TU WIEN Informatics

Inverse Material Synthesis via Sub-Shader Extraction

A Neural Shader Parametrisation Approach With Subsequent Shader Simplification

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Informatik

eingereicht von

Marcel Arthur Winklmüller
Matrikelnummer 01429490

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Projektass. Mag. Martin Ilcik

Wien, 15. März 2025

Marcel Arthur Winklmüller	Martin Ilcik

TU WIEN Informatics

Inverse Material Synthesis via Sub-Shader Extraction

A Neural Shader Parametrisation Approach With Subsequent Shader Simplification

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Informatics

by

Marcel Arthur Winklmüller
Registration Number 01429490

to the Faculty of Informatics

at the TU Wien

Advisor: Projektass. Mag. Martin Ilcik

Vienna, 15th March, 2025

Marcel Arthur Winklmüller	Martin Ilcik

Erklärung zur Verfassung der Arbeit

Marcel Arthur Winklmüller

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. März 2025

Marcel Arthur Winklmüller

Kurzfassung

Ich erforsche die Möglichkeit, prozedurale Blender-Shader für selbstähnliche, semi-reguläre Oberflächen zu generieren, die auf einem Eingabebild basieren, das von einem Foto oder digitalen Rendering abgeleitet ist. Der resultierende Blender Shader soll einen Render eines Materials erzeugen können das dem eingabe Bild ähnelt.

Dies wäre nützlich, da prozedurale Materialien 3D-Künstlern Flexibilität und kreative Kontrolle über das Aussehen der Materialien bieten, die bildbasierte Materialien nicht können, während sie alle Vorteile dieser bildbasierten Materialien, wie Verfügbarkeit und Benutzerfreundlichkeit, erben.

Diese Shader-Erzeugung erfolgt durch Schätzung von Parametern eines gegebenen, vorab erstellten, allgemeinen Shader-Graphen (dem Super-Shader) und anschließender extraktion eines simpleren Untergraphen. Die Parameter werden durch ein MLP-Neuronennetz geschätzt, das Texturstatistikbeschreiber als Eingabe verwendet. Diese Beschreiber werden von einem vortrainierten Bildklassifizierungs-CNN extrahiert, das das ursprüngliche Foto/Rendering als Eingabe nutzt. Mit dieser Pipeline kann ein großer Lern-Datensatz für das MLP künstlich leicht mit Blender und dem Super-Shader generiert werden, indem mit zufälligen Parametern gerendert wird.

Abstract

I explore the possibility of generating procedural Blender shaders of self-similar semi-regular surfaces from an input image derived from photograph or digital render. The resulting Blender shader should produce a Material with a look similar to the input image.

This would be useful because procedural materials give 3D artists flexibility and creative control over the materials look, that image based materials cannot, while inheriting all the benefits of said image based materials, like availability and ease of use.
This shader generation is done by a neural *parameter predictor* and subsequent extraction of a *sub-shader* from a given, pre-built, general shader graph (the *super-shader*). The parameters are predicted by a MLP neural network using statistical texture descriptors as its input. Those descriptors are extracted from a pre-trained image classification CNN, which uses the original photograph/render as its input. Using this pipeline a large learning dataset for the MLP can be easily artificially generated with Blender by rendering the *super-shader* with randomized parameters.

Contents

CHAPTER 1

Introduction

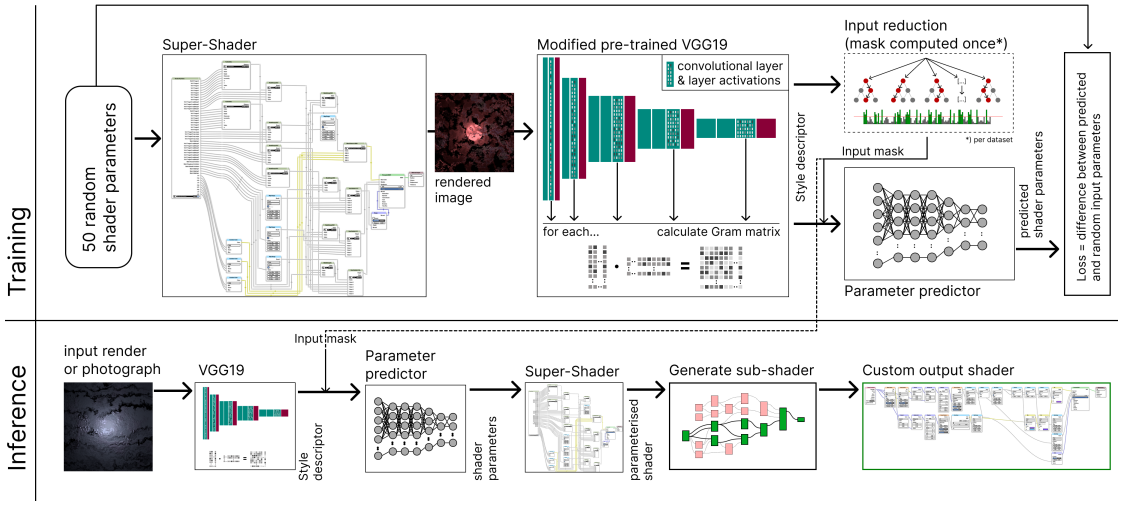


Figure 1.1: Overview of the training (top) and inference (bottom) pipelines for generating custom shader outputs. During training, a *Super-Shader* is assigned 50 random parameters, and rendered to an image, which is then fed into a *modified pre-trained VGG19* to compute style descriptors. An *input mask*, computed via a RFR, reduces the descriptor's dimensionality. The *Parameter Predictor* gets trained to predict shader parameters based on the descriptor, minimizing the difference to the ground-truth input parameters. During inference, a user-supplied image is processed through the same VGG19 and *Parameter Predictor* to estimate the super-shader parameters. A script then extracts a simplified *sub-shader* that reproduces the render with reduced complexity, making it easier for the user/artist to modify.

Texturing plays an essential role in creating photorealistic 3D renderings, defining surface properties such as colour, roughness, shininess, and small surface details which are used by a light transport algorithm [Kaj86; VG97]. In modern workflows, materials are typically described using one of two main approaches: image-based materials and

procedural materials. Each method offers unique advantages but also comes with significant limitations. Table 1.1 and Figure 1.2 provide a comparison of both.

Image-Based Materials: Image-based materials rely on pre-generated texture maps, often created from high-resolution photographs or manually painted in specialized software such as Adobe Substance Painter. In PBR [Kum20] maps describe various surface properties like colour, roughness, and bumpiness, and they are widely available and relatively easy to use. However, their reliance on fixed-size textures introduces challenges. When applied to surfaces larger than the original texture, tiling artifacts become apparent, disrupting visual continuity. Additionally, customization is limited; modifying the material's appearance often requires creating entirely new texture maps, which is time-intensive and demands significant expertise.

Procedural Materials: Procedural materials, in contrast, calculate surface properties dynamically using mathematical noise generators and functions. This approach offers several benefits, including resolution independence, seamless scalability, and significant customizability. Artists can tweak parameters to modify material properties, such as adjusting the coarseness of concrete or adding metallic inclusions, without needing new texture maps. Procedural materials also require less storage space since they do not rely on large image files. However, their creation is complex and time-consuming, demanding advanced skills and a deep understanding of node-based systems.

Challenges with Existing Methods: While image-based materials excel in accessibility and realism, their limitations in scalability and customizability make them less suitable for large surfaces or varied artistic requirements. Procedural materials address these shortcomings but at the cost of accessibility and ease of creation. Realistic procedural materials are particularly challenging to create, even for skilled artists, as shown by their relatively lower adoption compared to image-based methods.

Bridging the Gap: A system that bridges these two approaches could combine their strengths, offering artists the scalability and customization of procedural materials with the ease of use and visual fidelity of image-based materials. Such a hybrid workflow would benefit novice artists by simplifying complex tasks and empower experts with greater artistic freedom. By leveraging the advancements in neural networks, particularly in image analysis and procedural generation, it is possible to create a workflow that inherits the best of both worlds.

In this thesis, I explore the feasibility of using neural networks to generate procedural shaders directly from image-based materials or even real-world photographs. This approach aims to create a unified texturing workflow that combines the strengths of both methods while addressing their limitations.

1.1 Motivation

Image-based materials are abundantly available, easy to use but hard to make, not freely scalable because of tiling artifacts and hardly customizable. Procedural materials are

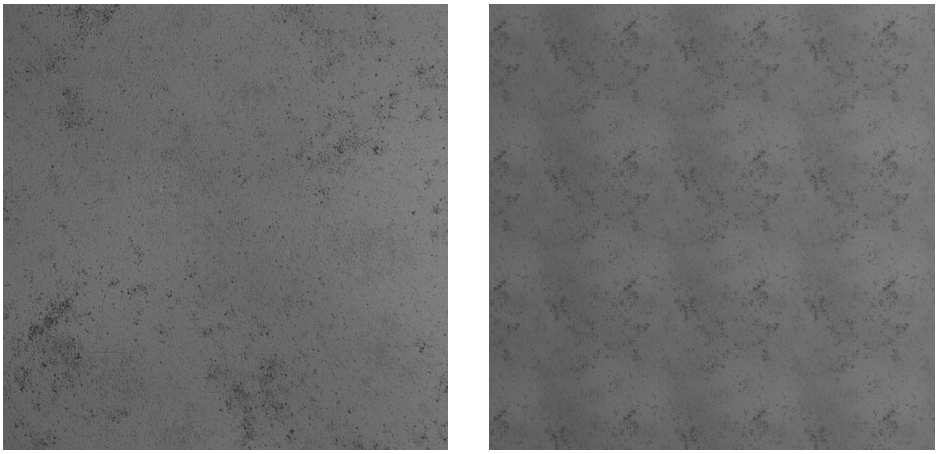
Feature	Image-Based Materials	Procedural Materials
Definition	Use pre-generated images (textures) to define surface properties.	Generate surface properties dynamically using mathematical functions.
Flexibility	Limited to the provided texture; customization requires creating or sourcing new textures.	Highly customizable; properties can be adjusted dynamically.
Resolution Independence	Fixed resolution; scaling leads to tiling or loss of detail.	Fully resolution-independent; no tiling issues.
Ease of Use	Easy to apply pre-existing textures but harder to create from scratch.	Requires expertise to create but offers powerful control once set up.
Storage Requirements	Requires storage for multiple texture maps.	Requires minimal storage as textures are generated on-the-fly.
Realism	Often highly realistic due to reliance on actual images.	Can achieve realism but typically requires more effort to match image-based fidelity.
Performance	May require less computational power during rendering.	Computationally intensive, especially with complex noise functions.

