passed down for each fragment as explained in section 2.5. In our system, this is the sole purpose of vertex shaders, and thus the same vertex shader is used for all types of procedural texture models and only has to be compiled once. Therefore, when we refer to

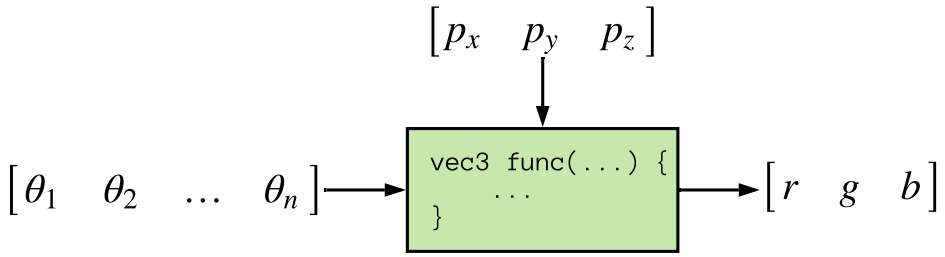


Figure 3.3: A shader in DiPTeR's forward rendering system is a function, applied to every point p on an objects surface, that takes the normalized coordinate of p and optional parameters $\vec{\theta}$ as input, and outputs a color vector for p.

a "shader" we refer to the fragment shader. Conceptually, a shader is nothing more than a function applied to each point p on an objects surface that takes a set of optional parameters $\vec{\theta}$ and the normalized coordinate of p as input, and outputs a color for p calculated from the input parameters, as shown in Figure 3.3.

3.2.2 Composing Shaders

Much of the power of procedural textures comes from the fact that they are just functions, and as such can be composed and reused as explained in section 2.1. Unfortunately however, neither OpenGL nor GLSL have any support for shader composition or even code reuse. The latter can fairly easily be supported by implementing a custom preprocessor directive that simply prepends needed source code to a GLSL shader source file before compilation, see section 4.1.1 on its implementation. Supporting dynamic function composition from Python at run-time however is more complex as GLSL is a compiled language, forcing us to recompile the shader source code each time a change to the node graph's structure is introduced. Furthermore, due to OpenGL restricting us to a single source file for our fragment shader, code from different files need to be assembled into a single file at run-time before compilation. To achieve this, we have built a custom parsing engine that reads and parses GLSL source files once, breaking down files, functions and arguments into Python objects. This code is dynamically reassembled and recompiled according to the structure of the node graph that a user has designed. OpenGL shaders accept input during runtime using so called *uniforms* which have to be uniquely named. This can be tricky when a large node graph is interpreted, as each unconnected node input will be turned into a uniform in the resulting single file source code. If all nodes are of different types this does not pose much of a problem, but as nodes of the same type use the same code (and thus the same underlying argument names), an additional system to ensure unique uniform names is required. Therefore, each node of the same type is assigned a number unique within that type, which the uniform name is based on, see section 4.1 for details.

3.3 Differentiable Rendering

While designing shaders in OpenGL is fairly easy using GLSL, reusing code and composing shaders dynamically poses a real challenge. For our differentiable shading, the problem is essentially reversed as setting up a rendering system that can reliably mimic the OpenGL renderer is difficult, while composing shaders is trivial and requires no parsing, as dynamically composing functions is innately supported by the Python programming language.

While OpenGL is a fully fledged computer graphics framework, our Python rendering engine is implemented as a simplified and highly specialized differentiable 2D procedural texture rendering engine. No 3D support is needed, as it is only used for parameter estimation by way of 2D texture similarity optimization. Furthermore, it assumes uniform directional lighting, meaning lighting is not calculated as we want to focus on reproducing underlying texture patterns and colors, and parameters governing lighting is not necessarily part of the procedural texture function. By default, OpenGL includes a fourth color channel for opacity which we ignore as it is too ambiguous to be used in inverse rendering and thus handle all images as having three channels.

To be able to differentiate the rendering procedure reliably and dynamically, we need to utilize automatic differentiation. Therefore, we implemented our differentiable renderer using Facebook's machine learning framework *PyTorch* with a built in automatic differentiation engine [9]. A few other candidates were considered early on, like Google's automatic differentiator *Jax* [33]. This framework has the advantage that it can automatically differentiate code that is almost identical to normal Python or Numpy syntax, whereas most other frameworks require the user to implement code with a specific syntax, a sort of mini-language. Additionally, TensorFlow was temporarily evaluated, but both frameworks were found to be slightly slower and not as robust as PyTorch.

As explained, an important requirement when designing PyTorch shaders is that for a specific parameter set $\vec{\theta}$ and procedural texture model, both OpenGL and our PyTorch renderer should produce the same 2D texture matrix. Therefore, the simplest way to perform shading would be to iteratively call a shader function, or composition of shader functions, on each pixel of the 2D plane being rendered, mimicking the OpenGL shader in section 3.2.1. This approach in PyTorch is explained in section 3.3.1, but proves very inefficient, why a second much more efficient but unfortunately less readable solution was devised, dubbed *matrix shading*. However, seeing as the rendering times were cut by upward a thousand times, some decrease in readability was deemed a justifiable sacrifice.

3.3.1 PyTorch Shader

Each OpenGL shader has an associated Python class in DiPTeR which keeps a reference to the GLSL source file. This class will parse the GLSL source code, compile it as well as run tests to ensure compatibility between the two implementations. Additionally, a Python shader class needs to communicate to our system what inputs the shader function accepts and their datatype (used by the node graph subsystem) as well as the actual PyTorch implementation of the GLSL shader function. How the shade function is translated from GLSL into PyTorch's own mini-language is explained in section 4.2. As in other languages, GLSL comes with a standard library of useful functions that are automatically available when writing shaders

and has to be reimplemented in Python. Fortunately, the details behind the mathematical implementation of many of these functions are revealed in the GLSL documentation, but not always. It is therefore important to run tests on each implemented library function to assure that it returns the same result as the GLSL version.

Generating Fragment Coordinates

By default convention, the pixel positions in OpenGL are located at *half-pixel coordinates*, meaning that the actual coordinate for a pixel is the position of the center of that pixel [34]. In Python, this means that the index of an element in the rendered image matrix does not directly correlate with the fragment coordinate. This does not matter much if the resolution is high, as the maximum error of the fragment position we can get if we incorrectly assume that the coordinates are given by the lower left corner of a pixel, is half a pixel, or $\frac{1}{2r}$, where r is the pixel resolution in some direction. Let w and h be the image width and height in pixels and (x, y) the index of a pixel element in the matrix, where $0 \leq x \leq w-1$ and $0 \leq y \leq h-1$. In general, a pixel's (or fragment's) position at index (x, y) is given by equation 3.2. Note that this is the coordinate in three dimensions.

$$p(x, y) = \left[x + \tfrac{1}{2w}, \quad y + \tfrac{1}{2h}, \quad 0\right] \tag{3.2}$$

In our system however, we are only interested in the normalized fragment coordinate, so as to be independent of the rendering resolution. The normalized fragment coordinate `frag_pos` is given by equation 3.3. Because we need to model our shaders in Python as similar as possible to the OpenGL standard, this is used when generating fragment positions.

$$\text{frag\_pos}(x, y) = \left[\tfrac{x}{w} + \tfrac{1}{2w}, \quad \tfrac{y}{h} + \tfrac{1}{2h}, \quad 0\right] \tag{3.3}$$

Iterative Shading

Iterative shading attempts to model the way shaders operate in OpenGL, where a fragment shader is called for each fragment (effectively pixel) independently and returns a color for that one fragment. To render an image, we first need to define its width and height in pixels and create an empty matrix of shape $(width, height, 3)$ to act as our image placeholder. Then, each position in the matrix is assigned to the output of the render function, where the normalized pixel coordinate `frag_pos` is relative to the position in the matrix while the other arguments are kept constant. This algorithm is explained in pseudocode in Algorithm

2.

Input : A shade function `Shade`, a set of parameters `params` and an image size tuple (`width`, `height`).
Output: A matrix of size `width` × `height` × 3 containing the pixel data for the rendered image.

Initialize `img` as empty matrix of size (`width`, `height`, 3);
for x = 0 **to** width **do**
  for y = 0 **to** height **do**
    frag_pos ← `GenerateFragPos(`*x*,*y*`)`;
    img [x, y] ← `Shade(`frag_pos, params`)` ;
  end
end
return img;

Algorithm 2: Iterative rendering algorithm

While this implementation is straight forward and easily translated from GLSL, the two nested Python for loops become a major performance bottleneck. All PyTorch functions are implemented and highly optimized in a C++ back end while Python for loops are notoriously slow. Furthermore, some shaders do not use the `frag_pos` variable, and will thus have the same output for each pixel. Calling the shader function $width \times height$ times, when we could call it once and replicate the output over the entire image, is a huge waste of time. Another, even worse implication is the fact that PyTorch will add a node to the computational graph for each call to a PyTorch function. As such, this approach will add $width \times height$ duplicate operation nodes to the graph, making it much slower to traverse in the backward step as well as memory inefficient. Using only built in functions as far as possible can give a thousandfold performance boost, especially when rendering higher resolution images. How this is achieved is explained in the next section.

Matrix Shading

One major disadvantage to PyTorch is that it does not have a `map`-function that can apply a function to each position in a matrix. This would solve our problem, as we could keep our iterative method while removing the need for slow Python loops. Instead, we can generate input and output parameters for every pixel simultaneously in a matrix the same size as the final rendered image. PyTorch uses its own data structure called a `Tensor` that represents data of any shape or dimension of any primitive Python type. The functions in the PyTorch library operate on these tensors element-wise, which we can utilize to calculate parameters and color data for the entire image in one call to the shade function. Any parameter of size d, be it a scalar where $d = 1$ or a color vector where $d = 3$, can be extended to a matrix of size $(width, height, d)$ to represent that parameter over the entire image. The fragment coordinates can now be generated once in the superclass, and used by all shaders, as a matrix of size $width, height, d = 3$), where each element is a vector $[p_x, p_y, p_z]$. The output of a shader function is now not only the color of a single pixel but the color of all pixels, in other terms; the fully rendered image. Thanks to PyTorch's element-wise operators, the implementation does not have to change at all in some cases, and in most cases, it is only a matter of handling the extra dimensions. The performance gains are particularly significant

for shaders that do not utilize the fragment coordinates, and therefore have a uniform color output across the image. In these shaders, the color only have to be calculated once, and then repeated using PyTorch's `repeat` function, instead of running the shader thousands of times, once for each pixel, like in the iterative method. The other advantage of this method is that the resulting computational graph is much smaller, as each operation only has to be added once.

$$
\begin{aligned}
F[:,:,0] &= \begin{bmatrix} f_x(0,0) & f_x(1,0) & \dots & f_x(w,0) \\ f_x(0,1) & f_x(1,1) & \dots & f_x(w,1) \\ \vdots & \vdots & \ddots & \vdots \\ f_x(0,h) & f_x(1,h) & \dots & f_x(w,h) \end{bmatrix} \\
F[:,:,1] &= \begin{bmatrix} f_y(0,0) & f_y(1,0) & \dots & f_y(w,0) \\ f_y(0,1) & f_y(1,1) & \dots & f_y(w,1) \\ \vdots & \vdots & \ddots & \vdots \\ f_y(0,h) & f_y(1,h) & \dots & f_y(w,h) \end{bmatrix} \\
F[:,:,2] &= \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}
\end{aligned}
\tag{3.4}
$$

For visual reference, the fragment coordinates matrix is shown in equation 3.4. Let $f(x, y)$ denote the function that generates the fragment coordinate in equation 3.3, at each image integer index (x, y), and let f_x, f_y, f_z denote the x,y, and z coordinates returned by f, respectively. The equation shows the matrix F that contains all the fragment positions, such that each position i, j in the matrix contains the vector $f(i, j)$. The matrix rendering method has an important speed advantage, but there does exist a truly limiting drawback when it comes to loops, where the loop count is dictated by an argument to the shader function. As mentioned, all arguments are matrices, and can therefore not be used as arguments to most Python control flow such as `if`-statements and `for`-loops but some of these cases can be solved by using PyTorch alternative functions, see section 4.2.

3.3.2 Composing Shaders

As Python is our main programming language and the shaders are defined in Python, shaders are already symbolically composed via the node graph. Rendering a composed shader can therefore be easily achieved by traversing the node graph in a correct order and using the output of shaders as input to other connected shaders. Rendering with composite shaders have to be initialized from a node in the graph rather than a shader, as shaders are isolated units, in essence a single function, without any references to other shaders.

The node graph is traversed in depth first order starting from the Material Output root node. Rendering is recursive, and for current node N, the algorithm starts by checking each input socket if it is connected and if so, the connected node is recursively rendered and the output is saved to be used as a parameter for Ns shading function. If the socket is not connected, the parameter value is fetched from the input widget, set by the user in the graphical interface. The unconnected parameters are saved in a dictionary and constitutes the proce-

dural texture parameters $\vec{\theta}$ we want to optimize during parameter estimation. Finally, the shade function is called using the collected parameters and the rendered matrix is returned together with the parameter dictionary for N. For every recursive render call, this dictionary is extended until all connected nodes have been rendered and the parameter dictionary is complete for the graph. This algorithm is shown in Algorithm 3. The `Shade` function being called at the end is the shade function of the shader that the current node being rendered contains. The call to `GetUniqueUniform` is not strictly needed to render an image in Python, but is needed when translating the parameter values to OpenGL in order to know which parameter value belongs to which uniform.

```
Function Render(width, height)
    Initialize params_dict as empty dictionary;
    Initialize arguments as empty dictionary;
    for socket ∈ GetInputSockets() do
        arg ← GetArg(socket);
        uniform ← GetUniqueUniform(arg) ;
        if socket is connected then
            node ← GetConnectedNode(socket);
            // recursive rendering
            value, pd ← node.Render(width, height);
            Update params_dict with pd;
        else
            value ← GetValueFromSocket(socket);
            Add value to arguments with key arg;
        end
        Add value to params_dict with key uniform;
    end
    return Shade(arguments), params_dict;
end
```

Algorithm 3: Recursive rendering algorithm for composite Python shaders.

3.4 Parameter Estimation

Parameter Estimation, also referred to as *Texture Matching* in DiPTeR, is our subsystem that lets users automatically estimate parameters of a procedural texture model they designed in order to minimize the difference between a target bitmap texture and the procedural texture output. This is posed as an optimization problem as described in equation **??**, where the goal is to find optimal parameters $\vec{\theta}^*$ such that they minimize a loss function $\mathcal{L}(\vec{\theta}^*)$. In our system however, we split the rendering of our procedural texture $\hat{X} = R(\vec{\theta})$ and the loss measurement $l = \mathcal{L}(\hat{X}, X)$ because in PyTorch, all operations performed on Tensors are tracked locally in the Tensor datastructure. This enables us to have the loss function only indirectly depend on the procedural parameters and instead be a function of two parameters, a Python rendered image and a user supplied target image. In other words, we can render the image in one function and calculate the loss in another function, but all operations performed on the parameters in either function will be registered in the Tensors. This is crucial, as

it enables us to find the derivative $\frac{\partial \theta_i}{\partial l}$ of any parameter θ_i in $\vec{\theta}$ with respect to the final loss $l = \mathcal{L}(\hat{X}, X)$, even though the loss function does not directly depend on parameter θ_i. Parameter estimation is controlled through our texture matching GUI, see section 3.5.2.

3.4.1 Loss Functions

We make three loss functions available to users: a simple Mean Squared Error loss, Squared Bin Loss as well as a more sophisticated Neural Loss. The MSE loss function works exactly as described in section 2.3.2 and is very fast and exact, as it measures similarity between individual pixels. It can perform well for very simple procedural textures where an exactly matching result is possible to obtain. In the general case however, this will not be the case, especially not when noise is used in the textures. The Squared Bin loss function is similar to an MSE loss function, but divides the image into regions or bins of $N \times N$ pixels and calculates the average color of each bin, before finally returning the mean squared error of the averages. This loss function can work well for finding general patterns in an image, but can fail within the regions. Making the regions larger makes it less sensitive to noise, but performs worse within regions, and if the regions are made too small, the loss function will exhibit the same traits as the normal MSE.

Building on findings by Guo et. al., an even better solution should be to utilize machine learning to automatically unveil the underlying patterns in the textures, and directly comparing them [7]. This type of loss function is dubbed a Neural Loss, as the general idea is to pass the images through a neural network, where image features will be extracted by the different layers in the networks, as explained in section 2.3.2. This loss function is much more spatially independent than the other two and should focus more on the general look of the textures. For example, the other two loss functions would be fairly bad at recognizing that two different textures of the same type of wood are similar, because locally, these textures can have very different color values. A neural loss function however, could extract features pertaining to the direction of the grain or the presence of knots etc. A comparison of the performances of these loss functions is found in section 5.3.1.

3.4.2 Gradient Descent

Gradient Descent is performed much like described in Algorithm 1. Each iteration, an image is rendered using parameters $\vec{\theta}$ and a loss value is calculated between the rendered image $\hat{X}$ and the target image X. All operations performed directly on the parameters in $\vec{\theta}$ or on variables created as a result of an operation on a parameter in $\vec{\theta}$ are registered in the computational graph. The next step is to utilize PyTorch's automatic differentiation package to traverse the computational graph backwards from the loss, calculating gradients for each of our parameters. Given the gradients, the direction of the steepest ascent of the loss function relative to each parameter's dimension is known. A small gradient g_i for parameter θ_i means that only a small correction of this parameter is needed in order to reach an optimal loss value, and vice versa. Lastly, it is up to the chosen optimizer to calculate new values for each parameter based on its gradient. Before the next iteration is started the rendered Python image is updated in the graphical interface and likewise the new parameters are sent to the OpenGL shader program, rendering the updated texture. This is an important visual

tool and enables users to verify that the algorithm is correctly optimizing the parameters. Additionally, the new loss value is plotted to display the progress over time. The algorithm terminates when the maximum number of iterations has been reached, or the loss value is less than a specified threshold and the new parameter values can be applied to the procedural texture, if the results are satisfying.

3.5 Graphical User Interface

While it is possible to build procedural textures and perform parameter estimation programmatically with our framework, it is faster and more convenient to interact with DiPTeR through our graphical user interface. The GUI is created using *PyQt5*, the Python bindings for the popular C++ library *Qt*, an extensive suite for cross-platform embedded and desktop UI development [11, 35].

3.5.1 Node Editor Interface

Node graphs, and thereby procedural textures, are designed by the user in a *node editor interface* shown in Figure 3.4. This is a graphical user interface that lets users spawn nodes from a menu and drag edges to connect nodes' input and output sockets. Next to an unconnected input socket, a widgets allows users to manually control parameter values of a node's shader. DiPTeR currently supports input values of a number of different types: integer or floating number input, color input by opening a color selector window or scalar vector input. To the right of the node editor is the OpenGL render area, where the procedural shader is rendered onto a chosen object in real time that can be rotated and zoomed by the user and any changes to input values or the structure of the graph will immediately be reflected in the rendering. A menu with available node types can be opened by right clicking the node editor area. Just above the node editor scene is a toolbar where users can create new materials by clicking the *plus* button, as well as select a material to display from the drop-down list. DiPTeR also supports saving and loading materials to JSON format.

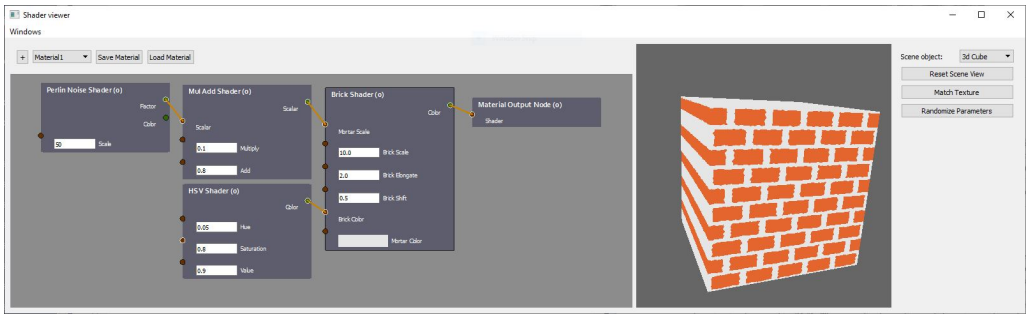


Figure 3.4: Node Editor Interface in DiPTeR. To the right of the node editor is the OpenGL rendering area, where the procedural texture is shown being rendered onto a cube.

3.5.2 Texture Matcher

Parameter estimation is controlled by the user through our texture matching interface shown in Figure 3.5. Here, a user can control various aspects of the gradient descent algorithm such as maximum number of iterations or set a loss threshold that, once reached, signals the completion of the algorithm. Furthermore, the user can select the loss function and optimizer to be used in gradient descent and change their settings. These are the two components that have the largest influence on the success of the parameter estimation process. The loss function sets the upper bound of our result, as a loss of zero is the best outcome we can achieve and if that does not correspond to an optimal similarity between our two images, then the loss function is clearly not good enough. On the other hand, it is the optimizer's job to make sure this value is reached, or at least as close as possible. Apart from the settings panel, the interface is divided into four views. In the top left the procedural texture is shown as rendered by OpenGL, and in the middle, as rendered by the Python back end. Both of these views are updated each iteration and allows the user to assert that the two systems render a similar result and that the algorithm is progressing properly. The top right view displays the target image and hopefully, by the end of the last iteration, all three views will display visually similar images. Lastly, the bottom view plots the loss value versus iteration, hopefully showing a steady decline as the optimization progresses.

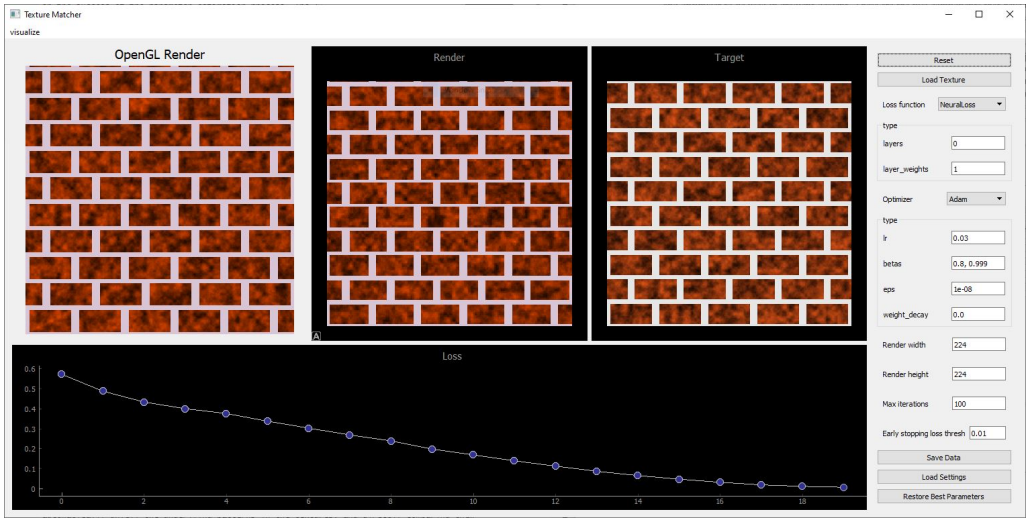


Figure 3.5: Graphical interface of the texture matcher, allowing the user to control parameter estimation. The loss has successfully converged to the threshold and all three views show a similar image.

3.5.3 Loss Visualizer

The loss value at the end of the gradient descent procedure will almost always be lower than the initial value. However, it is very possible that the final value is a local minima and that a better parameter estimation could be found. One way that this process can be debugged and analyzed is by explicitly plotting the loss for a range of parameter values as a surface plot. Unfortunately, it can only be visualized for two parameters at most, occupying the x- and y-axes, as the third axis is used by the loss value itself. Figure 3.6 shows the Loss Visualizer

interface, where a user can select up to two parameters from their procedural texture in the list on the left and view the resulting, interactive loss surface. A user can also override the minimum and maximum values for each parameter, thereby focusing or restricting the parameter space. From this, it is possible to visually and manually find the optimal loss value relative to the selected parameters. Furthermore, the gradient descent progress can be plotted as a line on top of this surface, allowing the user to see exactly where the algorithm potentially got stuck. In Figure 3.6 we can clearly see that the gradient descent algorithm, marked with red dots, gets stuck in a local minimum "valley" and does not reach the minimum loss marked with a magenta *plus* symbol.

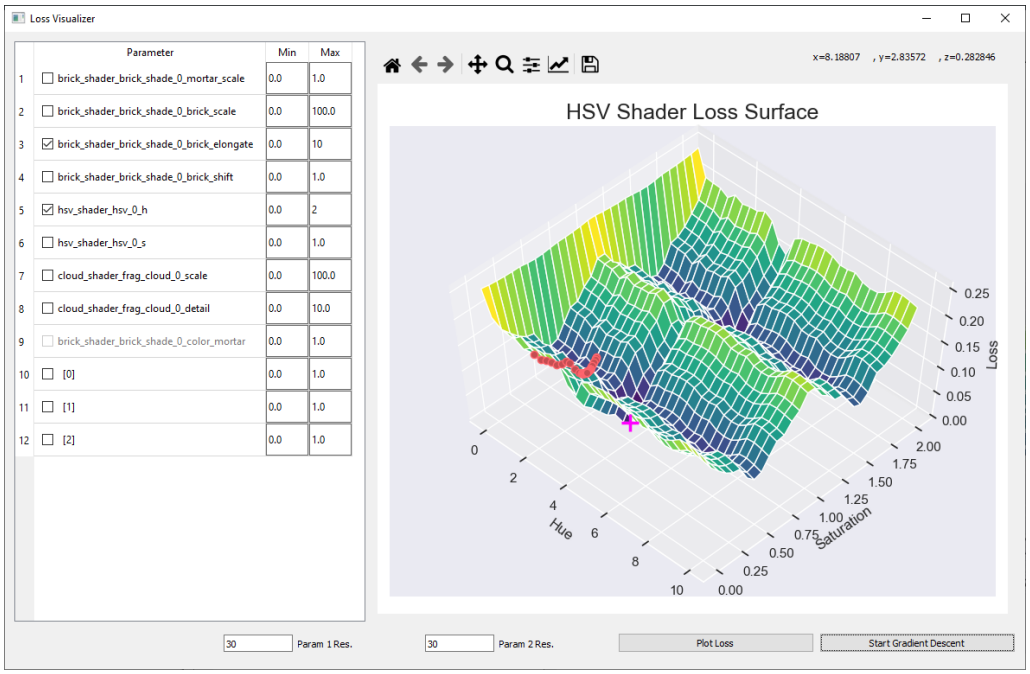


Figure 3.6: Loss Visualizer interface showing available parameters and their value ranges to the left and the resulting loss surface to the right. The minimum loss value is marked with a magenta "plus" symbol.

Chapter 4
Implementation

This chapter outlines the details behind some of the implementation choices in DiPTeR, specifically related to GLSL shader composition and code reuse in section 4.1, and how PyTorch shaders were implemented to mimic the GLSL shaders in section 4.2. Finally, some useful details behind the OpenGL rendering pipeline are presented.

4.1 GLSL Shader

As in many other programming languages, a fragment shader written in GLSL needs a main function as an entry point of execution. However, to simplify shader composition, all shaders in DiPTeR are defined as a single function (but may call other functions), with one exception; the Material Output shader which contains the only `main` function and must therefore always be the root node in any procedural texture. The different types of shaders in the OpenGL graphics pipeline, see section 4.3.1, are all assembled into a *Program* whose inputs are controlled via parameters called *uniforms*. Each time a new node is connected to the node graph, all the needed code is assembled into one file by our GLSL code parser, where each unconnected parameter of each shader node in the graph is controlled via a uniform. These uniforms, much like any function parameters, need to have unique names which can be partly solved by appending the name of the shader function to the parameter name. However, while the parser ensures that needed function definitions are only inserted once, there are no guarantees that the names of the functions are unique. A modified function name is therefore constructed by also appending the name of the containing source file to the function name and because all shaders are defined in the same folder, this guarantees a unique function name. However, in the case where multiple shader nodes of the same kind are used in a node graph, this is not enough to guarantee a unique uniform name. A `Material` class is used to add nodes to the node graph as well as assigning a number to each node that is unique among nodes of the same kind. This number is passed to the GLSL code parser responsible for generating the assembled GLSL code and uniforms can now be uniquely named by following the

format `<unique function name>_<node number>_<parameter name>`.

4.1.1 `#import` Preprocessor Directive

OpenGL has a number of so called *preprocessor directives*, operators that are processed before shader compilation and are called using the pattern `#<name-of-directive> <arguments>`. None of these allow the user to import and reuse code from other files and while this is not a vital function for our project, it is certainly very helpful. Consequently, we implemented our own preprocessor directive called `#import` which takes the name of a file to import as an argument. This directive is then parsed by our GLSL parser and the needed files are appended to the code to be compiled. This allows us to call functions from other files as if they were a part of the importing file's definition.

4.2 PyTorch Shader

As explained in section 3.3.1, instead of implementing shading as a process that iteratively renders a texture pixel by pixel, we implement rendering as a function of matrices and render the image all at once. This means that a parameter in GLSL of type `vec3`, a vector of length 3, corresponds to a `Tensor` of shape $(W, H, 3)$ in Python, where W and H are the width and height of the texture being rendered and the last dimension matches the GLSL parameter type. Unfortunately this implementation is a necessity due to lack of rendering support in PyTorch. While such a solution is immensely faster, it comes with some serious drawbacks in readability that can be mostly fixed with a few tricks that we will describe below, but also a complete lack of support for some important functionality pertaining to control flow in GLSL. Under the hood, all unconnected shader inputs are handled as scalars or vectors and are dynamically converted to matrix form just before rendering, while connected inputs are fetched by rendering from the connected node.