Table 1.1: Comparison of Image-Based and Procedural Materials

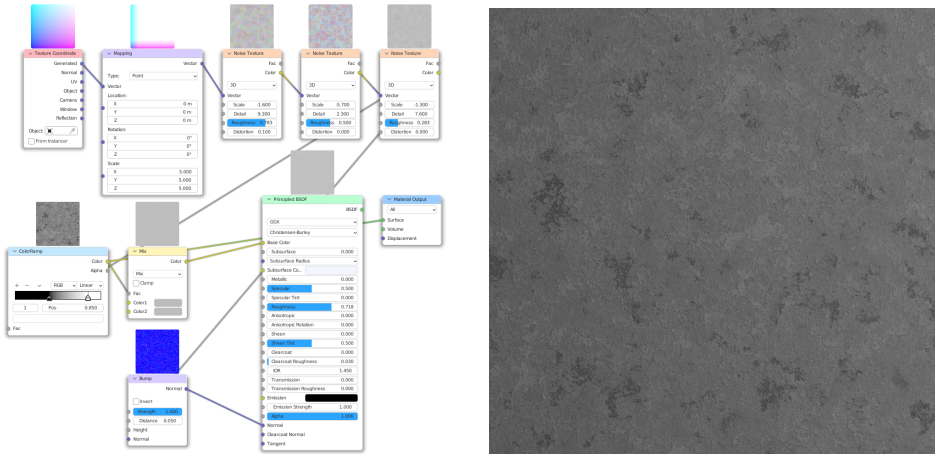
also hard to make, not as common as image-based materials but highly customizable, storage space efficient, and borderless and therefore useable on large surfaces without the danger of running into tiling issues.

In this thesis, I explore the possibility of a system which can generate a procedural shader from an image-based shader, or even from a photograph of a real-world surface. Creating a texturing workflow that inherits the benefits of both methods. Such a system would be valuable for novice as well as expert artists, by speeding up the texturing workflow and increasing artistic freedom.

The focus of this work is to explore the possibility of using neural networks to do this procedural shader generation with minimal user input. An overview of the approach is illustrated in Figure 1.1. A *Super-Shader* with 50 parameters being able to simulate a variety of materials is taken as the underlying model. The main goal is to train a *Parameter Predictor* for estimation of it's parameter values that would produce the input image. Instead of working directly on the image, a *modified pre-trained VGG19* computes a style descriptor based on Gram matrices. Details are discussed in sections 4.2.3 and 4.3. A random forest regressor (RFR) further determines a mask for dimensionality reduction



(a) Photo texture approach



(b) Procedural approach.

Figure 1.2: Comparison between texture based (a) and procedural concrete (b). Both resulting renders (right) use the same scaling. The photo based approach leads to tiling issues because the source texture has to be repeated once the surface size exceeds the source image size.

of the descriptor. Details follow in Section 5.3.1. Various material classes usually require only a subset of parameters, therefore in Section 5.1.2 I also present a simplification utility that given the estimated shader parameters isolates a *sub-shader* that reproduces the render with reduced complexity, making it easier to control.

Having outlined the motivation for bridging image-based and procedural materials, this chapter has set the stage for the investigation by highlighting the challenges and potential benefits of a unified workflow. In the following chapter, I review related work in texture synthesis and material generation, providing the necessary background and context that informs the design of my system.

CHAPTER 2

Related Work

The field of computer graphics and material synthesis has witnessed substantial advancements over the past few decades. In this chapter, I will discuss key contributions in areas that directly inform this work; especially texture synthesis, material synthesis, and shader parameter estimation. The goal of this review is to give context to my approach and identify the strengths and limitations of existing methods.

We begin by taking a look at texture synthesis, where the concept of representing image style through statistical measures; most notably using Gram matrices derived from layer activations, was first introduced. This approach has not only revolutionized style transfer applications but also provided a foundation for extracting rich texture descriptors from images.

Next, I explore the evolution of material synthesis techniques, comparing image-based methods with procedural approaches. As previously detailed in Table 1.1, while image-based materials offer ease-of-use and high realism, they are often limited by issues such as tiling and lack of scalability. Procedural methods, on the other hand, promise greater flexibility and resolution independence, at the cost of increased complexity.

Finally, I look at methods for shader parameter estimation. Recent work in this area explores how to automatically map visual features from images to the parameters used in procedural shaders.

Overall, this chapter sets the stage by showing how earlier research has built the foundation for my work and where there is still room for improvement. It explains the reasoning behind the methods used in my thesis and how my contributions fit into the larger picture.

Figure 2.1: Gatys et al [GEB15a] texture generation using back propagation on white noise image using Gatys style descriptor L2 distance as loss. Top: generated texture. Bottom: source image. Image courtesy of Gatys et al. [GEB15a]

2.1 Texture Synthesis

Texture synthesis refers to the process of creating new textures that share statistical or visual properties with a source texture. Traditional approaches, such as pixel resampling [WL00; EL99] and patch-based methods [EF01], rearrange elements of the source texture to generate new appearances. While effective for many applications, these approaches are inherently limited as they rely on rearranging existing data, making it difficult to produce genuinely new textures.

Parametric approaches that rely on measurement outputs instead of pixel values have been done as early as 1962 [Jul62] and again later inspired by the early mammalian visual system [HB95; PS00; SF95]. While these methods lead to good results in selected categories of textures, Gatys et al. [GEB15a] introduced a parametric approach to texture synthesis, leveraging the power of convolutional neural networks (CNNs). Instead of comparing raw pixel values, their method measures the statistical properties of features extracted by a pre-trained CNN, referred to as the Gatys style descriptor (explained in Section 4.3). This innovation allowed for the generation of visually similar but unique textures by optimizing a white noise image to match the source texture's style descriptor using gradient descent and an L2 loss function. The method demonstrated exceptional results, producing textures that are both realistic and unique. However, it does not address previously mentioned challenges like tiling artifacts or the lack of customizability inherent in fixed-size image-based textures. These limitations motivate the exploration of procedural approaches, as discussed in this work.

While this approach of texture synthesis is only of limited use in the 3D rendering world, Gatys et al. demonstrates that the statistical properties of textures can be effectively captured through Gram matrix computations of CNN activations. This approach is fundamental to this thesis, as it directly describes the method for extracting style descriptors from rendered images used in this thesis. By leveraging a pre-trained VGG19 network, the pipeline abstracts the complex visual characteristics of a material without relying on direct pixel comparisons, thereby mitigating common issues like tiling. This procedure is detailed in Section 4.3, more implementation details can be found in Section 5.2.2.

2.1.1 Unified Neural Noise Generator

Maesumi et.al. proposed in their recent and catchy named work *One Noise To Rule Them All* [Mae+24], a novel way to generate noise textures that could replace or greatly improve the *super-shader* while at the same time simplifying it. Unlike the traditional fixed set of discrete noise functions (such as Perlin, Worley or Voronoi noise), this approach leverages a denoising diffusion probabilistic model (DDPM) to learn a continuous latent space of noise patterns. By doing so, the model can generate a wide range of noise textures and interpolate smoothly between them.

A set of relatively easy to interpret parameters like scale, distortion or noise type parameters are used to condition the transformer network, to guide it in creating desirable outputs. Importantly, these parameter spaces are fully differentiable.

The paper also demonstrates an application of this unified noise model within the context of inverse procedural material design, particularly through its integration with the MATch approach. In this setup, the neural noise generator replaces conventional noise nodes within a procedural material graph. By feeding its interpretable parameters into the optimization process, the model not only enhances the quality of results but also broadens the space of possible materials. Artists are no longer limited to selecting from a discrete set of noise functions; instead, they can explore a continuous spectrum of noise behaviours that better capture subtle variations and transitions.

The differentiable nature of this approach would greatly benefit the *super-shader*. By removing the discreet switching between noise generators that is currently employed by the use of multiplexers, and therefore also the variable parameter mapping; the unified noise generator and its seamless interpolation between noise types proposed by Maesumi et.al., would reduce abstraction significantly and at the same time make large parts of the *super-shader* differentiable.

Altogether, this method presents a promising approach for simplifying the work of the *parameter predictor*; however, since it was not originally intended for integration with the Blender's shader framework, adapting it would likely require some significant and modifications.