Ultimately, measuring and confirming the correspondence between the GLSL and PyTorch implementations is done by explicitly measuring the difference between the two frameworks' renders for a variety of parameter combinations. However, it will greatly help the developer to correctly implement shaders in Python if the two languages can utilize similar syntax and call the same functions. To achieve this, we first have to implement parts of the GLSL standard library using PyTorch functions. This can be fairly tricky as the actual source code for the functions are not publicly accessible and may differ between GPU vendors, but some functions have a direct equivalent already implemented in PyTorch, or the mathematical formula for a given function may be revealed by the GLSL documentation. Furthermore, the use of some functions like the `step` function is problematic as it is not inherently differentiable. We implement it using the `sign` function as seen below, which is not strictly differentiable either, but can be backpropagated. Note that this and all other PyTorch functions support both scalars, vectors and matrices as all operations used are applied element-wise.

```
1 def step(edge: Union[float, Tensor], x: Tensor) -> Tensor:
2     """
3     `step` generates a step function by comparing x to edge.
```

```
    For element i of the return value, 0.0 is returned if x[i] < edge[i],
    and 1.0 is returned otherwise.
    """
    return (torch.sign(x-edge) + 1) / 2
```

Next, we demonstrate how the usage of vectors in GLSL can be translated to PyTorch. It is possible to extend the `Tensor` class in Python, one for each of the vector types in GLSL so that new vectors can be created by for example calling `vec3(...)`, but this is difficult as the implementation in PyTorch is convoluted, and we do not want to risk breaking the differentiability and portability. Instead, we create a library `vec.py` to handle the creation of vectors (that are actually matrices), as well as supporting GLSL's syntax of retrieving elements of a vector by calling `var.x`, `var.y`, `var.z` or `var.w` as well as creating new vectors. In Code 4.2.1 the GLSL implementation of our HSV shader is shown and in Code 4.2.2 the equivalent PyTorch implementation is shown. By implementing the GLSL standard library and the `vec` library, the implementations can be kept almost identical.

```
vec3 hsv(vec3 frag_pos, float h, float s, float v) {
    vec3 c = vec3(h,s,v);
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
```

Code 4.2.1: GLSL implementation of the HSV shader function.

```
# Import standard GLSL library as gl and vector library vec
from dipter.shaders.lib import glsl_builtins as gl, vec
def shade_mat(self, h: Tensor, s: Tensor, v: Tensor) -> Tensor:
    c = vec.vec3(h,s,v)
    K = vec.vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0)
    p = torch.abs(gl.fract(vec.xxx(c) + vec.xyz(K)) * 6.0 - vec.www(K))
    return vec.z(c) * \
        gl.mix(vec.xxx(K), torch.clamp(p - vec.xxx(K), 0.0, 1.0), vec.y(c))
```

Code 4.2.2: PyTorch implementation of the HSV shader function, equivalent to the GLSL implementation in Code 4.2.1.

GLSL supports the basic control statements that we are used to in programming like `while`, `for` and `if` statements. One of the flaws with using matrices as parameters becomes obvious if we try to use them in the condition of any Python control flow. For `if` statements, this can be solved with the PyTorch function `where` which can evaluate a condition on each position in our matrix, and return a specific value at each position that evaluates to true or another value where it evaluates to false and the only drawback is breaking the syntax conformity. Unfortunately, there is no equivalent function for loops which means we are unable to handle the case where the loop counter is passed from a function argument.

There are a few other discrepancies that need to be managed. Firstly, while a shader in GLSL can utilize a number of different data types such as integers and booleans, the use of such datatypes will break differentiability in PyTorch and we will therefore always work with 32-bit floating point data. In practice however, we can support the use of parameters of any datatype in PyTorch as long as we do not let them be connectable, that is, prohibit the user from connecting such inputs to other nodes. This means that the value of such a parameter must be manually set by the user and are never converted to matrix form. This is utilized, for example, in our *math shader* which returns the result of applying a selected mathematical operator to two values. The mathematical operator, `+`, `-`, `/` or `*` is selected by comparing the value of an input scalar integer, marked unconnectable and therefore not converted to matrix form, using `if` statements. The same workaround can be used to address the problem of letting the user control a loop counter, with the only drawback that it will be uniform across the entire texture. Secondly, most implementations of OpenGL supports division by zero, or must at least not lead to an interruption, which differs from Python where division by zero leads to a runtime exception. To remedy this, we add a small float constant to the denominator of any division operations in Python. Finally, the use of bitwise operators are common in GLSL but are only supported for integer types. This is also true for PyTorch, but as integer types are not differentiable the use of bitwise operators in DiPTeR is not possible. Fortunately, it is almost always possible to rewrite code without these operators, possibly at the expense of performance.

4.3 Tools

4.3.1 OpenGL Rendering Pipeline

The OpenGL rendering pipeline consists of multiple different stages, defining a workflow starting with 3D vertex data to finally producing 2D pixels on a screen [36]. Some of these stages are programmable by the user, marked in green in Figure 4.1, where the Vertex Shader is the only one *required* to be defined by the user. In the Vertex Specification step, no shader is used, but a list of vertex positions is defined to be rendered as well as a list of triangles that define the faces between these vertices. The Vertex Shader is run on each of these vertices and takes exactly one vertex as input and outputs exactly one vertex. This shader typically also takes three transformation matrices as input that define the scaling, translation and rotation of the object. These are used to output vertices in *clip space* where coordinates are normalized to the interval $[-1, 1]$. Next up is the Tesselation Shader and the Geometry Shader. Both manipulate or create geometry but neither are used in our project, as we are only interested in the look of the objects texture, not its geometry. Before the Fragment Shader stage, there is a crucial *rasterization* stage that needs to be completed. This stage turns primitives (like triangles) into fragments, by looping over screen pixels and checking if they lie inside the boundaries of a primitive. If they do, a fragment is created for this pixel and the fragment is given per-vertex values that are interpolated between the vertices that make up the primitive, see section 2.5. The fragments are closely related to screen pixels, but contain more information and each pixel can spawn more than one fragment, depending on multisampling parameters. Each fragment is then sent, one by one, to a Fragment Shader (also called Pixel Shader) which is where all the computations are done that define the final color

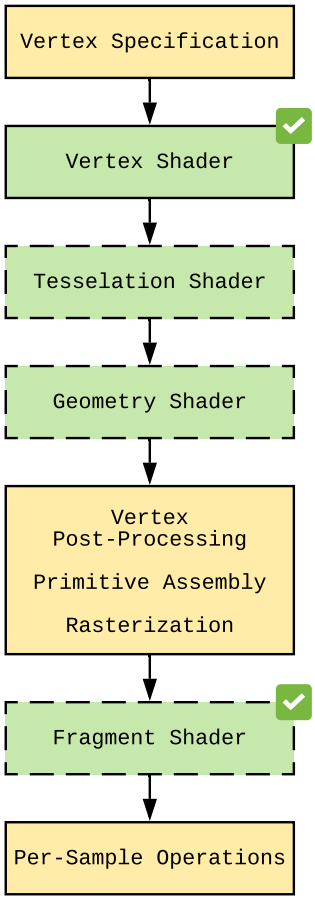


Figure 4.1: An overview of the different stages in the OpenGL Rendering Pipeline. The stages marked green are programmable by user-defined shaders, where the dashed outline marks optional stages, while the yellow stages are controllable through OpenGL function calls. A check mark icon marks the only stages that are used in this project.

of a single fragment. In essence, a fragment shader defines a function $f(V, P) \mapsto (r, g, b, a)$ that takes a set of interpolated per-vertex parameters V and a set of user defined parameters P as input, and outputs a fragment color on the form (r, g, b, a), where r, g, b are the value of the red,green and blue channels, respectively and a is the alpha value of the fragment.

Chapter 5
Evaluation & Discussion

In this chapter the performance of DiPTeR is evaluated by constructing three different test shader models for which rendering speed and parameter estimation accuracy is measured for a number of different combinations of target textures, loss functions and optimizers. All the experiments are performed on a desktop computer with the following specifications:

- Operating System: Windows 10
- GPU: NVIDIA GeForce GTX 670 2GB
- CPU: Intel Core i7 3770K @ 3.50GHz
- RAM: 16GB DDR3 @ 667MHz

Unfortunately, the version of CUDA supported by this graphics card is too old to utilize PyTorch's GPU accelerated Tensors and thus all operations are performed exclusively on the CPU.

5.1 Shader Models for Evaluation

To evaluate DiPTeR, three procedural test shader models of different levels of complexity were created. All models are required to contain a Material Output root node which serves as the output of the shader model. This node does nothing except return the final rendering and will thus be excluded from the explanations of the different node setups. For each shader $\boldsymbol{M}_i$, two textures are rendered: $\boldsymbol{T}_{i1}$ with only two parameters changed from their default values and $\boldsymbol{T}_{i2}$, where each parameter is assigned a random value. These textures are used as targets for our parameter estimation evaluation runs, denoted $\boldsymbol{X}$ in section 2.3.2 on loss functions. For the last most complex shader model we include an additional real life texture target $\boldsymbol{T}_{33}$ which is not rendered from the shader model itself.

5.1.1 HSV Shader Model M_1

The first shader model, M_1 is designed for simplicity using only a single HSV shader node which outputs an RGB color controlled by three values: *hue*, *saturation* and *value*. Consequently, the model depends on a total of three scalar parameters and uses 17 PyTorch functions to render, meaning an equal amount of function nodes make up the resulting computational graph for calculating gradients. The node graph and the rendered texture (using default parameters) are displayed in Figure 5.1.

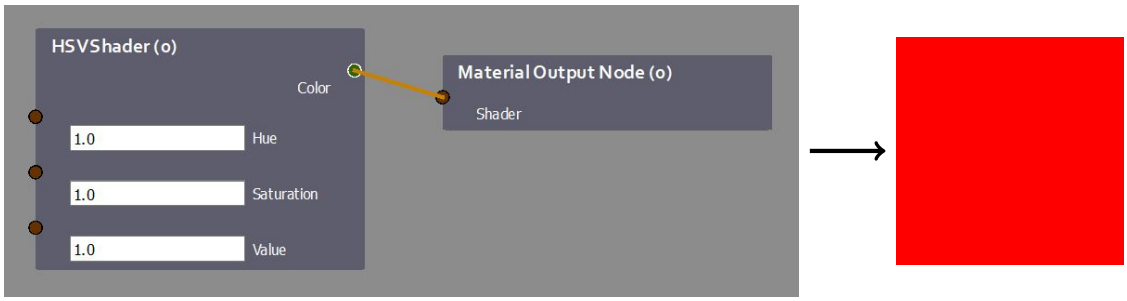


Figure 5.1: HSV test shader M_1 using a single HSV Shader node with the rendered texture to the right, using default parameters. The HSV Shader does not utilize the fragment position argument, resulting in a uniformly colored texture.

The two generated targets are presented in figures 5.2a and 5.2b where the former is rendered by changing only two parameters, the hue and the saturation values, and the latter is rendered by randomizing every parameter.

(a) Target texture T_{11} where the hue and saturation values are changed.

(b) Target texture T_{12} where all parameters have been randomized.

Figure 5.2: The two test target textures for HSV Test Shader model M_1.

5.1.2 Simple Brick Shader Model M_2

The next test shader M_2 is a basic brick shader with a total of three shader nodes: a cloud shader, HSV shader and a brick shader. The color of the bricks are controlled by a color from the HSV shader, where the *value* parameter is modulated by noise from the cloud shader. The model has a total of 9 user controllable input parameters, one of which is a color vector

parameter with three channels, so in reality the model depends on 11 parameters. The brick shader is fairly complex, resulting in a big leap in number of PyTorch functions used, from 17 for M_1 to 544 for M_2. The brick test shader node graph and resulting rendered texture for default parameters are displayed in Figure 5.3.

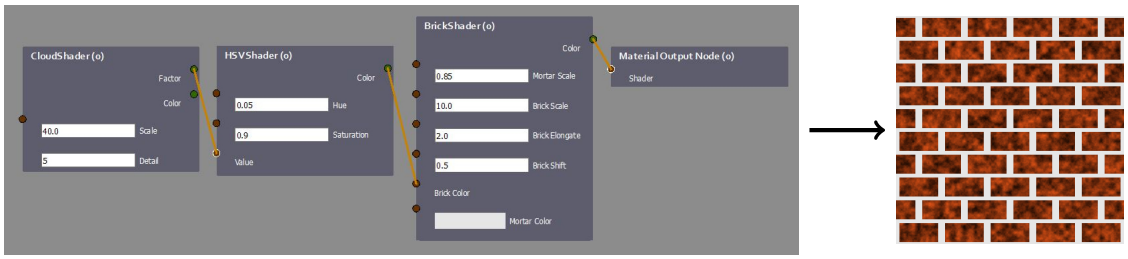
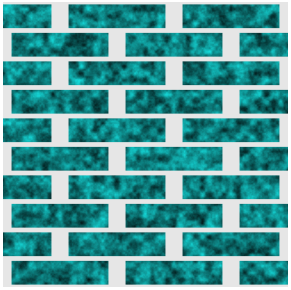
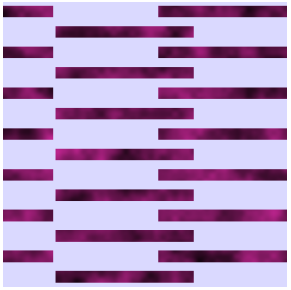


Figure 5.3: Brick shader M_2 using a Cloud Shader connected to a HSV Shader connected to a Brick Shader with the resulting rendered texture to the right, using default parameters.

The two generated targets for M_2 are presented in figures 5.4a and 5.4b respectively. The first is rendered by changing only two parameters, the elongation of the bricks and the hue of the bricks, and the last is rendered by randomizing every parameter.



(a) Target texture T_{21} where the elongation and hue of the bricks are changed.



(b) Target texture T_{22} where every parameter is randomized.

Figure 5.4: The two test target textures for shader model M_2.

5.1.3 Advanced Brick Shader Model M_3

The final shader model M_3 is an advanced version of M_2 using a different brick shader adapted from *Blender's* brick shader implementation in GLSL, as well as using classic 3D perlin noise, which is much more complex than the fractal brownian motion noise used in the cloud shaders. In total, M_2 depends on 26 parameters and a total of 2498 PyTorch functions. The node graph of M_3 and rendered texture using default parameters are shown in Figure 5.5.

For this shader model, we generate two targets in a fashion similar to M_1 and M_2 shown in figures 5.6a and 5.6b respectively. The first image shows the target rendered by changing only two parameters, the scale of the bricks and the color bias while the second image shows a target rendered by randomizing each parameter. Additionally, this shader is advanced enough

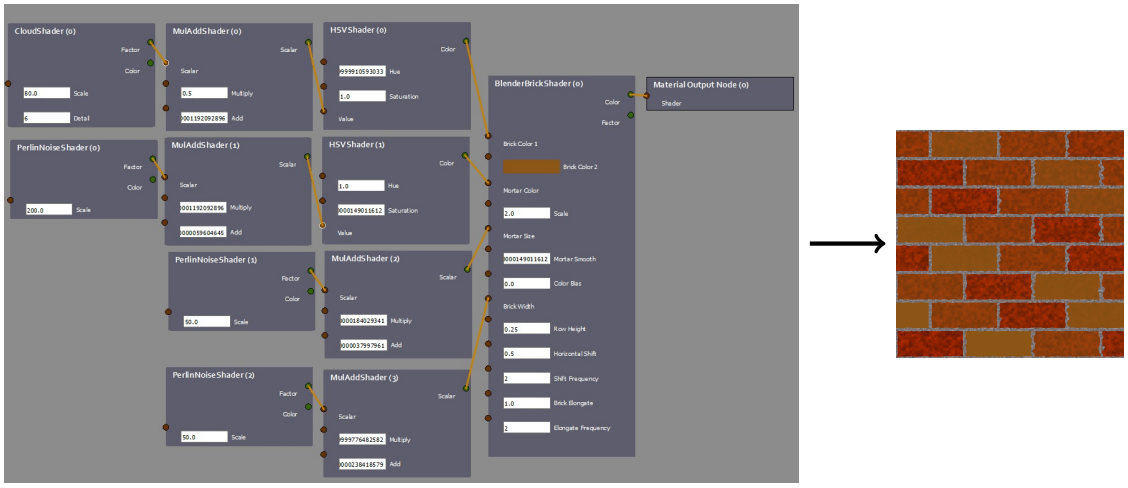
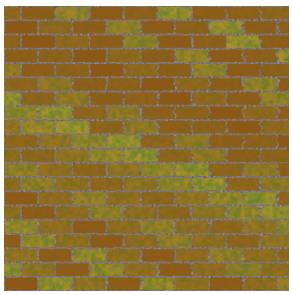
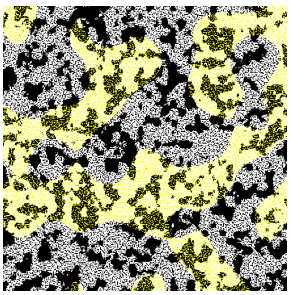


Figure 5.5: Advanced brick shader M_3 using an adapted version of *Blender's* brick shader as well as multiple instances of classic 3D perlin noise. The resulting rendered texture using default parameters is shown on the right.

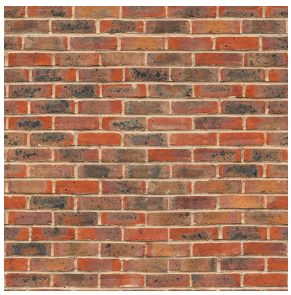
that it could render something resembling a real life texture, which is why we test it on a third, real life texture of a brick wall, shown in Figure 5.6c.



(a) Target texture T_{31} where the brick scale and color bias have been changed.



(b) Target texture T_{32} where every parameter has been randomized.



(c) Real life target texture (photograph) T_{33} of a brick wall.

Figure 5.6: The three test target textures for shader model M_3 including an additional real life target.

5.2 Python Rendering Performance

The rendering time of our Python shaders can be a major bottleneck during parameter estimation as one texture image is rendered per iteration of gradient descent. The rendering performance is therefore evaluated on the three shaders models described in section 5.1 for a number of sizes. We use 150 linearly sampled sizes between 10×10 and 1000×1000 pixels plus an additional selected size of 200×200 which is the default render size in DiPTeR. For each shader and render size, a texture is rendered three times and the average CPU execution time is recorded. The results are plotted in Figure 5.7 on a log-log plot where the y-axis shows the rendering time in milliseconds while the number of pixels is displayed on the x-axis. A

dotted horizontal line shows the threshold for real time performance, here defined as rendering 30 times per second or more, equivalent to a rendering time of approximately 33ms, and a dotted vertical line marks the default rendering size of 200×200 pixels.

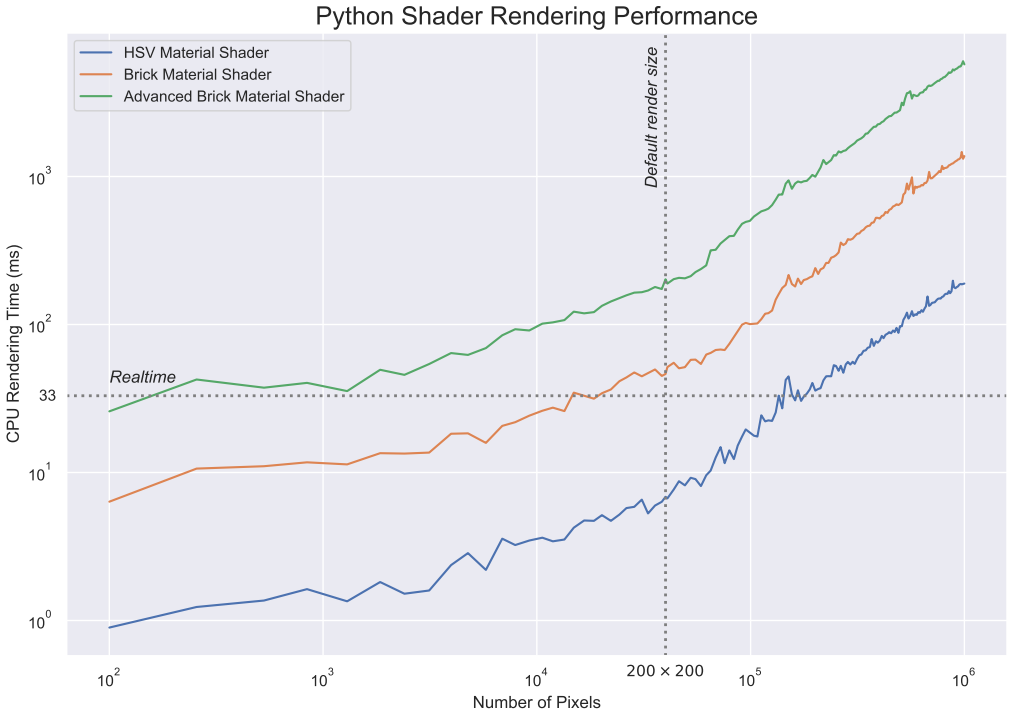


Figure 5.7: The CPU rendering time in milliseconds it takes to render an image of a certain size, measured in milliseconds per total number of pixels, for the back-end Python render engine. A horizontal dotted line indicates the threshold for real time performance (33ms/render) and a vertical dotted line indicates the number of pixels in the default render size (200×200 pixels).

We can observe that the rendering time scales linearly with the number of rendered pixels which is expected. Furthermore, rendering performance is largely dictated by the complexity of the shader function. For $\boldsymbol{M}_1$, rendering a texture of default size, 200×200, only takes around 7 milliseconds and we can achieve real time performance for up to a resolution of about 360×360 pixels. For $\boldsymbol{M}_2$, rendering a texture of default size takes considerably longer at about 46 milliseconds and we can only achieve real time performance for textures of up to around 116×116 pixels. Lastly, for the most advanced shader function real time performance is already surpassed at a resolution of 16×16 pixels and rendering a texture using the default resolution takes around 200 milliseconds. While real time performance is by no means a requirement for the Python shading, it will directly affect the time it takes to perform the parameter estimation, as seen in section 5.3. A constant part of the total rendering time does not directly depend on the resolution or the number of functions used, but on the number of nodes in the node graph as well as the number of inputs that have to be checked for those nodes. For $\boldsymbol{M}_1$, this is about 1.5 milliseconds, for $\boldsymbol{M}_2$ about 6 milliseconds and for $\boldsymbol{M}_3$ about 20 milliseconds or on average about 1.8 milliseconds per node.

5.3 Parameter Estimation

The algorithm itself used for parameter estimation is fairly simple but needs to solve two difficult problems. First of all, we need a reliable way of finding a path towards a minimum of our loss function and unfortunately there is no way of knowing if the reached minimum is actually a global minimum or a local minimum. Second, the loss functions global minima must correspond to a satisfactory similarity between the target and generated texture, and should ideally be as smooth as possible in all dimensions. In this section, we will therefore survey different optimizer strategies, which solve the first problem, and different loss functions, pertaining to the second problem, for our three test shaders and discuss the advantages and disadvantages of the different combinations.

We rendered a texture from each of our three test shaders where only two variables have been changed, $\boldsymbol{T}_{i1}$, which allows us to plot the loss as a surface dependent on these two variables and trace the parameter estimation progress along this surface. It also serves as an easy starting point of evaluation as we later evaluate parameter estimation using every variable, targets $\boldsymbol{T}_{i2}$.

Evaluating the parameter estimation is done separately for each of the test shader models $\boldsymbol{M}_1$, $\boldsymbol{M}_2$ and $\boldsymbol{M}_3$ for each possible combination of our three implemented loss functions, see section 3.4.1, and optimizers. The optimizers we chose to test are the popular *Adam* and the predecessor *RMSprop*, both of which comes bundled with PyTorch. When performing the gradient descent, the initial state of the shader model is equivalent to that shown in the node setups in figures 5.1, 5.3 and 5.5 respectively. Normally when gradient descent is used to optimize a neural network the initial parameters are randomized, which is not best practice in DiPTeR as a user will typically design their shader model as far as possible and then use the parameter estimation as a final step to find even better parameter values. Randomizing initial parameters will effectively undo all of the user's progress and will in most cases result in a higher initial loss. All textures are compared as matrices where the color values are stored as floating point values ranging from 0.0 to 1.0. All plots follow the convention that parameter estimations for $\boldsymbol{T}_{i1}$ are shown as solid lines while parameter estimations for the $\boldsymbol{T}_{i2}$ targets are shown as dotted lines, additionally using blue or orange color when using Adam or RMSprop, respectively.

5.3.1 Parameter Estimation Speed

Difficult optimization problems often require hundreds of iterations to reach a low loss value and the execution time of a loss function can thus greatly influence the overall parameter estimation time. In Figure 5.8 the average iteration time per shader per loss function is presented. This data was produced by executing 10 iterations of gradient descent three times with both Adam and RMSprop and then averaging over those times. We did not separate the data between the different optimizers as there was no significant difference between them. The rendering resolution was set to 224×224 pixels, the required resolution of input to VGG19 used in the neural loss function, and the rendering time is included in the plot. It is clear that Mean Squared Error and Squared Bin Loss are much faster functions than the Neural Loss and the almost constant difference between their execution time suggests that the total time mainly scales with the render resolution and complexity of shader model. In cases where a neural loss function does not clearly result in a more accurate result, there is not much jus-

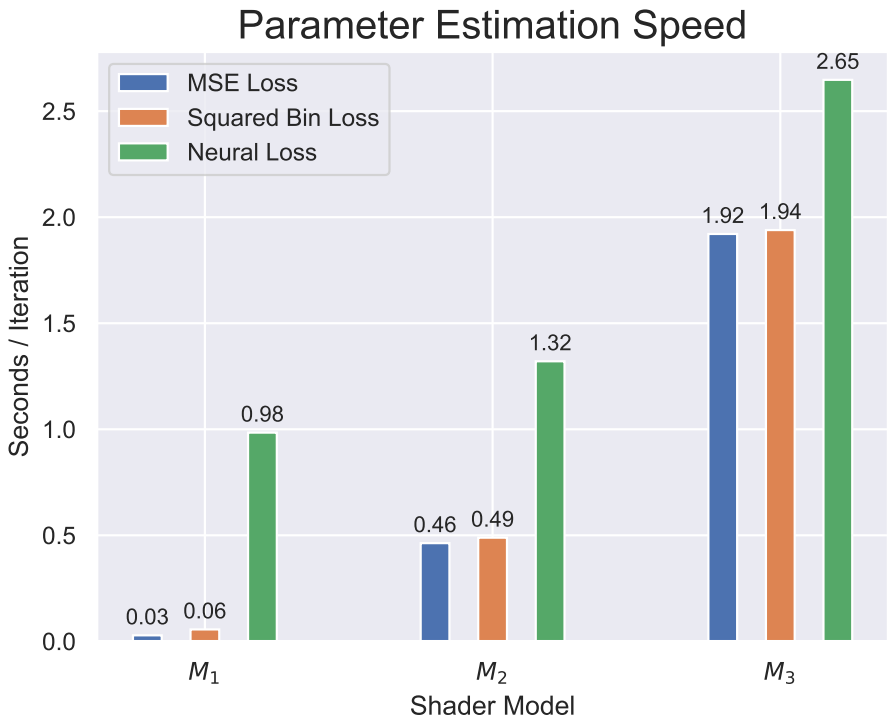


Figure 5.8: Execution time in seconds for an iteration of gradient descent for different shader models and loss functions. The iteration time includes the process of rendering an image from the shader model once using a resolution of 224×224 pixels.

tification for its greater execution time. However, when the model is sufficiently complex so that the rendering time is the largest factor, in terms of iteration time, it does not matter much which loss function is used.

5.3.2 Parameter Estimation using MSE

Mean Squared Error is a loss function that measures the difference in color values for each channel and pixel between our generated image $\hat{X}$ and target X. It is a very simple loss function and is such much faster than, for example, a neural loss function and is well suited for finding the similarity between simple textures without patterns as it heavily depends on spatial information. The minimum loss value that can be achieved is 0.0 corresponding to an exact match in color values and the maximum loss is 1.0 corresponding to a maximum difference between each pixel which stems from our use of a floating point image format. The settings used for each shader model and optimizer when evaluating Mean Squared Error loss is presented in table 5.1. Each optimizer and target requires different settings to perform well and typically, a more difficult problem requires a lower learning rate and more iterations to optimize successfully.

				Adam		*RMSprop*	
M_i	T_{ij}	**Optimizer**	**lr**	β_1	β_2	α	**Momentum**
M_1	T_{11}	Adam	0.15	0.5	0.999		
M_1	T_{11}	RMSprop	0.09			0.9	0.0
M_1	T_{12}	Adam	0.1	0.5	0.999		
M_1	T_{12}	RMSprop	0.09			0.9	0.0
M_2	T_{21}	Adam	0.05	0.9	0.999		
M_2	T_{21}	RMSprop	0.01			0.75	0.4
M_2	T_{22}	Adam	0.05	0.8	0.999		
M_2	T_{22}	RMSprop	0.02			0.75	0.6
M_3	T_{31}	Adam	0.05	0.9	0.999		
M_3	T_{31}	RMSprop	0.01			0.75	0.4
M_3	T_{32}	Adam	0.05	0.8	0.999		
M_3	T_{32}	RMSprop	0.02			0.75	0.6
M_3	T_{33}	Adam	0.04	0.9	0.999		
M_3	T_{33}	RMSprop	0.01			0.75	0.4

Table 5.1: Optimizer and loss function settings when running gradient descent using Mean Squared Error loss.

HSV Test Shader M_1

First, parameter estimation using an MSE loss function with $\boldsymbol{M_1}$ is evaluated and the results are shown in Figure 5.9. The target loss threshold is set to 0.0025 corresponding to an average difference in color values of $\sqrt{0.0025} = 0.05$ or 5%. We are able to reach the loss threshold in only a few iterations using either optimization methods. As expected, a few more iterations are required to reach the threshold for target $\boldsymbol{T_{12}}$ as more parameters have to be optimized, even though the initial loss is actually lower than for $\boldsymbol{T_{11}}$. Furthermore, a simple model like $\boldsymbol{M_1}$ seems to be more efficiently optimized using RMSprop than the more modern Adam, possibly due to Adam requiring a few initial iterations in order to build up momentum. For this experiment, using a learning rate of under 1.0 proved a necessity for RMSprop, where using a larger value resulted in a divergence of the loss. Lowering the first moment parameter β_1 for Adam proved very helpful in order to diminish its oscillating behaviour. Typically, MSE is not a very good loss function for image comparison, but as $\boldsymbol{M_1}$ is uniform across the entire image, the spatial dependence of MSE is not a problem and proves very efficient for similar problems.

Simple Brick Test Shader M_2

Next, we use the MSE loss function to estimate parameters for the more complex brick shader model $\boldsymbol{M_2}$. Unlike $\boldsymbol{M_1}$ this model is not uniform accross the image and features lots of patterns and even pseudorandom noise which MSE is very ill equipped to handle. Looking at the resulting data in Figure 5.10 we can see that for target $\boldsymbol{T_{21}}$ we are able to find a parameter set that significantly lowers the loss value from around 0.18 down to a minimum of 0.056 at iteration 126 for Adam and 0.065 for RMSprop significantly earlier at iteration 85. In all four cases we lowered the learning rate compared to $\boldsymbol{M_1}$ as this is a more difficult problem with more parameters, as well as increased the β_1 value to 0.9 in order to stabilize the progress.