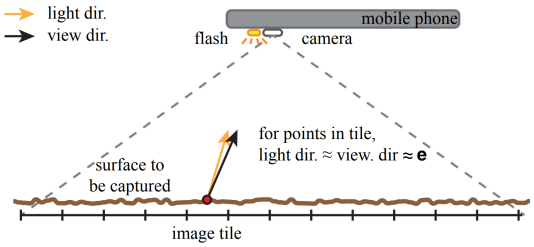


Figure 2.2: Texture sample photo setup. Image courtesy of Aittala et al. [AWL+15]

2.2 Material Synthesis

Material synthesis extends beyond texture generation to capture spatially varying properties like reflectance, bumpiness, and metallicity, often represented as SVBRDF maps. These maps are essential for creating photorealistic materials in physically based rendering (PBR) workflows and are a central topic of this thesis.

Research in this field has explored both image-based and procedural methods, each with their own advantages and limitations. Image-based techniques, including methods like image quilting and PBR map generation, offer ease-of-use and high realism but can suffer from issues like tiling. In contrast, procedural methods provide scalability and customization but are more complex to implement. By automatically generating procedural shaders from image inputs, this thesis seeks to bridge these approaches.

Early work, such as that by Efros and Freeman [EF01], laid the groundwork by demonstrating that textures can be synthesized by rearranging elements of an image. However, these methods do not capture the physical properties needed for complete material generation. Parametric approaches by Aittala et al. [AWL+15] improved upon this by using controlled lighting setups to estimate SVBRDF maps, thereby reconstructing materials with spatially varying properties. More recent methods, including Liang et al.'s MATch system [Shi+20], have taken a neural approach to parameterize procedural material graphs using differentiable shaders and dedicated networks.

Unlike traditional texture synthesis that focuses solely on visual appearance, material synthesis aims to capture both the visual and physical aspects of a material. The insights from these works have directly influenced the design choices in this thesis, particularly in using neural networks for shader parameter prediction within a unified procedural material framework.

2.2.1 Parametric PBR texture synthesis

One approach to overcome tiling issues in materials is to generate texture maps procedurally, while an alternative is to directly produce photo-based PBR texture maps at the desired size. In 2015, Aittala et al. [AWL+15] introduced a non-neural image quilting technique that stitches parts of two input images together to create larger, seamless

texture maps. They evaluated their results by comparing the synthesized textures with ground-truth measurements, such as laser-scanned normal maps and photographs.

In 2016, the same group expanded on this idea in their work "Reflectance Modelling by Neural Texture Synthesis" [AAL16]. This time, they proposed a method to generate SVBRDF parameters from a centrally lit input image. Their approach also employed the Gram style descriptor to quantify the stylistic differences between the input and output images, and they refined the textures iteratively by backpropagating the error through a differentiable pipeline.

2.2.2 Shader Graph Selection and Estimation

A closely related but significantly more intricate approach then the one presented in this thesis has been implemented by Liang et al. in their work on MATch [Shi+20]. This large-scale project involves the use of fully differentiable shader graphs, developed in *Substance Designer*, to represent various types of materials.

For each shader graph, a dedicated neural network is trained to estimate its parameters. The network takes as input the Gatys style descriptor of the target image and outputs parameter values for each shader graph. Once the parameters are estimated, an image is rendered with each shader. The output image's Gatys style descriptor of each shader is then computed and compared to the descriptor of the input image. Among all shaders, the one with the smallest style difference is selected.

Since the shader graphs in MATch are fully differentiable, a gradient-based iterative optimization step can further refine the parameters, enhancing the accuracy of the final rendered image.

The results of this approach are highly impressive, with parametrized shaders producing renders that closely resemble the input images. Additionally, the architecture of MATch offers a significant advantage: each shader has its own parameter prediction network. This reduces abstraction and simplifies the system by allowing new material types to be added easily. Training networks for new materials requires no retraining of the existing shader parametrization networks, enabling modular and scalable extensions of the system. MATch shows that it is able to deliver easily customisable, compact shader graphs, that produce results that resemble the inputs in many cases.

2.2.3 Procedural material synthesis through material decomposition

A recent article by Hu et al. [Hu+22] has the same goal as this thesis – generating procedural materials to eliminate tiling issues and enable artistic customization from texture-based materials. Hu et al. proposed a method which takes PBR textures as input, and produces procedural Adobe Substance painter materials that mimic the input. Their method decomposes the input SVBRDF pixel maps into sub-materials by segmenting and masking, and uses multiple low and high frequency noise generators to mimic those sub-materials. The sub-materials are then recomposed using the previously estimated

masks, and optimised by an iterative rendering based optimisation. As a similarity metric, they use the difference between by VGG generated Gram matrix style descriptors as well.

While the mask generation needs some minimal user interaction, this approach has the grand benefit of not needing any previously crafted shader graphs at all, in comparison to MATch ([Shi+20]) or the proposed method in this thesis. For the same reason it is not limited to the capabilities of the existing source shaders. Materials featuring complex structures or interwoven material differences like; tiles, multicoloured mosaics or brick, are just as possible as simple self-similar materials like leather, concrete, or metals. Although their approach needs SVBRDF maps to start with, there are multiple existing solutions to generate those maps from a centrally lit photograph or render.

2.2.4 Procedural shader parametrisation

Unlike earlier methods that focused mainly on texture synthesis or simple parameter mapping, Guo et al. [Guo+20] present a framework that addresses the entire process of procedural material parametrisation. Their work distinguishes between parameters that are continuous and differentiable versus those that are discrete and non-differentiable. This separation allows for sampling a range of material parameters that can recreate an input image, giving users the option to choose the most suitable parametrisation.

This concept is directly relevant to my thesis. The *super-shader* uses non-differentiable multiplexing parameters, which adds an extra layer of complexity. By adopting a parametrisation strategy similar to that of Guo et al., I could simplify the mapping between style descriptors and shader parameters. In practice, this means I could separate the challenge of handling discrete switching from multiplexing parameters from the continuous mapping of the value parameters, potentially leading to better performance in the *parameter predictor* network and more effective procedural shader generation.

CHAPTER 3

Methodology

In the previous chapters we built an understanding about the difficulties of 3D texturing; On one hand, image-based materials offer high realism but struggle with scalability and customization, often leading to tiling artifacts when applied to large surfaces. On the other hand, procedural materials provide the flexibility and resolution independence needed for creative applications, yet they require complex, time-consuming setups that are not easily accessible to all artists. We also took a look on how the scientific community tackled these issues in the past.

To overcome these limitations, this thesis proposes an automated system that generates procedural Blender shaders directly from image-based inputs by employing a neural network to parametrise a given hand-crafted *super-shader*, from which in a second step, a smaller *sub-shader* is extracted. The goal is to harness the rich, readily available image-based textures and convert them into flexible, easily customisable, procedural materials that retain visual fidelity without the drawbacks of tiling or high storage demands.

Central to this approach is a novel training pipeline that accomplishes two critical tasks:

1. **Dataset Generation:** By randomly sampling a comprehensive set of shader parameters and rendering the corresponding images, the pipeline creates a paired dataset where each sample consists of an image's style descriptor and its ground-truth shader parameters

2. **Parameter Prediction:** A neural network (specifically, a multi-layer perceptron) is trained to map these style descriptors; extracted using a pre-trained VGG19 network and computed via Gram matrices—to the shader parameters. This mapping captures the essential texture statistics needed to recreate the material appearance.

To synthesize a material's appearance without directly copying pixel values, and thereby avoiding tiling and border issues inherent to image-based textures, it is crucial to employ a metric that compares overall image statistics rather than individual pixel values. For this purpose, I use the method described by Gatys et al. [GEB15b], which extracts style information using a pre-trained deep convolutional neural network, VGG19 [SZ14] (detailed in Section 4.2.3). The network's layer activations are used to compute Gatys style descriptors through Gram matrix calculations (see Section 4.3). These descriptors have been successfully applied in style transfer and texture synthesis [GEB15a; GEB16; AAL16; Shi+20], and they form the foundation for both the training and inference pipeline mentioned in this chapter.

In the following sections, the training and inference pipelines, which are shown in Figure 1.1 are detailed. The training pipeline automatically generates a large and diverse dataset and trains a network, while the inference pipeline leverages the trained network to predict shader parameters from new input images and then extracts the relevant *sub-shader*. This unified approach provides a feasible and efficient pathway to generating high-quality procedural materials, laying the groundwork for future enhancements in automated shader creation.

3.1 Training Pipeline

The training pipeline leverages an automated process to generate a large dataset of paired style descriptors and shader parameters. This approach eliminates the need for manual labeling and reduces the computational complexity of training by abstracting the loss calculation away from direct pixel-level or Gram matrix comparisons.

Dataset Generation

The dataset is generated through the following steps:

1. **Random Parameter Generation:** Random parameters (uniformly distributed between 0 and 1) are assigned to the *super-shader*, which is capable of producing a wide variety of materials.

2. **Shader Rendering:** The *super-shader* is rendered to produce an image that simulates a close-up photograph of a material, capturing detailed surface characteristics.

3. **Style Descriptor Extraction:** The rendered image is processed through a modified VGG19 network to compute its Gatys style descriptor. This descriptor, obtained via Gram matrix computations from selected convolutional layers, encapsulates the overall texture and style of the material.

4. **Pair Formation:** Each style descriptor is paired with the corresponding shader parameters used for rendering, forming a single training sample.

MLP Training Pipeline

The MLP is trained to learn the mapping from style descriptors to shader parameters using the following procedure:

1. **Dataset Utilization:** The generated dataset of style descriptor–parameter pairs is employed to train the network. This process allows for a virtually unlimited number of training examples without the need for manual labeling.