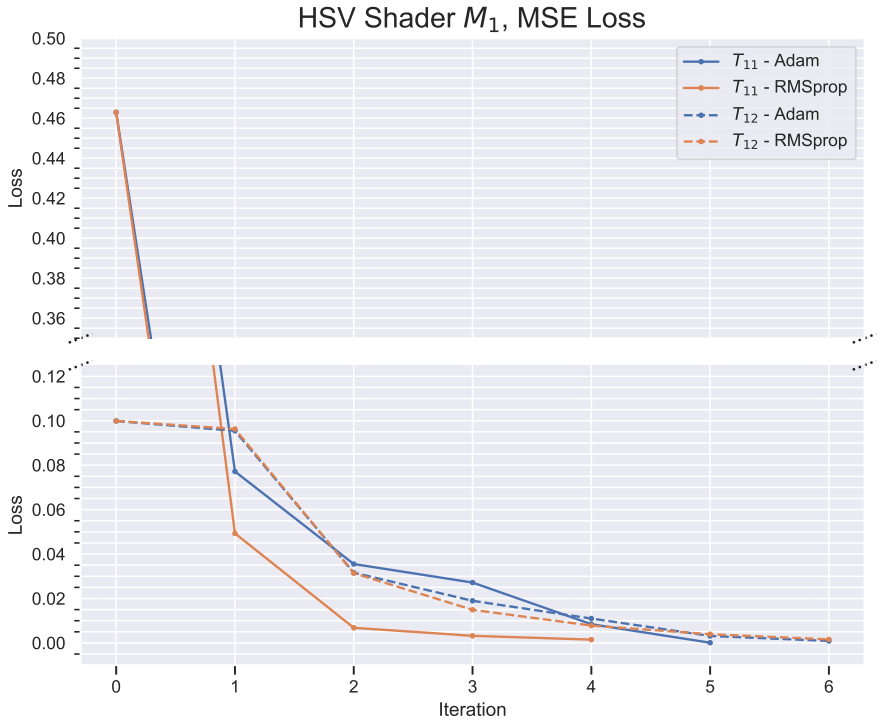


Figure 5.9: Results of evaluating parameter estimation of material $\boldsymbol{M}_1$ using Mean Squared Error loss. The runs with target $\boldsymbol{T}_{11}$ are plotted as solid lines while runs with target $\boldsymbol{T}_{12}$ are plotted with dashed lines.

Momentum was also utilized here for RMSprop which seemed to help with finding a slightly better minimum. As stated before, MSE is not a good choice for images with patterns but can be efficient at finding the overall color tone of the image. For both optimizers for target $\boldsymbol{T}_{21}$ the blue color of the bricks are successfully reproduced, but not the brick shape, even though Adam managed to produce a result with slightly elongated bricks. As expected, the optimization for target $\boldsymbol{T}_{22}$ was not as successful and did only reach a minimum loss of about 0.089 for both optimizers which converged around iteration 75. The final renders with the optimized parameters are shown in Figure 5.11. For a relatively complex shader like $\boldsymbol{M}_2$ it seems to be only possible to reproduce the overall color tone of the target texture, but any pattern is either completely lost as in renders 5.11b and 5.11d or the shape is not correctly recovered as in renders 5.11a and 5.11c.

Advanced Brick Test Shader M_3

Finally test shader $\boldsymbol{M}_3$ is evaluated using the MSE loss function and the results are shown in Figure 5.12 and best renders in Figure 5.13 where the same optimizer settings were used as with $\boldsymbol{M}_2$ for targets $\boldsymbol{T}_{31}$ and $\boldsymbol{T}_{32}$. For target $\boldsymbol{T}_{31}$, the initial loss is relatively low at just above 0.025 and both optimizers reaches a minimum loss of 0.0052, fairly close to our threshold of 0.0025. This demonstrates the fact that MSE often does not correspond with how humans judge similarity, because when comparing the initial render in 5.5 and the target $\boldsymbol{T}_{31}$ in 5.6a they clearly portray very different brick walls. The optimization looks rather successful on paper, but looking at the render from the parameters at the point of minimal loss in Figure

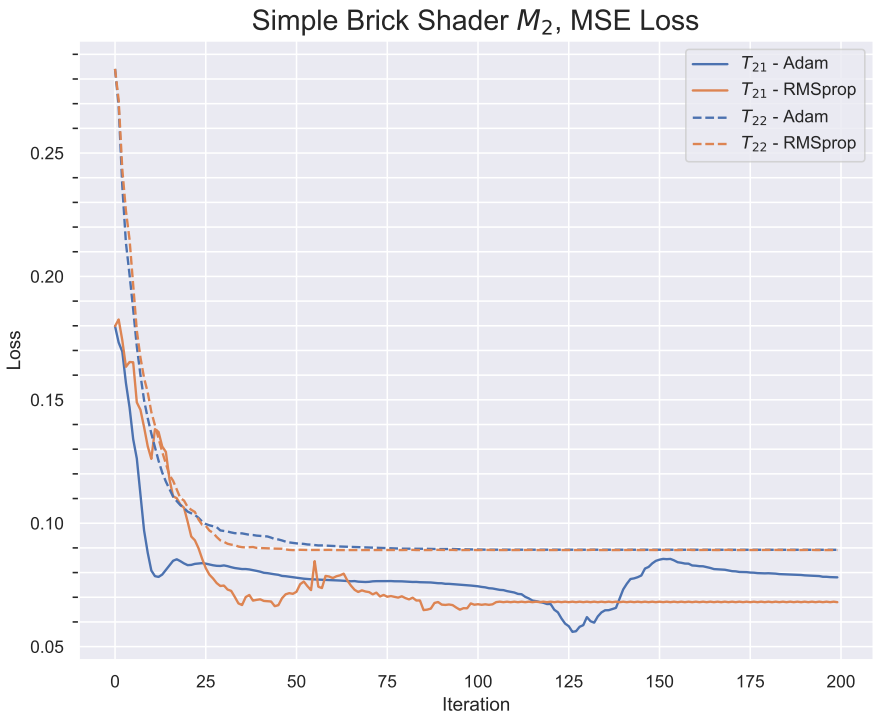
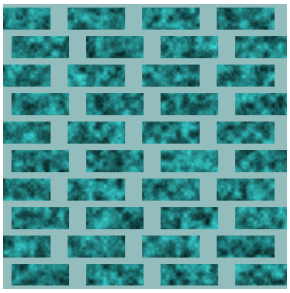
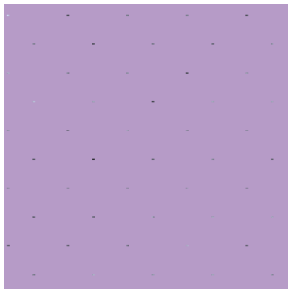


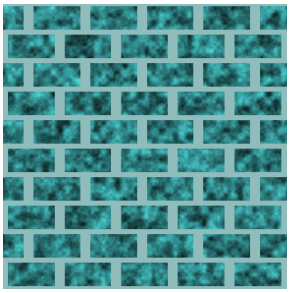
Figure 5.10: Results of evaluating parameter estimation of material M_2 using Mean Squared Error loss. The runs with target T_{21} are plotted as solid lines while runs with target T_{22} are plotted with dashed lines.



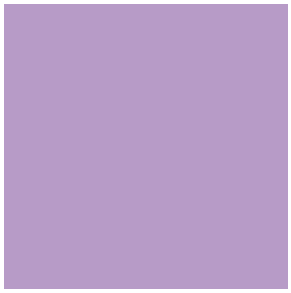
(a) Target T_{21} using Adam.



(b) Target T_{22} using Adam.



(c) Target T_{21} using RMSprop.



(d) Target T_{22} using RMSprop.

Figure 5.11: The four rendered textures at the point of minimal loss for M_2 using MSE loss corresponding to the four plots in Figure 5.10 with combinations of optimizers Adam or RMSprop and targets T_{21} or T_{22}.

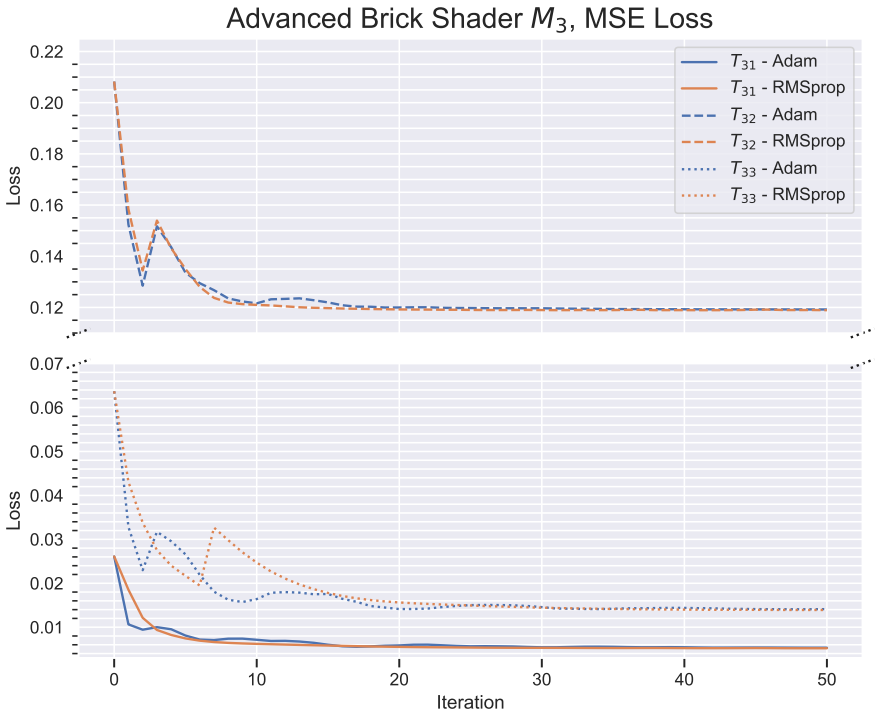
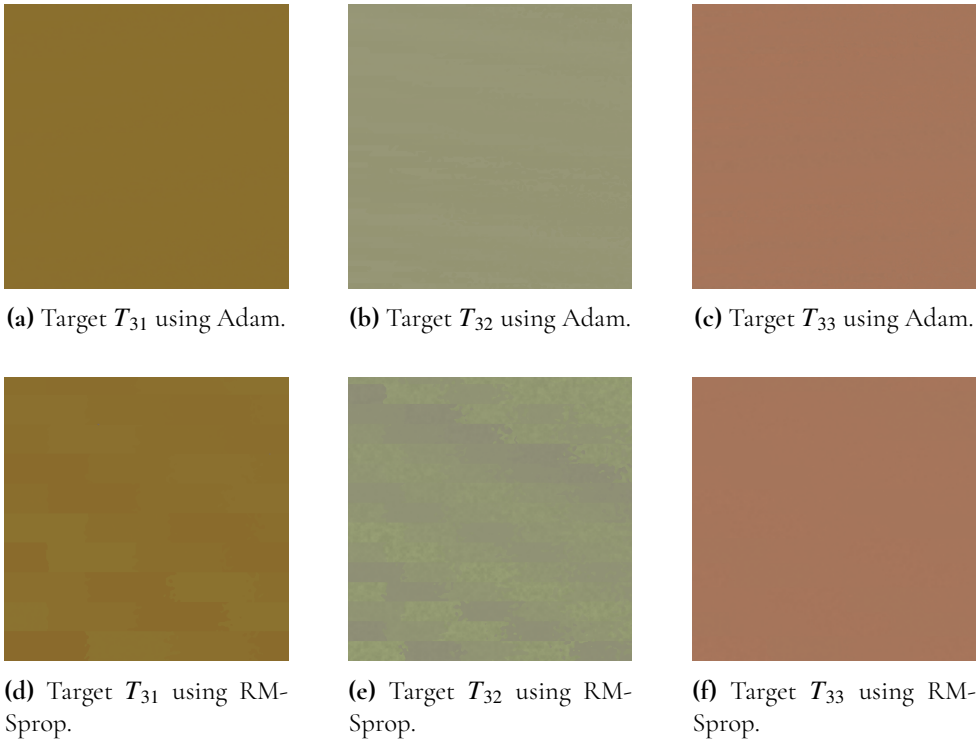


Figure 5.12: Results of evaluating parameter estimation of material M_3 using Mean Squared Error loss. The runs with target T_{31} are plotted as solid lines, target T_{32} with dashed lines and the additional real-life target T_{33} with dotted lines. The iterations have been truncated from 200 to 50 iterations as the loss converges before iteration 50.

5.13, only the average color has been captured. As expected, the performance for target T_{32} using random parameters is worse, only reaching a minimum loss of about 0.12 around iteration 20 for either optimizer. The additional real-life target T_{33} plotted with dotted lines lies somewhere in between, starting with a loss of about 0.065 and converging to a loss of about 0.014. The same behaviour is shown here as seen before, where the renders in sub-figures 5.13c and 5.13f show that the MSE loss only restores the overall color of the target. Clearly a Mean Squared Error loss is not optimal when estimating parameters for complex shaders with an extensive use of patterns.



(a) Target T_{31} using Adam. **(b)** Target T_{32} using Adam. **(c)** Target T_{33} using Adam.

(d) Target T_{31} using RMSprop. **(e)** Target T_{32} using RMSprop. **(f)** Target T_{33} using RMSprop.

Figure 5.13: The six rendered textures at the point of minimal loss for M_3 using an MSE loss function corresponding to the six plots in Figure 5.12 with combinations of optimizers Adam or RMSprop and targets T_{31}, T_{32} or T_{33}.

5.3.3 Parameter Estimation using Squared Bin Loss

Squared Bin Loss or SBL is effectively a version of Mean Squared Error that first downscales images into squared bins of a user controllable size and calculates the mean of each bin before returning the Mean Squared Error of the averages. This allows the loss function to better judge image similarity without comparing every pixel individually and can for some types of images help discern underlying patterns among noise. The loss value limits are the same as for the MSE loss, a minimum loss of 0.0 and a maximum loss of 1.0. Unlike MSE however, a loss value of 0.0 for SBL does not necessarily mean that the images are identical, simply that the average of each bin is identical. The optimizer settings used when running gradient descent for SBL is presented in table 5.2.

				Adam		*RMSprop*		*SBL*
M_i	T_{ij}	**Optimizer**	**lr**	β_1	β_2	α	**Momentum**	**Bin Size**
M_1	T_{11}	Adam	0.15	0.5	0.999			10
M_1	T_{11}	RMSprop	0.09			0.9	0.0	10
M_1	T_{12}	Adam	0.1	0.5	0.999			10
M_1	T_{12}	RMSprop	0.09			0.9	0.0	10
M_2	T_{21}	Adam	0.03	0.8	0.999			8
M_2	T_{21}	RMSprop	0.02			0.9	0.4	8
M_2	T_{22}	Adam	0.03	0.8	0.999			8
M_2	T_{22}	RMSprop	0.02			0.9	0.4	8
M_3	T_{31}	Adam	0.03	0.8	0.999			8
M_3	T_{31}	RMSprop	0.015			0.99	0.4	8
M_3	T_{32}	Adam	0.03	0.8	0.999			8
M_3	T_{32}	RMSprop	0.01			0.99	0.2	8

Table 5.2: Optimizer and loss settings when running gradient descent using Squared Bin loss.

HSV Test Shader M_1

For uniform textures like those produced by $\boldsymbol{M}_1$, SBL should give exactly the same result as a normal MSE loss, which is why it is curious that the loss threshold is reached, on average, one iteration earlier for SBL than MSE as shown in Figure 5.14, even if the exact same optimizer settings are used. This is probably more or less the result of chance, as the gradients are affected by the additional PyTorch functions used and so even if the same loss value is returned for the same data, the gradients are slightly different. Generally, this loss works equally well for uniform shaders as MSE and we were able to reach the loss threshold of 0.0025 for both targets in five or less iterations. Similar to using MSE loss, RMSprop managed to slightly outperform Adam in this simple use case as well.

Simple Brick Test Shader M_2

For test shader $\boldsymbol{M}_2$ we used much lower learning rates than for $\boldsymbol{M}_1$ and a higher β_1 for Adam while adding some momentum for RMSprop. The bin size was set to 8 meaning bins of 8×8 pixels are averaged, as lower values did not contribute to the accuracy and larger values would introduce too much averaging and thereby lose detail. The results are shown in Figure 5.15 where it is apparent that for target $\boldsymbol{T}_{21}$ RMSprop is outperforming Adam by a significant amount. At iteration 45 RMSprop has reached its best loss value of approximately 0.025 while Adam reaches its best loss value of 0.03 at iteration 197. For target $\boldsymbol{T}_{22}$ the results are more even, Adam reaching a loss of around 0.059 at iteration 127 while RMSprop reaches a loss of around 0.062 at iteration 146. In Figure 5.16 we can see that the results are similar to those for MSE; the overall color has been captured, at the expense of the background color, but not the shape of the bricks although the results look slightly better for target $\boldsymbol{T}_{22}$ where the bricks have been conserved but scaled down.

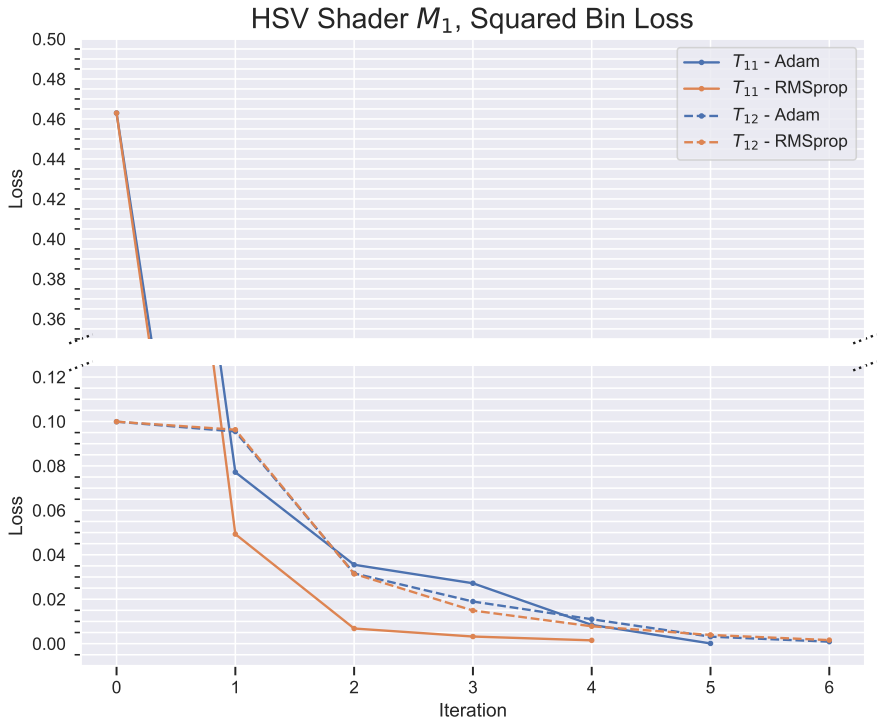


Figure 5.14: Results of evaluating parameter estimation of material M_1 using Squared Bin loss. The runs with target T_{11} are plotted as solid lines while runs with target T_{12} are plotted with dashed lines.

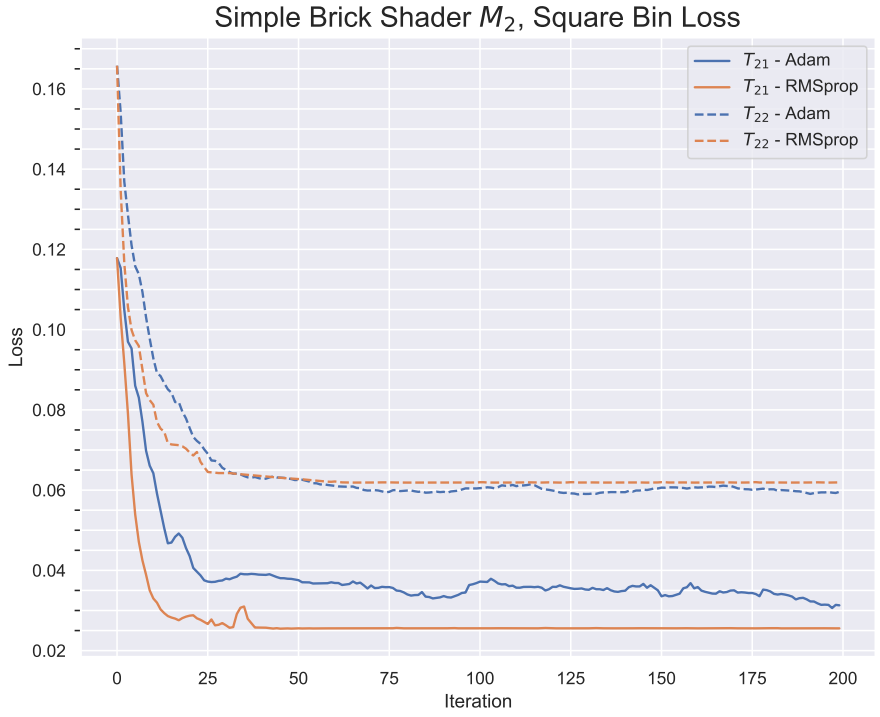
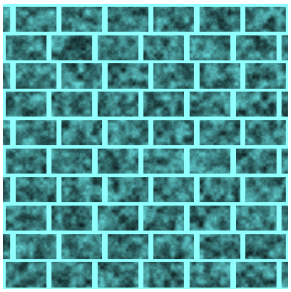
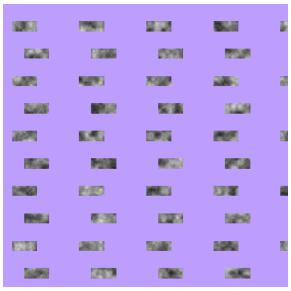


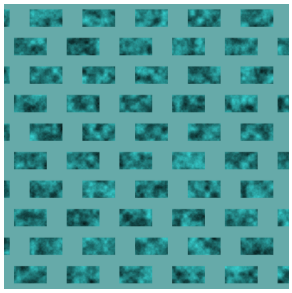
Figure 5.15: Results of evaluating parameter estimation of material M_2 using Squared Bin loss. The runs with target T_{21} are plotted as solid lines while runs with target T_{22} are plotted with dashed lines.



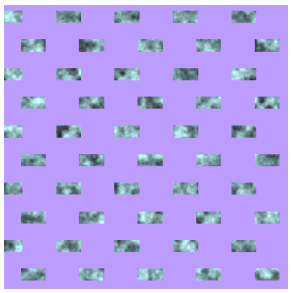
(a) Target T_{21} using Adam.



(b) Target T_{22} using Adam.



(c) Target T_{21} using RMSprop.



(d) Target T_{22} using RMSprop.

Figure 5.16: The four rendered textures at the point of minimal loss for M_2 using SBL corresponding to the six plots in Figure 5.15 with combinations of optimizers Adam or RMSprop and targets T_{21} or T_{32}.

Advanced Brick Test Shader M_3

For parameter estimation using SBL on shader M_3, similar settings to M_2 were used and the bin size was kept at 8. The results are shown in Figure 5.17 and the best loss renders in Figure 5.18. For both the two-parameter target T_{31} and the real life target T_{33} the final loss is relatively low at 0.001 and 0.0026 respectively for both optimizers. However, this is fairly easily achievable due to the targets having an obvious and uniform color scheme, which was restored at the expense of the bricks themselves disappearing, as the optimizer could reach a low loss value simply by assigning each pixel the average color for that bin. The results for the random target T_{32} are not very satisfactory and poses a very difficult minimization problem due to the amount of noise in the image.

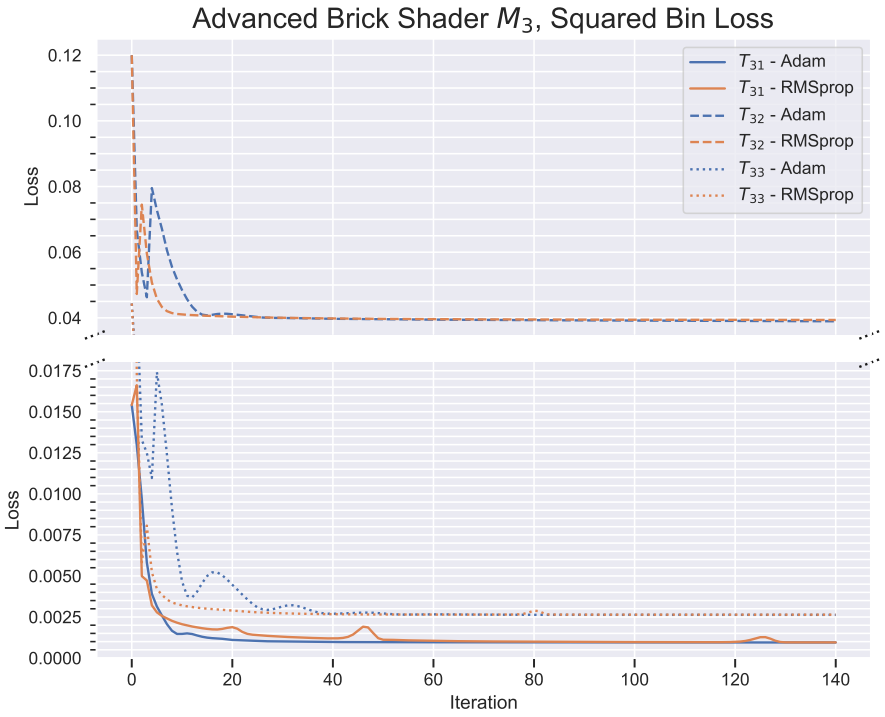
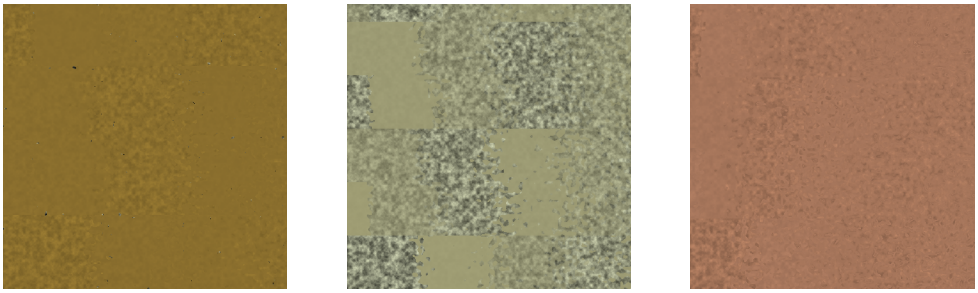


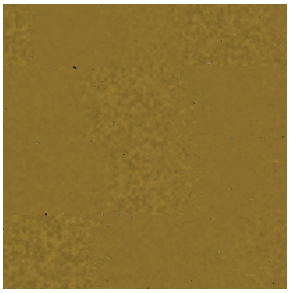
Figure 5.17: Results of evaluating parameter estimation of material M_3 using Squared Bin loss. The runs with target T_{31} are plotted as solid lines, target T_{32} with dashed lines and the additional real-life target T_{33} with dotted lines.



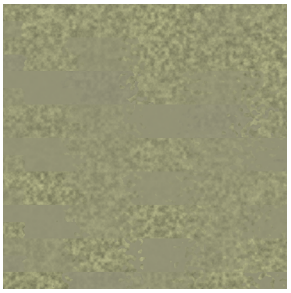
(a) Target T_{31} using Adam.

(b) Target T_{32} using Adam.

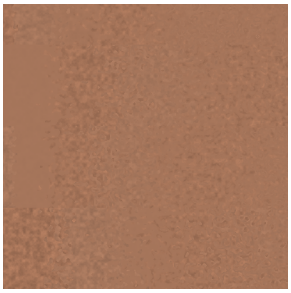
(c) Target T_{33} using Adam.



(d) Target T_{31} using RMSprop.



(e) Target T_{32} using RMSprop.



(f) Target T_{33} using RMSprop.

Figure 5.18: The six rendered textures at the point of minimal loss for M_3 using SBL corresponding to the six plots in Figure 5.17 with combinations of optimizers Adam or RMSprop and targets T_{31}, T_{32} or T_{33}.

5.3.4 Parameter Estimation using Neural Loss

				Adam		*RMSprop*		*Neural Loss*	
M_i	T_{ij}	**Optimizer**	**lr**	β_1	β_2	α	**Momentum**	**layers**	**weights**
M_1	T_{11}	Adam	0.15	0.5	0.999			[0]	[1.0]
M_1	T_{11}	RMSprop	0.09			0.9	0.0	[0]	[1.0]
M_1	T_{12}	Adam	0.05	0.7	0.999			[0]	[1.0]
M_1	T_{12}	RMSprop	0.05			0.9	0.0	[0]	[1.0]
M_2	T_{21}	Adam	0.04	0.9	0.999			[0, 4]	[1.0, 4.0]
M_2	T_{21}	RMSprop	0.02			0.99	0.4	[0]	[1.0]
M_2	T_{22}	Adam	0.04	0.9	0.999			[0]	[1.0]
M_2	T_{22}	RMSprop	0.02			0.95	0.4	[0]	[1.0]
M_3	T_{31}	Adam	0.01	0.9	0.999			[5]	[1.0]
M_3	T_{31}	RMSprop	0.01			0.99	0.0	[4]	[1.0]
M_3	T_{32}	Adam	0.02	0.9	0.999			[0]	[1.0]
M_3	T_{32}	RMSprop	0.02			0.99	0.4	[0]	[1.0]
M_3	T_{33}	Adam	0.005	0.85	0.999			[0, 2, 4]	[1.0, 4.0, 4.0]
M_3	T_{33}	RMSprop	0.003			0.95	0.3	[0, 2]	[1.0, 4.0]

Table 5.3: Optimizer and loss settings when running gradient descent using Neural loss.

The neural loss can be controlled via two parameters: `layers` which is used to specify layer indices to extract features from and `weights` which is a list of weights for each selected layer as explained in section 2.3.2. In a Neural Network, specific features are picked up by the early layers close to the input while general features are found by layers closer to the output. This means that we want to extract features from the early layers so as to not capture features that are too generic and experiments showed that features extracted from layers 0, 2, 4 and 5 generally give the most accurate results. Unlike MSE loss or SBL, the range of possible loss values greatly depends on choice of layers and weights, which makes it more difficult to judge whether the returned loss value is an objectively good result. In these experiments, we will therefore not set a threshold (except for $\boldsymbol{M}_1$), but instead run gradient descent for 400 iterations and compare the initial loss to the minimum loss as well as subjectively judging the resulting rendered textures. The optimizer and loss settings used in each experiment with a neural loss is displayed in table 5.3.