2. **Loss Calculation:** Instead of comparing the rendered image directly to an input image using pixel values or Gram matrices, the loss is computed as the difference between the shader parameters estimated by the MLP and the ground truth shader parameters used for rendering. This abstraction reduces computational overhead by removing the need of rendering as part of the training pipeline.

3. **Parameter Prediction Training:** The MLP is optimized using the above loss function to accurately predict shader parameters from the extracted style descriptors. This guides the network to capture the mapping from image statistics to shader configurations despite the high level of abstraction and decoupling from the final rendered output.

Unlike the approach by Gatys et al. [GEB16], where the output of a training step is used to render an image, derive the style descriptors and use the difference of that style descriptor with the the one of the input image, my method computes the loss as the difference between the ground truth shader parameters used for rendering and the parameters predicted by the MLP. This strategy reduces the computational load during training significantly, as no rendering or style descriptor calculation has to be done, at the cost of yet another layer of abstraction, as the compared values are themselves, abstractions of the image.

3.2 Inference Pipeline

During inference, the goal is to generate a procedural material that resembles a user-provided image while avoiding tiling and border issues. The process works as follows:

1. **User Input:** The user supplies an image, such as a close-up photograph of a flat surface (e.g., taken with a flash) or a rendered photo-based texture.

2. **Style Descriptor Extraction:** The input image is processed by the same modified VGG19 network to compute its Gatys style descriptor.

3. **Parameter Prediction:** The computed style descriptor is fed into the trained MLP, which outputs estimates for the 50 shader parameters.

4. **Shader Generation and Simplification:** The estimated parameters are applied to the *super-shader* to generate a parametrised shader.

5. **Sub-shader extraction:** Due to the branching structure introduced by multiplexing, many nodes in the shader may be redundant or inactive after parametrization. To address this, a *sub-shader* extraction script is employed to simplify the parametrised shader, by extracting a *sub-shader* graph, reducing the complex graph from over 500 nodes to a more manageable structure (approximately 30 nodes). This simplification improves usability for artists and enhances rendering efficiency.

By integrating these components, the proposed methodology demonstrates a unified approach to procedural shader generation that leverages style descriptors for parameter prediction. While simpler than methods such as MATch [Shi+20], this approach provides a feasible pathway for creating high-quality, scalable procedural materials within the constraints of a bachelor's project.

In this chapter, I have detailed the comprehensive methodology for generating procedural shaders from input images. Key processes such as style descriptor extraction, shader parametrization, and automated data generation were introduced. The next chapter delves into the technical background underpinning these methods, including an exploration of convolutional neural networks, Gram matrix derivation, and Blender's node-based shader system.

CHAPTER 4

Technical Background

Building on the methodology discussed above, this chapter outlines the technical concepts essential for this work. I will start with giving an overview of Blenders node based shader system to provide a solid foundation to support this thesis's ideas.
The following Section (4.2) gives an overview of convolutional neural networks, focusing on VGG19, a pre-trained CNN used for image classification and instrumental in calculating Gatys style descriptors, which will be detailed right after in Section 4.3.

4.1 Blender Material Node Graph

Like many other 3D modelling applications, blender uses a node based material editor. These editors make it possible for beginners as well as trained artists to create materials with varying complexity relatively easy, without any need for coding skills. There are multiple different types of nodes, I will describe a small subset;

- **Input nodes:** Deliver different values, like surface normal vector, vertex colour, texture coordinates, ambient occlusion values but also more complex information like which type of ray is currently requesting the shading information; shadow ray, camera ray, etc.
- **Value nodes:** Can be used to manually input values into the network like a colour, vector, texture or float.
- **Maths nodes:** Offer a wide array of different calculations for vectors (colours, coordinates, etc) and floats.
- **Noise generator nodes:** Voronoi, Musgrave, White or general noise generation nodes deliver the possibility to generate one or more dimensional noise textures. A wide range of input parameters can be used to alter the output.

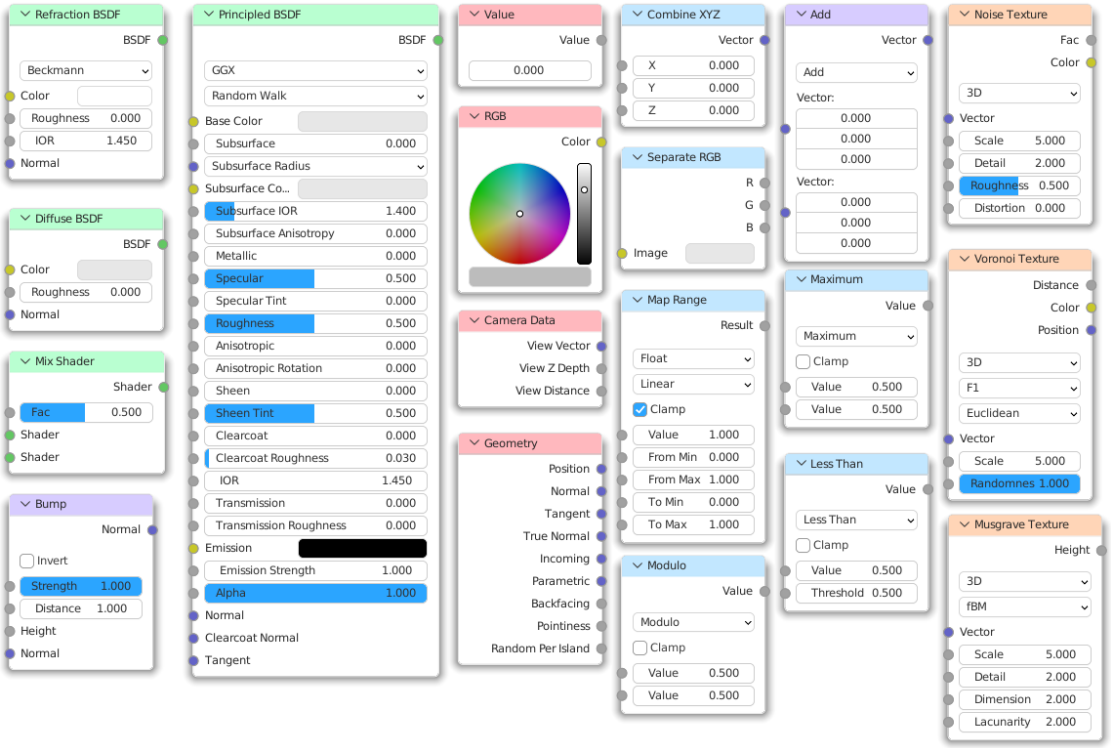


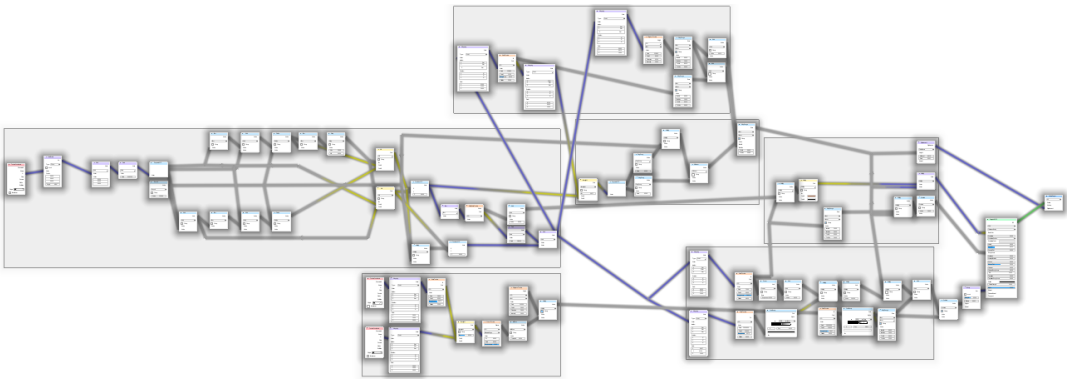
Figure 4.1: Selection of different Blender shader nodes. **Teal:** *Shaders*, **Pink:** *Inputs*, **Blue:** *Numerical maths and converters*, **Purple:** *Vector maths*, **Orange:** *Noise Generators*

- **Group node:** Are used to group nodes together to make the graph more readable.
- **Shader nodes:** Here the previous calculated values are fed into a light transport algorithm to do the actual shading. For most materials the Principled BSDF node can be used

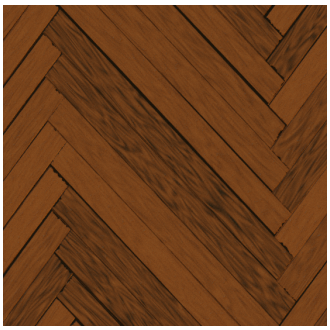
The *super-shader* uses only a small subset of available nodes. Those nodes are used to calculate 4 texture maps that are fed into the *Principled BSDF shader*; *Base colour* (Albedo), *Metallic*, *Roughness* and *Normal*. The Principled BSDF shader output is connected to the surface input of the output node. The other outputs (Volume and Displacement) are not used.
Node inputs and outputs are colour coded to describe the offered or requested type: float, vector, colour and shader. Vector and colour do have different colours but are practically the same and can be used interchangeably.

4.1.1 Procedural Materials

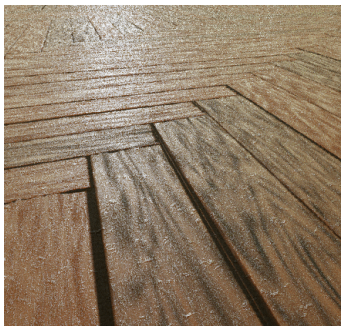
Procedural materials differ from traditional (non-procedural) materials in how texture maps are generated. While non-procedural materials rely on multiple 2D image texture



(a) Complex node setup for a hardwood floor material.



(b) Resulting hardwood floor material.



(c) Specular detail of the material.

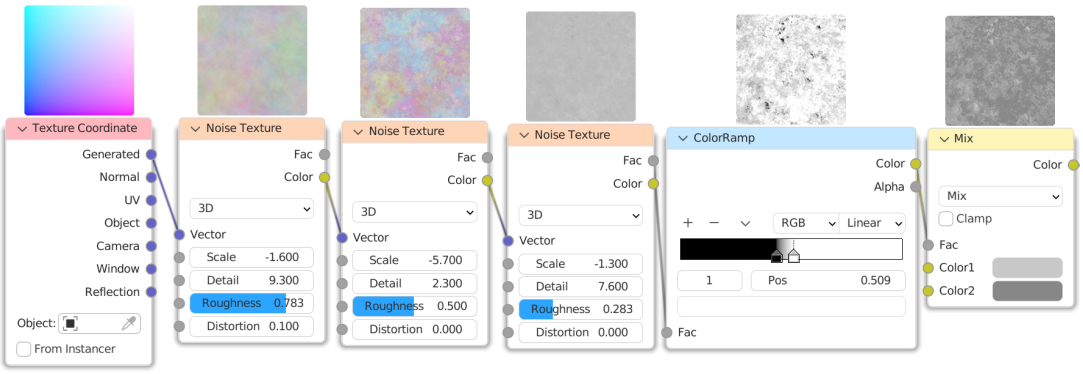
Figure 4.2: A complex procedural material node setup to produce a hardwood floor material. Customizable parameters, such as grout width, damage amount, and grain size, allow for artistic control.

maps (e.g., colour, roughness, metal-ness, normal) to describe the surface's response to light, procedural materials derive these maps mathematically.

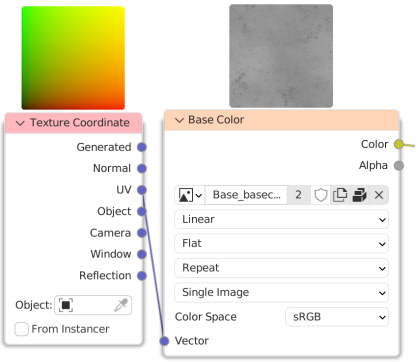
Noise Generators in Procedural Textures

In Blender, noise generator nodes play a central role in creating procedural textures. These nodes can generate different types of noise, typically controlled by just a few parameters. Textures can be generated with dimensions ranging from 1D to 4D:

- **1D and 2D Noise:** Suitable for simple textures like gradients or surface patterns.
- **3D Noise:** Allows textures to seamlessly extend across three spatial dimensions without discontinuities, making it ideal for volumetric textures or seamless surfaces.
- **4D Noise:** Adds an additional parameter (often time or evolution), enabling dynamic effects.



(a) Procedural generation using three noise textures feeding into each other, combined with a color ramp and mix node to map values to colors.



(b) Image texture setup with a single image outputting colour information.

Figure 4.3: Comparison of shader node setups for image texture and procedural shaders. This excerpt shows the generation of the color texture map only. Principled BSDF and output nodes are omitted for brevity.

Noise generators typically provide at least one intensity output (`float` values), with many also offering colour outputs. This versatility enables the generation of textures for various applications, including volumetric materials or multi-surface objects, without visible repetition or seams. Figure 4.3 compares a simple procedural shader setup with an image texture setup.

Complex Procedural Materials

Creating believable procedural materials often requires multiple noise generator nodes working in tandem. For instance:

- **General Noise Generator:** Simulates micro-surface roughness and base colouration using a low scale.
- **Voronoi Noise Generator:** Creates larger pores, cracks, or surface details.

- **General Noise Generator with different parameters:** Adds irregular patches of discolouration or roughness.

The outputs of these generators can be further processed to generate color, roughness, and normal maps.

Grouping and parameter exposition

To simplify complex procedural node setups, Blender allows artists to group nodes together. Parameters can then be exposed and named. By exposing only the relevant parameters, artists can focus on creative decisions rather than technical details. As an example; for a concrete material; *grout size*, *damage amount* or *wetness* could be exposed, enabling the artist to tune those parameters without editing the underlying node setup.

Figure 4.2 illustrates a complex procedural material for a hardwood floor. The setup includes multiple noise generators and mathematical operations, all customizable through user-defined parameters.

The Blender material system I just described should provide a basic understanding of the concepts needed for this thesis as it is used to create the training dataset, as well as the output of the inference pipeline. A completely different part of these pipelines is how the system tries do understand and replicate style, without relying on pixel values. To give some background into Gatys style descriptors, and the underlying CNN architecture, follow me into the next section.

4.2 Convolutional Neural Networks (CNN)

In order of creating materials with the aforementioned shader system, that try to replicate a given materials look without copying pixel values, we need to gain some understanding in how a Gatys style descriptor works. To get a step closer to that goal, I will give some foundational insight into CNNs in this section. Right after we dive into Gatys style descriptors.

When working with images and neural networks, CNNs are a natural first choice. They mimic how biological visual systems process information, progressing from simple to complex patterns, making them both efficient and effective in understanding visual features. In this work, input images are processed by VGG19, a CNN, to extract the Gatys style descriptor [GEB15b].

CNNs operate similarly to the brain's visual processing and are particularly powerful in image classification tasks, though they can be applied to other data types as well. Compared to traditional ANNs, CNNs offer several key advantages [Che+21]:

- **Parameter sharing:** Unlike fully connected ANNs, where each neuron connects to all neurons in the next layer, requiring a weight for every connection, CNNs

use the same set of weights (the kernel) across the entire input. This reduces the number of parameters and allows the network to detect similar features across different regions of the input image.

- **Spatial awareness and hierarchical structure:** CNNs analyse neighbouring pixel values in earlier layers to detect simple features like edges and lines. In deeper layers, they combine these features to detect more complex patterns and structures.
- **Translational invariance:** The sliding nature of kernel filters and the abstraction provided by pooling layers make CNNs less sensitive to the exact positions of features in the input.
- **Context preservation:** By applying convolutions to local neighbourhoods, CNNs maintain spatial relationships between pixels and later between features, improving their understanding of the input.

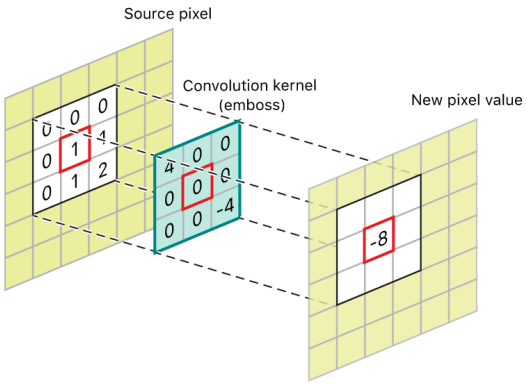


Figure 4.4: Convolution on a 2D image with a 3×3 kernel. Pixel values in the input image are multiplied by the corresponding kernel values and summed to produce a new pixel value. This example shows an emboss filter. Image courtesy of Apple [App].

A CNN is a specialized type of ANN that uses the mathematical operation of convolution instead of general matrix multiplication in at least one of its layers. In a convolutional layer (which is further explained in Section 4.2.1), each neuron applies a filter kernel to the input image (or the output of the previous layer). The kernel has dimensions $w \times h \times k$, where w and h are the width and height of the kernel, and k corresponds to the number of input channels (e.g., for an RGB image, $k = 3$).

For example, consider a neuron using a $3 \times 3 \times 3$ kernel on an RGB image. The neuron requires $3 \cdot 3 \cdot 3 = 27$ weights to process the input. Sliding the kernel over the input generates a greyscale output with the same resolution as the input, containing information about the presence of a specific feature in each local neighbourhood. In contrast, a neuron in a fully connected network would require a separate weight for every pixel in the input, losing spatial information and being more prone to over fitting.

The hierarchical structure of CNNs, combined with pooling layers (detailed in Section 4.2.2), allows them to detect relationships between features at multiple levels of abstraction. Early layers detect simple features, such as edges, while deeper layers combine these features to recognize complex patterns and objects.

While convolutional architectures like VGG19 use a large number of parameters (e.g., 144 million [SZ14]), their efficiency lies in the reuse of weights through convolutions. This approach allows CNNs to scale with input resolution and maintain spatial awareness without an exponential increase in weights.

4.2.1 Convolutional Layers

Convolutions are used extensively in image processing as filters, where a numerical matrix (the kernel) slides over the input image. At each position, the pixel values in the current window are multiplied by the corresponding kernel values and summed:

$$y_{ij} = \sum_{m=1}^{w} \sum_{n=1}^{h} \sum_{c=1}^{k} x_{(i+m)(j+n)c} \cdot w_{mnc}$$

where y_{ij} is the output at position (i, j), x is the input image, and w is the kernel. Figure 4.4 illustrates this process for a single-channel image. In multi-channel images (e.g., RGB), the kernel depth k matches the number of input channels. Kernels can detect specific patterns, such as edges or textures. For tasks like image classification, these kernels are trainable, enabling the network to learn features relevant to the input data.