HSV Test Shader M_1

Using a neural loss for uniform images of a single color without any patterns is not only considerably slower than using a simple MSE or SB loss, but did in this example yield less accurate results. For all runs with shader $\boldsymbol{M}_1$ we only extract features from the first layer and use a weight of $\mathbf{1.0}$. For the easier target $\boldsymbol{T}_{11}$ we could keep the learning rate relatively high, but had to lower it for target $\boldsymbol{T}_{12}$ to prevent oscillation. The results are shown in Figure 5.19 where the loss threshold was set to $\mathbf{0.05}$, an arbitrary value judged to be low enough considering the high initial loss of almost $\mathbf{7.0}$. Yet again, RMSprop outperforms Adam for both targets and reaches the threshold two or three iterations earlier. The best loss renders are shown in Figure 5.20, where we see that target $\boldsymbol{T}_{11}$ was successfully reproduced with a very high similarity. For target $\boldsymbol{T}_{12}$ however, the same loss threshold did not result in an equally accurate render, where the brown color is too light. This shows how the neural loss value is not as reliable as in the case of MSE loss or SBL and is not the best loss function to use with simple shaders like $\boldsymbol{M}_1$ lacking patterns, being both slower and less accurate.

Simple Brick Shader M_2

The results of running gradient descent on test shader $\boldsymbol{M}_2$ using a neural loss function are shown in Figure 5.21 where the initial loss when running parameter estimation using Adam with target $\boldsymbol{T}_{21}$ is relatively high. As shown in table 5.3, we used a different set of layers to extract features from when running with target $\boldsymbol{T}_{21}$ using Adam, which is reflected in Figure 5.21, as it yielded much better results. It is interesting that the single layer 0 works well for RMSprop, but gave very poor results with Adam. This is probably due to RMSprop by chance finding a better path to a minimum and we could have probably reached the same results with Adam had we explored a much wider range of optimizer settings. Using Adam and target $\boldsymbol{T}_{21}$ we were able to reach a minimum loss of $\mathbf{0.07}$ at iteration 341 and with RMSprop a minimum loss of $\mathbf{0.004}$ at iteration 334. The renders at these points of minimal loss are shown in subfigures 5.22a and 5.22c respectively where a slightly better result was achieved by using RMSprop but is surprisingly not much better than using a simple MSE loss. The color was retrieved fairly well but the elongation is only about 60% of the target. The real advantage of using a neural loss is instead reflected in the cases with the random target $\boldsymbol{T}_{22}$ where the initial

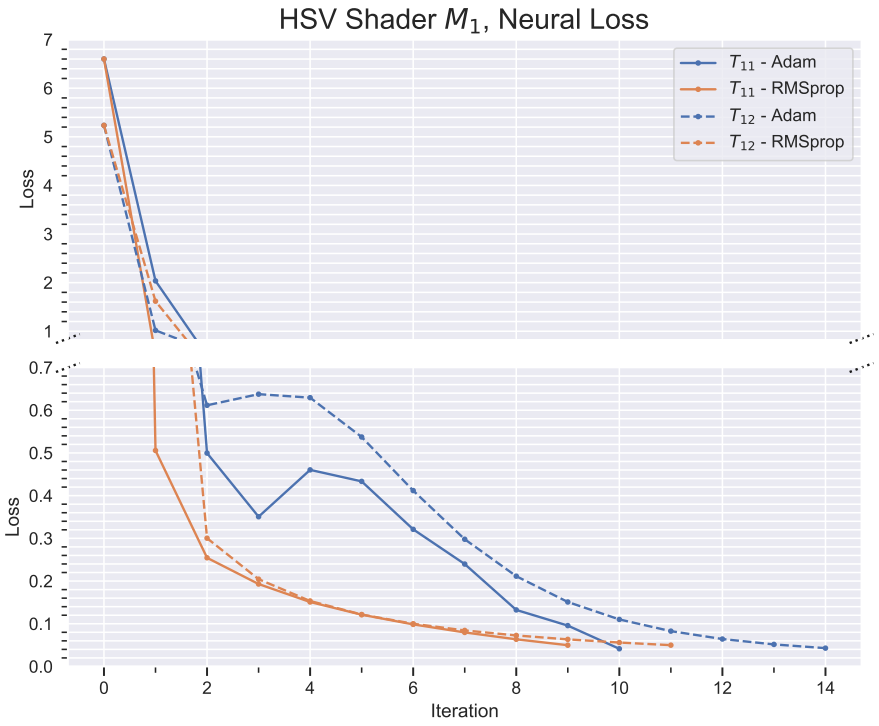
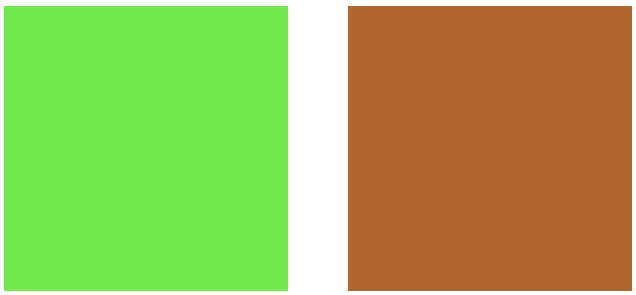
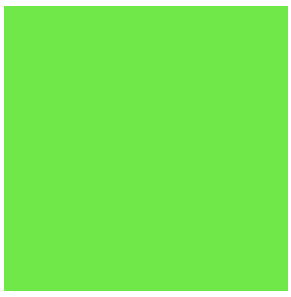


Figure 5.19: Results of evaluating parameter estimation of material M_1 using Neural loss. The runs with target T_{11} are plotted as solid lines and target T_{12} with dashed lines.



(a) Target T_{11} using Adam.

(b) Target T_{12} using Adam.



(c) Target T_{11} using RMSprop.

(d) Target T_{12} using RMSprop.

Figure 5.20: The four rendered textures at the point of minimal loss for M_1 using Neural loss corresponding to the four plots in Figure 5.19 with combinations of optimizers Adam or RMSprop and targets T_{11} or T_{12}.

loss starts at around **0.3** and is minimized down to **0.02** for Adam at iteration 381 and **0.036** for RMSprop at iteration 21. While RMSprop is clearly much faster to reach convergence, the results are better for Adam as seen in subfigures 5.22d and 5.22b respectively. As opposed to using a MSE loss or SBL the shape of the bricks were actually retained and even elongated when using Adam, although not nearly as much as in the target. Clearly a neural loss is a better choice for pattern heavy images, especially when optimizing a large number of parameters, although it is still difficult to reproduce the exact shape.

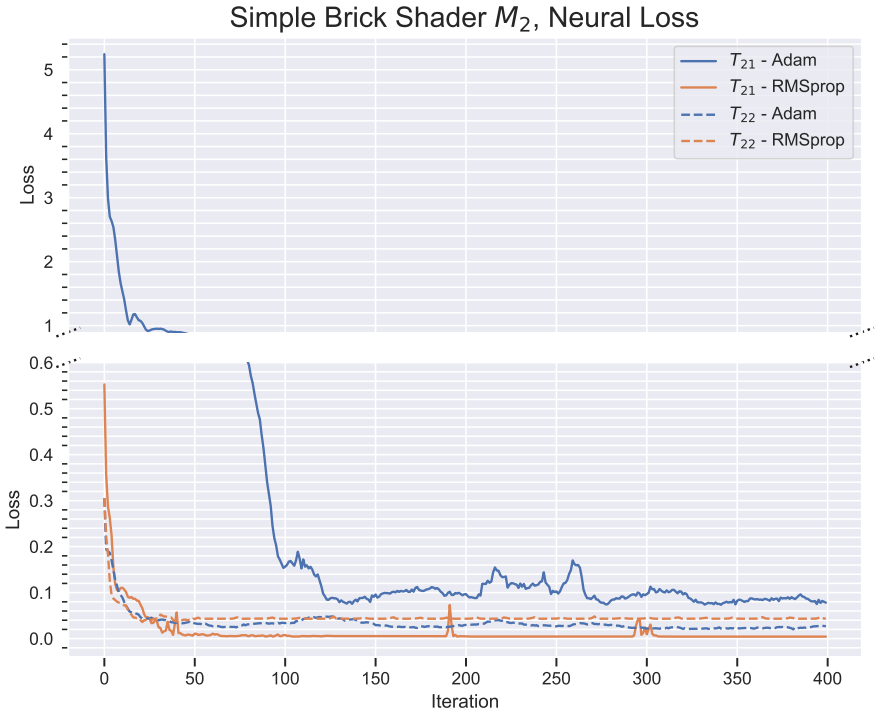
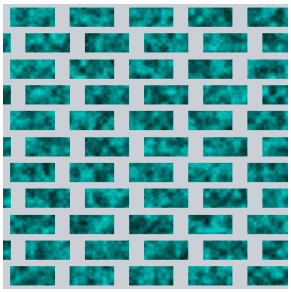


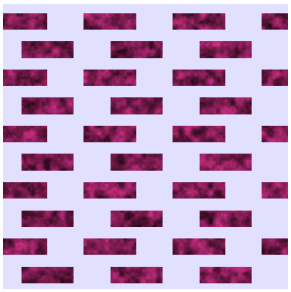
Figure 5.21: Results of evaluating parameter estimation of material $\boldsymbol{M_2}$ using Neural loss. The runs with target $\boldsymbol{T_{21}}$ are plotted as solid lines while runs with target $\boldsymbol{T_{22}}$ are plotted with dashed lines.

Advanced Brick Test Shader M_3

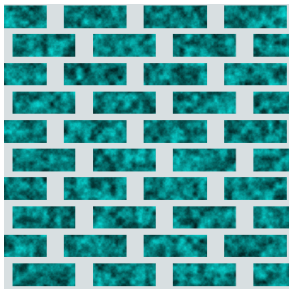
Finally, we evaluate parameter estimation on $\boldsymbol{M_3}$ using a neural loss function where the loss data is shown in Figure 5.23. As seen in table 5.3, this case necessitated the use of a range of different settings for the neural loss function. Interestingly enough, there were big differences in performances between the two optimizers depending on which layers were used. Most notably for target $\boldsymbol{T_{31}}$, where the same learning rate was used for both optimizers, but we only used layer **4** for RMSprop and only layer **5** for Adam as the reverse resulted in both optimizers performing much worse. The loss per iteration for $\boldsymbol{T_{31}}$ is plotted using solid lines, and we can recognize a familiar pattern: RMSprop converges much earlier than Adam with a minimum loss of **0.0014** at iteration 164 while Adam reaches a minimum loss of **0.009** at iteration 397. Note that we can not directly compare the final loss values as we use different layers but the final renders shown in subfigures 5.24a and 5.24d respectively makes it clear that the result for Adam is superior. Both optimizers successfully retain the overall color



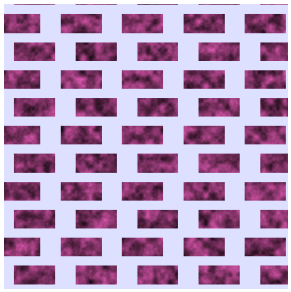
(a) Target T_{21} using Adam.



(b) Target T_{22} using Adam.



(c) Target T_{21} using RMSprop.



(d) Target T_{22} using RMSprop.

Figure 5.22: The four rendered textures at the point of minimal loss for M_2 using Neural corresponding to the six plots in Figure 5.21 with combinations of optimizers Adam or RMSprop and targets T_{21} or T_{32}.

of the target, but only Adam manages to correctly retain the scale, albeit at the expense of correct brick elongation.

The loss data for the random target T_{32} is plotted using dashed lines where both optimizers experience a fairly unstable few initial iterations. However, in this case RMSprop not only converges much earlier at iteration 192 versus Adam that does not converge at all, the minimum loss for RMSprop is much lower at 0.05 against 0.13 for Adam where both used layer 0 with the same weights for the neural loss. Judging the resulting renders in subfigures 5.24b for Adam and 5.24e for RMSprop however, its difficult to see much similarity between either and the target.

The last and perhaps most interesting target T_{33} is represented as dotted line plots where the additional layer 4 was used for Adam which did not seem to provide any benefits for RMSprop. Both manage to quite successfully minimize the loss where Adam reached a minimum value of 0.2 at iteration 363 and RMS a minimum value of 0.19 at iteration 340. Furthermore, Adam experienced a much smoother progression whereas RMSprop saw several large spikes which might be preventable with better tuned learning rate and α parameters. The rendered results are shown in subfigures 5.24c and 5.24f, proving that the overall color has been very well reproduced in both cases, even the color of the mortar. What differentiates both cases is the scale of the bricks, where Adam managed to achieve better similarity, but again, both results show too elongated bricks. Furthermore, the noise function influencing the color of the bricks has not been changed much, although that could be the fault of the shader model composition itself. Ultimately, using a real life texture worked well but required a fair amount of tuning of optimizer and loss parameters.

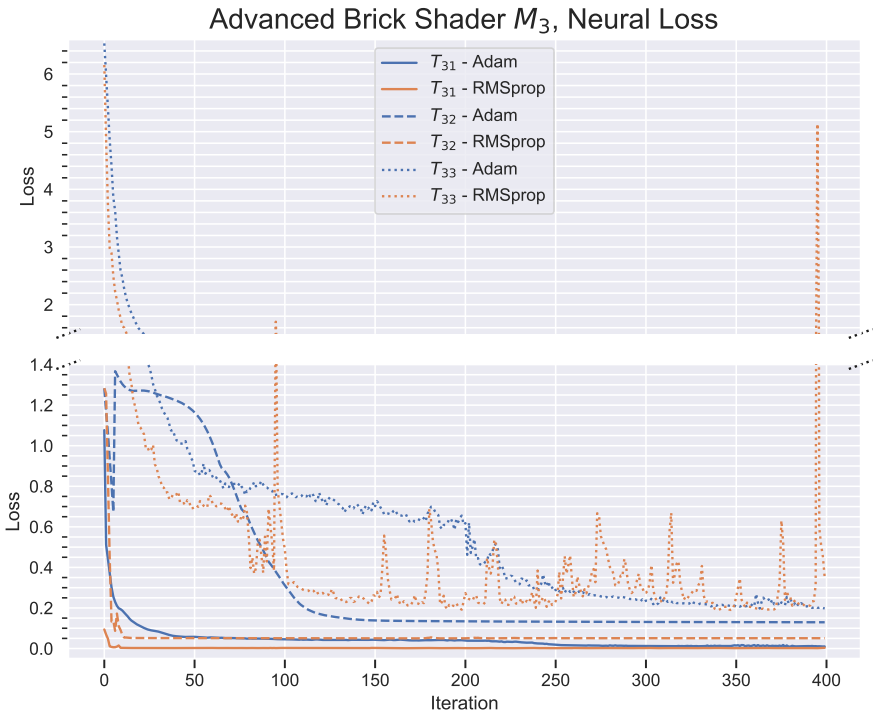


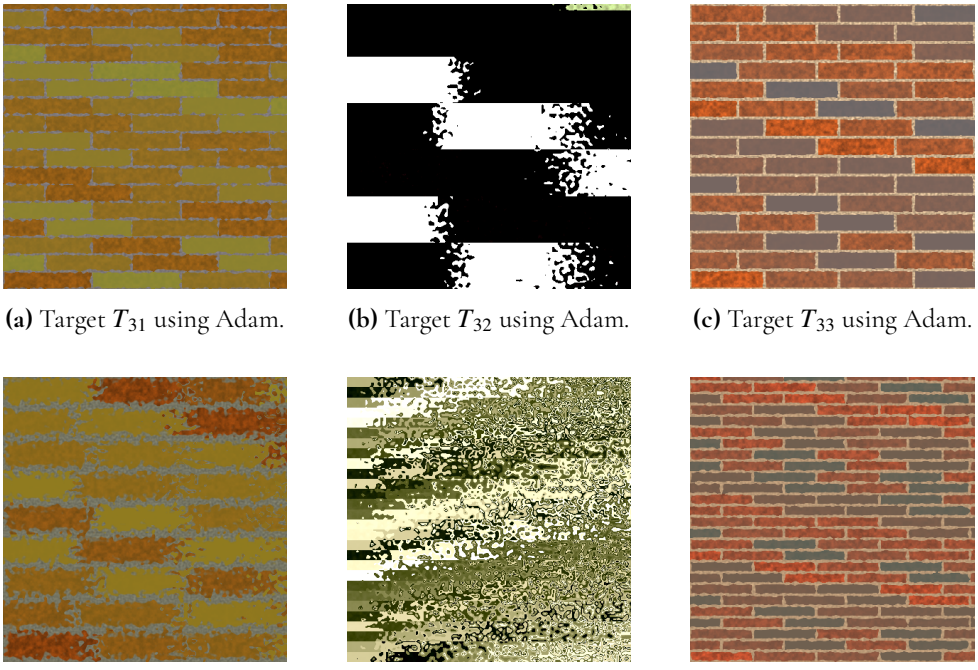
Figure 5.23: Results of evaluating parameter estimation of material M_3 using Neural loss. The runs with target T_{31} are plotted as solid lines, target T_{32} with dashed lines and the additional real-life target T_{33} with dotted lines.