4.2.2 Pooling Layers

Pooling layers perform a down-sampling operation along the spatial dimensions (width and height) of the input image. The most common pooling method is max pooling, which selects the maximum value in a defined neighbourhood (e.g., a 2×2 window):

$$y_{ij} = \max\{x_{(i+m)(j+n)} \mid 0 \leq m, n < p\}$$

where p is the pooling window size. Average pooling, another common method, computes the mean value within the neighbourhood.

Pooling reduces the spatial resolution of the input, lowering the computational load in subsequent layers and providing some additional translation invariance. It also increases the receptive field of neurons in deeper layers, enabling them to consider larger regions of the input. This abstraction is crucial for tasks like object recognition, where exact feature positions are less important. The explanations by Jason Brownlee [Bro19; Ian16; Cho17] were particularly useful in understanding these mechanisms.

4.2.3 VGG19

Simonyan et al. investigated the impact of network depth on CNN performance [SZ14]. In their research, they developed and trained multiple CNNs with depths of up to 19

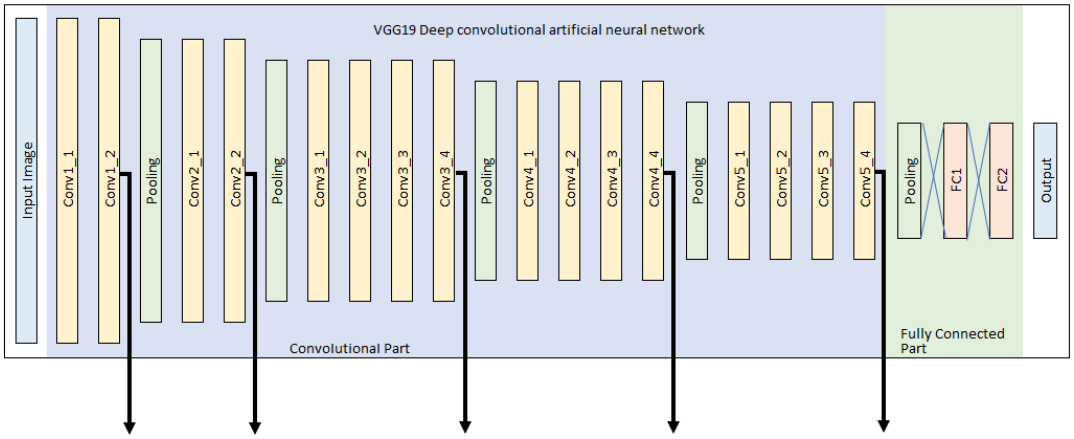


Figure 4.5: VGG19 network layer overview. For style extraction, only the convolutional part is necessary. The fully connected part is discarded for style extraction. The layer activations before every pooling layer (here signified by arrows) are used to calculate the Gatys style descriptors. (ReLU layers are not shown for brevity.)

layers for image classification. The 16-layer and 19-layer weighted versions of these networks were made publicly available for further research. The term **VGG19** refers to the version with 19 weighted layers.

They demonstrated that even a CNN using very small convolutional kernels (3×3 pixels) can achieve excellent results when the network depth is sufficiently increased. The network comprises 19 weighted layers, including 16 convolutional layers and 3 fully connected layers, which are used for classifying the convoluted image data. In addition to the weighted layers, there are 5 pooling layers that divide the convolutional layers into 5 groups, as illustrated in Figure 4.5.

Style Transfer and VGG Networks

Style transfer algorithms have extensively utilized the VGG networks for style extraction [GEB15b; GEB15a; GEB16; AAL16]. For this purpose, the fully connected part of the network is not necessary, as it is solely used for classification. The convolutional layers, particularly the activations before each pooling layer, are used to calculate the Gatys style descriptors, as explained in Section 4.3.

Now that we achieved some foundational understanding of CNNs, we can take a look into how style descriptors are extracted from them.

4.3 Gatys Style Descriptor

In the previous section we learned how CNNs are trained to abstract and understand features form an input image. Now we will take a look on how these features can be used

to gain statistical information about the same input image.

The Gatys Style Descriptor, first introduced by Gatys et al. [GEB15b], was originally used for artistic style transfer. Since then, it has been extensively applied in style extraction, style transfer, and material generation [GEB15b; GEB15a; GEB16; AAL16; Li+17]. The descriptor represents texture and style as a statistical image feature by capturing relationships between features in a convolutional neural network (CNN) layer. This is achieved by computing the Gram matrix, which encodes correlations between neuron activations in the same layer.

Understanding the Gram matrix is crucial, even though its quite a simple calculation, its effect is very important; as it enables the style descriptors to capture the relationships between different features in the input image.

4.3.1 Gram Matrix Calculation

The Gram matrix measures how frequently different visual features appear together in an image, effectively summarizing its overall texture and style. It does this by computing the inner products between pairs of feature maps. In this thesis, a Gram matrix is computed for each of five different VGG layers by taking the inner product between that layer's activations and their transpose, capturing the relationships among detected features at various levels of abstraction.

The Gram matrix for a convolutional layer l is defined as:

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

where:

- F^l is the feature map of layer l,
- i and j are indices corresponding to the feature maps, and
- k iterates over the spatial dimensions of the feature maps (height and width).

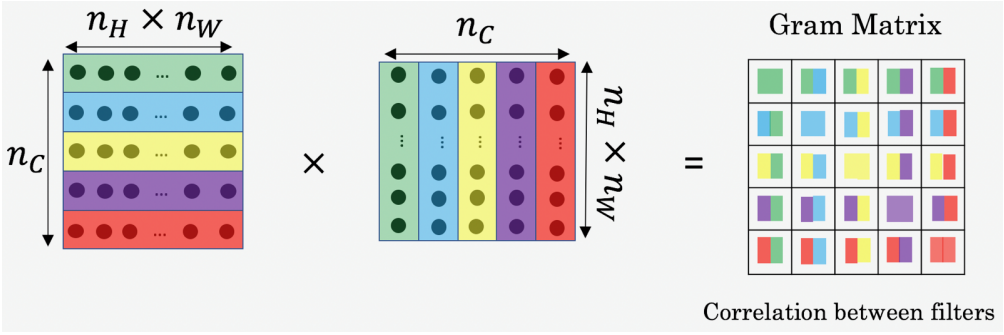


Figure 4.6: The Gram matrix represents correlations between feature maps by computing the inner product of the activation vectors in a CNN layer. Graphic adapted from [Res24].

The resulting matrix G^l describes how the activations of different feature maps in layer l correlate with one another, capturing the style or texture without relying on spatial information. A visual aid for this calculation can be found in Figure 4.6.

4.3.2 Multiple Layers for Varying Levels of Abstraction

Gatys et al. proposed using activations from multiple CNN layers to generate a more comprehensive style descriptor. This approach leverages the fact that different layers capture different aspects of an image's style. Lower layers tend to preserve fine, local details—such as textures and edges—while higher layers capture more abstract and global features like overall patterns and color distributions.

In practice, the style representation is constructed by computing the Gram matrix for several layers and then combining them. This multi-layer strategy allows for:

- **Enhanced Detail:** Lower layers contribute precise texture information.
- **Robust Abstraction:** Higher layers provide a holistic view of the style.
- **Balanced Representation:** The combined descriptor reflects both local and global style features, leading to more precise and versatile style extraction.

This layered approach is central to achieving effective style extraction, as it ensures that both subtle details and overall aesthetics are incorporated into the final material generation process. In this thesis I the activations of 5 layers of the VGG19 network as further explained in Section 5.2.2.
With this overview of underlying technical ideas, the next chapter will dive head first into how these concepts were implemented.

CHAPTER 5

Implementation

Now that the technical foundation has been laid, I will go into detail how those concepts were used and implemented. I will again start with the Blender shader system, this time focusing on the construction of the *super-shader* and the extraction of *sub-shaders* after it was parametrised. Right after, in Section 5.2 I will explain how the training data-set is generated. Following that we will take a look into the training process of the *parameter predictor*.

5.1 Super-Shader

The system is capable of delivering a wide variety of materials, all of which are derived from a single, large, and non-differentiable shader referred to as the *super-shader*. It was built manually, with the goal of providing a way of control different approaches to noise layering with parameters. The resulting noise textures are used for multiple SVBRDF maps (base colour, metallic, roughness and normals). Because the same few noise textures are used for all four SVBRDF maps, just combined differently, the resulting materials make sense, and do not look/feel random. Parameters coming from the MLP

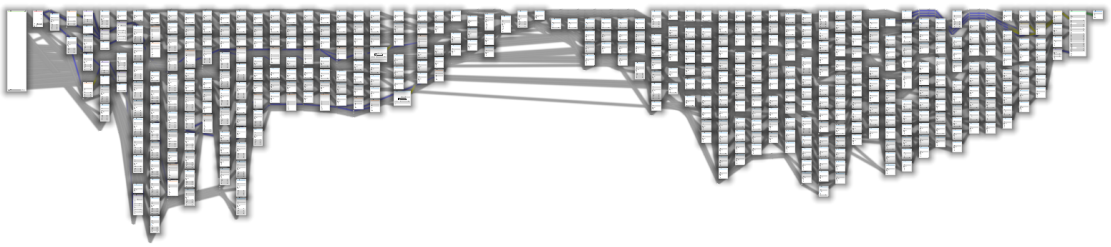


Figure 5.1: While the *super-shader* is capable of delivering a wide range of different materials by changing its parameters, with its 507 nodes it is too large and complicated to be manageable by an artist.

or from randomisation on data generation are always between 0 and 1. But most node parameters need other value ranges. The mapping to sensible ranges is done via nodes as well, and the ranges have been selected by hand.

The super-shader employs different modules; noise generation, mathematical operations and value mapping. Multiplexers in those modules are used to select which type of noise, mathematical operation or mapping should be used. These multiplexers make the shader non-differentiable because they involve discrete choices between operations. This modular design, while powerful, adds significant complexity and abstraction.

All together, the shader contains 507 nodes, organized into modules and is controlled by 50 mutable parameters. These parameters allow the shader to produce diverse materials by selecting combinations of mathematical operations, value mappings, and noise generators. However, its complexity and size make it difficult to use directly in practical applications.

To address this, the shader parametrisation is followed by an optimization step executed by a script (theory and implementation details can be found in Section 5.1.2). This process extracts a smaller, simplified *sub-shader* from the parametrised *super-shader* that is easier to use, comprehend and customise. After execution of the script, the amount of nodes is reduced by more then a magnitude, from 507 to around 30 (exact number is different for different parametrisations) and still yields the same result. This optimisation step significantly reduces rendering times by removing unused nodes and modules.

5.1.1 Node Modules

Because the same group of nodes are useful in different places in the shader, a modular approach was chosen. These modules in itself use multiplexing modules, which select the operations that will be used, the operations themselves, and the handling of the additional parameters the operations might need.

Noise Generators: In Blender current version (4.2), it has three different noise generator nodes (general noise, Voronoi noise and white noise, although the latter is not used in the super-shader) that can be setup with different parameters to produce a wide variety of different types of noise pattern. The module defines nine different noise patters that themselves are further customized with numerical parameters.

Mathematical Operations: Implements addition, subtraction, multiplication between signals as well as taking the minimum or the maximum or lastly getting the relation between the first and second signal by division.

Multiplexing: For most other modules, hard switching multiplexers are used. A floating point input between 0 and 1 is used to select between different amount of inputs. For the colour blending a interpolating, smooth switching multiplexer is used.

5.1.2 Sub-Shader Extraction

A script is used to make the very large and hard to work with *super-shader* more manageable for artists. When the *super-shader* is parametrised, the multiplexing parameters will have already switched off large parts of the shader, and some value parameters will have lead to redundant or trivial calculations. Those switched off sub graphs can be removed entirely, and calculations that are trivial can be removed and replaced with precomputed values as well. A detailed description of the scripts process can be found below.

Extraction Process

The script walks through the shader graph multiple times and performs following operations:

1. **Node Pruning:** Removing or bypassing nodes where the calculation can be performed by the script:
   - Detecting maths nodes with only static values as inputs; The mathematical operation is computed by the script, and the result is assigned as a static value to the input of the next node that would have received the original result.
   - Maths nodes with one varying input v and one static input s, where the operation always returns the varying input. For example:

     $$f(v, s) = v \quad \text{if } s = 1 \text{ (e.g., multiplication: } v \cdot 1 = v).$$

     In this case, the varying input v is directly connected to the input of the subsequent node. Leaving the currently optimised node without having any outgoing connection.
   - If the operation results in a static output regardless of the varying input v, the static value can replace the operation entirely. For example:

     $$f(v, s) = 0 \quad \text{if } s = 0 \text{ (e.g., multiplication: } v \cdot 0 = 0).$$

     This static value is then directly assigned as the input to the following node. Here the connection to the following node is just removed entirely, making the currently optimised node having no outgoing connection again.
2. **Node Clean-up:** Remove nodes whose output is not connected to any subsequent node. These nodes do not contribute to the shader tree and can be safely deleted.
3. **Ungrouping:** Ungroup all node groups, as their functionality is no longer necessary after pruning and they hinder usability by obscuring the simplified shader structure.
4. **Reordering:** After removing nodes and groups, the shader tree may become disorganized. The script will reorder all nodes into a structured grid layout, aligning them according to the data flow from left to right.

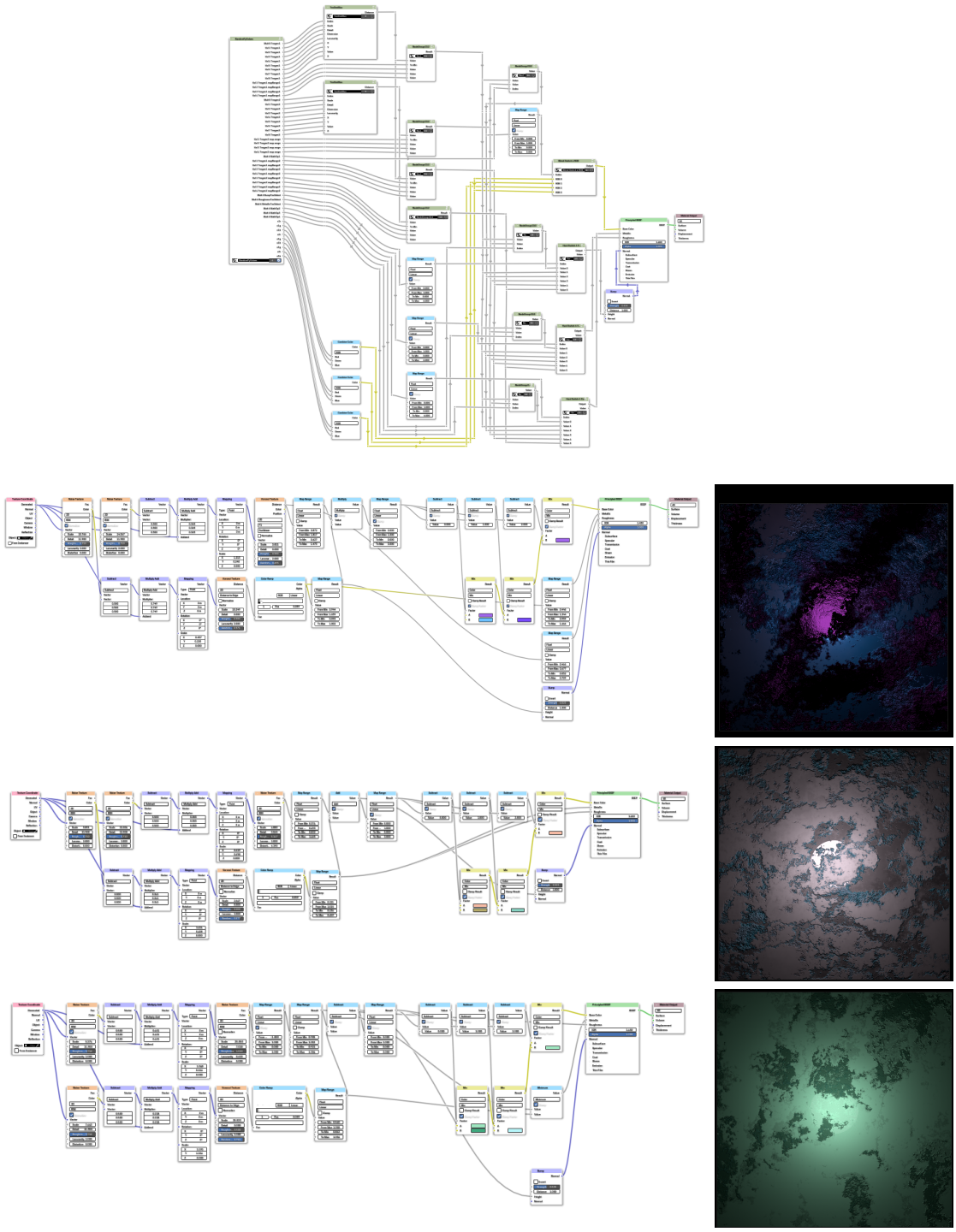


Figure 5.2: Comparison between full *super-shader* with its 507 nodes (with its groups intact, ungrouped is displayed in 5.1), and minimised and reordered shaders produced by the *sub-shader* extraction script with around 30 nodes and their corresponding renders.

Improved Usability

After the *sub-shader* extraction process, the resulting shader retains only the parameters necessary for material calculation and noise configuration. Meaning the resulting render will look exactly the same. This simplification enhances usability for artists, providing a streamlined interface while maintaining the shader's flexibility. Results of this script can be seen in 5.2.

5.2 Learning Data Generation

To train the *parameter predictor* to infer shader parameters from style descriptors, a comprehensive training dataset must be generated. This dataset comprises pairs of network inputs and their corresponding ground truth outputs. In my approach, the inputs consist of transformed Gatys style descriptors, while the outputs are the shader parameters used to generate the corresponding material.

These input-output pairs are generated by randomly selecting 50 shader parameter configurations for the super shader, rendering the resulting image, and then extracting its style descriptor. The random shader parameters serve as the ground truth outputs, and the extracted style descriptors become the network inputs.

5.2.1 Rendering with Random Shader Parameters

Rendering with random parameters is an essential step in generating a diverse dataset for training. Blender's Python scripting capabilities are leveraged to automate this process, ensuring reproducibility and efficiency.

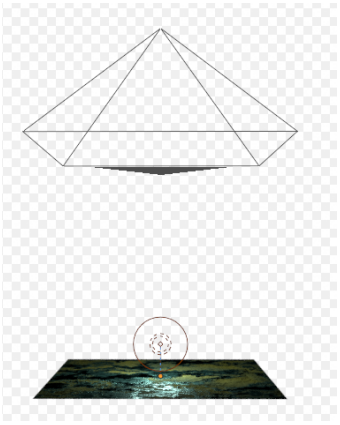
Random Parameter Generation

The super-shader is parametrized using 50 uniformly distributed random values between 0 and 1. These values are generated by a Python script registered to run every frame. The script uses the frame number as the seed for the random number generator, ensuring that the parameter set for each frame is unique and reproducible. The script logs the random parameter values into a file and includes them in the rendering metadata.