Overall, the results are much better than when using MSE loss or SBL and except for target T_{32}, the overall color was very well retained, and when using Adam even the scale of the bricks was fairly well restored, although the elongation of the bricks remains a difficult problem to solve for both optimizers. For this case, Adam is the superior optimizer, although only marginally.

5.3.5 Finding a "Correct" Minimum

We have demonstrated that using gradient descent and a differentiable rendering system we can easily minimize a loss function applied to a static bitmap target and a generated image. However, there are no guarantees that the minimum found is a global minimum, or even that a low loss value corresponds to a high image similarity. In the end, the similarity is judged by a human user with a subjective idea of what constitutes similar images. Using a neural loss function is a big step in the right direction but in order to make judgements for other kinds of shaders, further evaluation is needed.

We impose practical limits on parameter values when controlled by the user, which can introduce problems during parameter estimation. For example, the hue, value and saturation parameters of the HSV shader are all limited to the interval $[0, 1]$. The reason being that the hue parameter is cyclical meaning a value of 1.5 is equivalent to 0.5 while the saturation and value parameters are effectively capped to the interval so that a negative value is equivalent to 0 and any value > 1 is equivalent to 1. PyTorch tensors do not support such limits and



(a) Target T_{31} using Adam.

(b) Target T_{32} using Adam.

(c) Target T_{33} using Adam.

(d) Target T_{31} using RMSprop.

(e) Target T_{32} using RMSprop.

(f) Target T_{33} using RMSprop.

Figure 5.24: The six rendered textures at the point of minimal loss for M_3 using Neural loss corresponding to the six plots in Figure 5.23 with combinations of optimizers Adam or RMSprop and targets T_{31}, T_{32} or T_{33}.

neither do the optimizers. It would be possible to develop a custom optimizer that clamps parameters to their intervals, but this will inevitably have a negative impact on the optimization as a whole. Figure 5.25 shows the loss surface generated by evaluating the MSE loss of shader M_1 over a range of hue and saturation values using target T_{11}, as well as the gradient descent progress. A magenta plus symbol marks the minimum loss value and markers along the gradient descent route indicates each iteration. This proves the cyclic behaviour of the hue value as the loss is repeated along the x-axis. This means that we have one minimum every `hue=0.3,1.3,2.3` etc and because the initial value of the hue is 1.0, the closest minimum for the optimizer to seek is the one where `hue=1.3`, which unfortunately is outside of our limits, and simply clamping this value would result in a worse loss with `hue=1.0`.

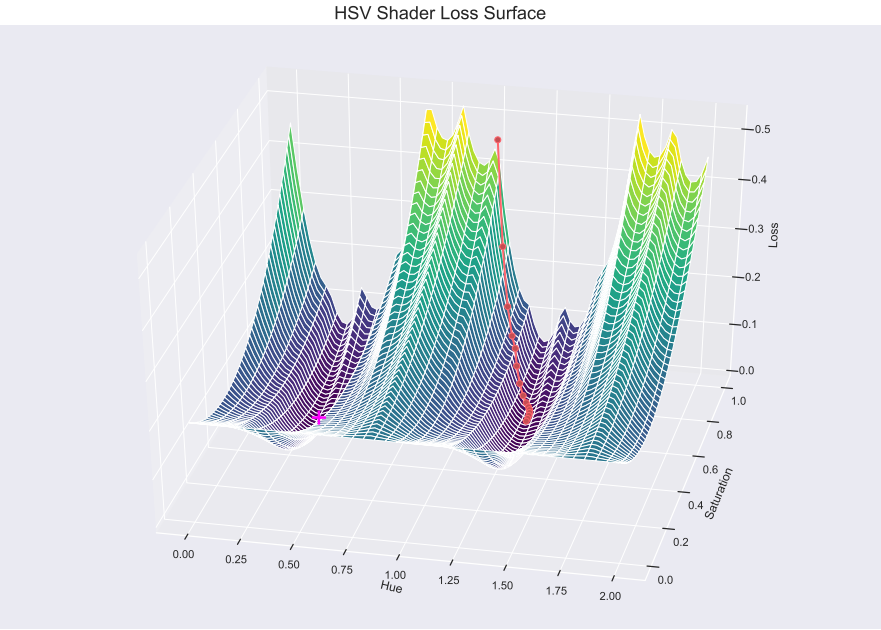


Figure 5.25: Loss surface generated by evaluating an MSE loss between shader M_1 and target T_{11} for a range of different `hue` and `value` parameter values. A magenta "plus" symbol denotes the point of minimum loss and the red line marks the gradient descent progress.

Chapter 6
Conclusion

In this thesis we developed a specialized framework for both differentiable procedural texture rendering using PyTorch in Python and conventional procedural texture rendering using OpenGL. The back-end PyTorch renderer was implemented to mimic the results of the user-facing OpenGL renderer so that results from parameter estimation using gradient descent on our back-end differentiable renderer would be valid and correct in OpenGL as well. Our framework supports dynamic creation of procedural texture models at runtime using a graphical node editor interface. As OpenGL does not support shader composition, we implemented a GLSL code parser that breaks down the source code into Python objects that can be reassembled and recompiled at runtime as the user designs a procedural texture model. Finally, we designed three different test models to evaluate our system and several textures for each model to serve as targets for parameter estimation. Each shader was evaluated with a combination of different loss functions, optimizers and target textures.

We found that by using an automatic differentiation framework like PyTorch it was possible to implement a fully differentiable texture renderer. However, as PyTorch is not optimized for computer graphics it was only usable for real time rendering for very low resolutions when using more complex texture models. Real time rendering performance was never needed though, as we could restrict the use of the differentiable renderer to parameter estimation by mimicking the rendering process in OpenGL to a high degree by adapting our Python syntax to that of GLSL.

Through evaluation we learned that our system could, in many cases, find a more optimal parameter set that faithfully restored colors and scale but not elongation of our brick textures. As noise is innately very volatile it was also difficult to optimize and we rarely saw any change in noise parameters during parameter estimation. For simple optimization problems DiPTeR required fewer than 10 iterations, taking only a few seconds in total to execute. However, for more complex models, especially when using a neural loss, a few hundred iterations were needed where each iteration took over a second to perform. This is comparable to the results of Guo et. al. where a model with 23 parameters (similar to our M_3) is optimized, also using PyTorch, in 290 seconds over 1000 iterations, although they tested on much newer hardware

and could thus utilize GPU acceleration [7].

We evaluated two optimizers, Adam and RMSprop, which in many cases performed comparatively well although Adam tended to find a lower loss while RMSprop tended to optimize more efficiently. Overall, RMSprop performed slightly better for simpler shader models and loss functions while using Adam proved advantageous when using the neural loss function on targets with a significant amount of noise. Using the neural loss function proved a necessity for our most complex shader model $\boldsymbol{M}_3$ and overall produced the most accurate results, but required a significant amount of experimentation with its settings to work well. Ultimately, using gradient descent to estimate parameters in a differentiable rendering framework proved a flexible and fairly simple method for dynamically created procedural textures. The biggest drawbacks being that it needs a custom built rendering framework and the accuracy of the result relies heavily on a correctly tuned loss function and could benefit from an implementation more optimized for computer graphics.

6.1 Future Work

Our differentiable renderer is a highly specialized and simplified version of OpenGL, developed exclusively for 2D rendering of procedural textures. As explained, it is not optimized for computer graphics and although our rendering evaluation is performed on fairly old hardware without GPU acceleration support, thus missing out on a significant amount of performance potential, even better results can be obtained by using a more optimized tool. The team behind PyTorch are currently working on a dedicated general differentiable renderer named PyTorch3D which should bring useful features and a needed performance boost if implemented in our project [6]. If such a system performs well enough, exclusively using it for rendering would eliminate any problems with discrepancies between our two current systems but would also forego the exportability that comes with using OpenGL shaders.

In section 5.3 we evaluated three different shader models on a sizeable number of combinations of loss functions, optimizers and targets. However, we only had enough time to implement and test shaders that produce brick textures of varying complexity which have a rather specific uniform pattern. In the future, more types of shaders should be tested with different and more irregular patterns. We also found that while the neural loss function gave the best results in most cases, it requires a good amount of adjustment of its `layers` and `weights` parameters to perform well. A look into alternative loss functions or more research on neural loss functions is needed.

References

[1] B. Foundation, "blender.org - home of the blender project - free and open 3d creation software," blender.org, 2020. [Online]. Available: https://www.blender.org/

[2] "Substance designer | substance 3d," Substance 3D, 05 2020. [Online]. Available: https://www.substance3d.com/products/substance-designer/

[3] *TileGAN*, vol. 38. ACM Transactions on Graphics, 07 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3306346.3322993

[4] L. , "Comparison of badly uv-unwrapped bitmap texture and a procedural texture," Blender Market. [Online]. Available: https://blendermarket.com/products/carvature

[5] *OpenDR: An Approximate Differentiable Renderer*, vol. 8695. Computer Vision – ECCV 2014, 2014. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-10584-0_11

[6] facebookresearch, "facebookresearch/pytorch3d," GitHub, 05 2020. [Online]. Available: https://github.com/facebookresearch/pytorch3d

[7] Y. Guo, M. Hasan, L. Yan, and S. Zhao, "A bayesian inference framework for procedural material parameter estimation," arXiv.org, 2019. [Online]. Available: https://arxiv.org/abs/1912.01067

[8] J. CREST, "Jst crest | hci for machine learning project," HCI for Machine Learning, 2020. [Online]. Available: https://www-ui.is.s.u-tokyo.ac.jp/crest/index.html

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, ser. Advances in Neural Information Processing Systems 32. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[10] N. P. Rougier, "Glumpy: fast, scalable & beautiful scientific visualization," Glumpy, 2011. [Online]. Available: http://glumpy.github.io/

[11] Riverbankcomputing, "What is pyqt?" Riverbankcomputing.com, 2020. [Online]. Available: https://www.riverbankcomputing.com/software/pyqt/

[12] Y. Hu, J. Dorsey, and H. Rushmeier, "A novel framework for inverse procedural texture modeling," *ACM Trans. Graph.*, vol. 38, no. 6, 11 2019. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/3355089.3356516

[13] B. Guenter, J. Rapp, and M. Finch, "Symbolic differentiation in gpu shaders," Microsoft Research, 03 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/symbolic-differentiation-in-gpu-shaders/

[14] *Learned Multi-View Texture Super-Resolution*. 2019 International Conference on 3D Vision (3DV), 09 2019.

[15] B. Li, Y. Shi, S. Li, B. Wang, Z. Qi, and J. Liu, "A novel texture generation super resolution model," *Procedia Computer Science*, vol. 162, pp. 924 – 931, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050919320800

[16] K. Perlin, "An image synthesizer," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 287–296, 07 1985. [Online]. Available: https://doi.org/10.1145/325165.325247

[17] A. C. Faul, *A Concise Introduction to Machine Learning*. Crc Press, Taylor & Francis Group, 2020.

[18] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html

[19] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012. [Online]. Available: http://arxiv.org/abs/1212.5701

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv.org, 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[21] Y. Bengio and Y. LeCun, Eds., *Very deep convolutional networks for large-scale image recognition*. 3rd International Conference on Learning Representations, ICLR 2015, 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[22] *Texture synthesis using convolutional neural networks*, NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems. MIT Press, 2015.

[23] *Deep Convolutional Inverse Graphics Network*, vol. 28. Neural Information Processing Systems, 2015. [Online]. Available: http://papers.nips.cc/paper/5851-deep-convolutional-inverse-graphics-network

[24] *3D Pose Regression Using Convolutional Neural Networks*. 2017 IEEE International Conference on Computer Vision Workshops (ICCVW), 10 2017. [Online]. Available: https://ieeexplore-ieee-org.ludwig.lub.lu.se/document/8265464?arnumber=8265464

[25] *Face-to-Parameter Translation for Game Character Auto-Creation.* 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 10 2019. [Online]. Available: https://ieeexplore-ieee-org.ludwig.lub.lu.se/document/9010862?arnumber=9010862

[26] V. Deschaintre, M. Aittala, F. Durand, G. Drettakis, and A. Bousseau, "Single-image svbrdf capture with a rendering-aware deep network," *ACM Trans. Graph.*, vol. 37, no. 4, 07 2018. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/3197517.3201378

[27] K. Kang, Z. Chen, J. Wang, K. Zhou, and H. Wu, "Efficient reflectance capture using an autoencoder," *ACM Trans. Graph.*, vol. 37, no. 4, 07 2018. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/3197517.3201279

[28] O. Nalbach, E. Arabadzhiyska, D. Mehta, H. Seidel, and T. Ritschel, "Deep shading: Convolutional neural networks for screen space shading." *Computer Graphics Forum*, vol. 36, no. 4, pp. 65 – 78, 2017. [Online]. Available: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=123928902&site=eds-live&scope=site

[29] M. Aittala, T. Aila, and J. Lehtinen, "Reflectance modeling by neural texture synthesis," *ACM Trans. Graph.*, vol. 35, no. 4, 07 2016. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/2897824.2925917

[30] L.-Y. Wei, J. Han, K. Zhou, H. Bao, B. Guo, and H.-Y. Shum, "Inverse texture synthesis," *ACM Trans. Graph.*, vol. 27, no. 3, p. 1–9, 08 2008. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/1360612.1360651

[31] V. Kwatra, I. Essa, A. Bobick, and N. Kwatra, "Texture optimization for example-based synthesis," *ACM Trans. Graph.*, vol. 24, no. 3, p. 795–802, 07 2005. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/1073204.1073263

[32] *Image quilting for texture synthesis and transfer*, Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques. Association for Computing Machinery, 2001. [Online]. Available: https://doi.org/10.1145/383259.383296

[33] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne, "Jax: composable transformations of python+numpy programs," 2018. [Online]. Available: http://github.com/google/jax

[34] M. Segal and K. Akeley, "The opengl® graphics system: A specification," 02 2013. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/glspec43.core.pdf

[35] T. Q. Company, "Qt | cross-platform software development for embedded & desktop," www.qt.io. [Online]. Available: https://www.qt.io/

[36] "Rendering pipeline overview - opengl wiki," OpenGL Wiki, 2019. [Online]. Available: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

Appendices

Appendix A
Links

A.1 Example shader "Strawberry Pattern"

A live version of the example shader in section 2.1 can be viewed at *Shadertoy* at the following URL: `https://www.shadertoy.com/view/tlsyDS`

A.2 DiPTeR Repository

The code repository of DiPTeR can be viewed on GitHub here: `https://github.com/Zorobay/DiPTeR`