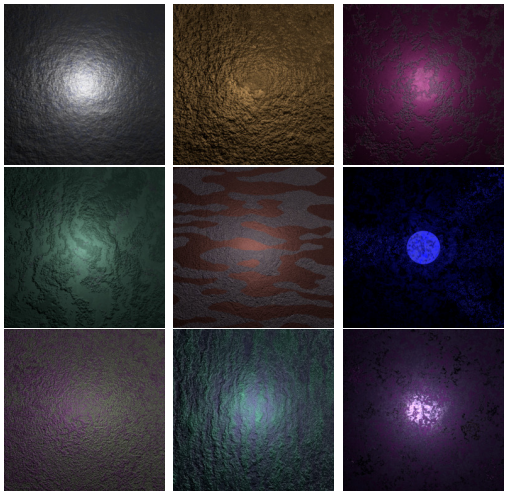
Scene Setup for Rendering

A simple rendering setup is used to accentuate surface properties like bumpiness and reflectance. The scene consists of:

- A single, relatively dim point light source placed close to the surface.
- A rendering resolution of 256×256 pixels.
- 252 samples per frame using Blenders cycles rendering engine.



(a) Blender scene setup. Camera perfectly frames the surface, single point light source close to the surface accentuates material bumpiness.



(b) Nine randomly selected material samples that have been generated for the learning data set. The effect of the single point light source is clearly visible.

Figure 5.3: Learning data set generation in blender and some results.

- no de-noising.

Figure 5.3a illustrates this setup. Using this configuration, 100,000 frames were rendered to create the training dataset. Additional frames can be generated as needed with minimal effort.

Rendering Automation and Scalability

The rendering process is automated using a Python script that efficiently manages render jobs. Users can specify a range of frames to be rendered, and the script checks the output folder for any missing frames; rendering only those to avoid redundant computations. Furthermore, the script supports concurrent rendering, thereby fully utilising available processing power to expedite dataset generation. This flexible setup ensures that new frames can be seamlessly generated on demand. A selection of rendered samples is shown in Figure 5.3b.

5.2.2 Gatys Texture Descriptor

The process of generating Gatys texture descriptors involves computing Gram matrices from layer activations of the VGG19 network (details on gram matrix calculation in Section 4.3.1). Five convolutional layers of the pre-trained VGG19 network were used to calculate style descriptors. Each layer produced a Gram matrix, which was flattened and

concatenated into a single 1D vector that served as the input for the *parameter predictor* (see Figure 4.5 for details on which layer activations were used)[1].

Descriptor Construction

Several modifications were made to adapt the pretrained VGG19 network for this task. To compute style descriptors just the layer activations of some of the convolutional layers are used. The classification will not be used at all, therefore all fully connected layers, normally used for image classification, were removed to improve computational efficiency. Additionally, new layers were added after each convolutional block to compute and store the Gram matrices, which capture pairwise relationships between feature maps, while passing the activations through unchanged. Following the approach of Liang et al. in the MATch paper [Shi+20], in-place ReLU layers were replaced with out-of-place versions to avoid overwriting activations during calculations.

To ensure compatibility with the training data used to train VGG19, the colour values of the input images were normalized using values defined by Simonyan et al. in their original paper [SZ14]. During the calculation of Gram matrices, the system also tracked the maxima and minima of each matrix, which were later used to normalize the values for consistency.

The descriptor calculation script processes the outputs of five selected layers in the network. Gram matrices are computed for each layer and saved to disk, along with metadata that records which renders have already been processed and min/max values over all gram matrices per layer. This approach ensures that previously processed renders are not recalculated unnecessarily, which is particularly important when working with datasets requiring hours or even days of computation.

Normalization is performed in a separate script, which loads the stored Gram matrices into memory, scales them based on the previously recorded min/max values, and saves the normalized results back to disk. The metadata file is updated to reflect the completion of this step. This setup allows for seamless expansion of the dataset—new frames can be rendered and integrated into the workflow without recalculating completed data. In cases where new renders introduce differing min/max values for a layer, all Gram matrices for that layer are re-normalized to maintain consistency.

This system's use of metadata not only ensures scalability but also makes it robust against interruptions. Previously computed results are preserved, and new frames can be added without redoing earlier calculations, enabling efficient incremental dataset generation.

[1] *Retrospective Analysis:* The chosen approach of flattening and concatenating Gram matrices captures intra-layer correlations effectively but fails to account for inter-layer relationships. A more effective method might involve concatenating the feature activations of all convolutional layers into a single vector and calculating one large Gram matrix. This combined Gram matrix would represent both intra- and inter-layer correlations, potentially yielding a richer and more comprehensive style descriptor. However, this alternate approach would result in a significantly larger descriptor, introducing challenges in terms of computational requirements and memory usage. These trade-offs should be carefully considered in future implementations, as discussed in Section 6.3.2.

5.3 Parameter Predictor Training and Evaluation

Input	Layer	Output
348170	FC 1	1000
1000	FC 2	1000
1000	FC 3	100
100	FC 4	50

(a) Prior input reduction via RFR.

Input	Layer	Output
2500	FC 1	2500
2500	FC 2	2000
2000	FC 3	1000
1000	FC 4	1000
1000	FC 5-10	1000
1000	FC 11	500
500	FC 12-14	500
500	FC 15	100
100	FC 16-18	100
100	FC 19	50
50	FC 20-22	50
50	FC 23	Num Classes

(b) After input reduction.

Table 5.1: Network layer setup (RELU layers omitted, layers with same parameters grouped). FC stands for Fully Connected.

The training process was implemented using PyTorch, utilizing a simple fully connected network architecture and the ADAM optimizer. Various configurations for network depth, layer width, and batch size were tested to balance VRAM limitations with performance. However, the large number of input parameters quickly overwhelmed the VRAM capacity of consumer hardware. To address this, a random forest regressor was tested for input reduction across multiple outputs (see Section 5.3.1). While PCA [Shl14] was considered as an alternative, it was ultimately not implemented due to its lack of support for multiple output nodes by default.

For the loss function, MSE loss was initially used. The MSE loss function is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i represents the true values, $\hat{y}_i$ are the predicted values, and n is the number of samples. While MSE is effective for many tasks, in this case the network to converged to the mean of the dataset, failing to capture finer details in the output.

To address this issue, MSLE loss was adopted. The MSLE loss function is defined as:

$$\text{MSLE} = \frac{1}{n}\sum_{i=1}^{n}\left(\log(1 + y_i) - \log(1 + \hat{y}_i)\right)^2$$

Unlike MSE, MSLE incorporates a logarithmic component that assigns greater weight to smaller discrepancies, making it more robust to data imbalances. This adjustment

helped slightly to mitigate the network's tendency to over fit the dataset mean, but did not succeed fully (further discussed in Chapter 6).

The best results were achieved with a very shallow fully connected network architecture consisting of two hidden layers (see Table 5.1a). Despite these promising results, the training run was terminated after nearly four days due to computational constraints, as further detailed in Chapter 6.1.1.

5.3.1 Input Reduction Using Random Forest Regression (RFR)

The initial approach was to use the flattened and concatenated Gatys texture descriptors as input nodes for the estimator network. However, the large number of input nodes ($348,170$ in total) made it infeasible to train a deep or wide neural network. The resulting small network lacked the capacity to learn effectively from such abstract data.

To address this, a RFR from Scikit-learn [sci25a] was employed to evaluate the importance of each input parameter in predicting the shader parameters. The RFR builds an ensemble of decision trees and combines their outputs, making it effective at capturing complex, non-linear relationships. However, the basic RFR is designed to work with only a single output variable.

To overcome this limitation, a multi-output regressor from Scikit-learn [sci25b] was paired with the RFR[2]. This wrapper fits a separate RFR model for each output, thereby allowing all target shader parameters to be evaluated independently.

Because training this multi-output model on the full dataset was computationally demanding, the regression was performed on a small subset of the data. After several days of computation, it became clear that only about $2,500$ of the input parameters had a significant effect on the target values.

This reduction in input features was crucial for lowering computational demands and for enabling the subsequent MLP to be trained more effectively. By filtering the inputs and retaining only the $2,500$ most impactful parameters, it became possible to train a much larger and deeper neural network (see Table 5.1b). Despite this improvement in network architecture, the training process still did not achieve convergence.

This chapter has outlined the practical realization of my system. From constructing the *super-shader* and employing the *sub-shader* extraction script, to training and evaluating the *parameter predictor*. The detailed implementation demonstrates both the capabilities and challenges of the approach. In the final chapter, I reflect on these challenges, discuss the successes achieved, failures encountered, and propose ways for future work that could build upon these findings. This concluding discussion will not only assess the current contributions but also hopes to set the stage for further advancements in procedural shader generation.

[2]After the fact, I discovered that scikit-learn's Random Forest Regressor can handle multi-output regression natively when provided with a multi-dimensional output array. Using this built-in capability might have had offered faster computation at the expense of some flexibility.

CHAPTER 6

Results and Conclusions

The core goal of this project was to train a network that could generate a Blender shader to produce a material resembling an input image. Although this main objective was not met, there are clear successes that can be built upon for further study and experimentation.

My experiments revealed that the chosen network architecture struggled with the complex mapping from the input image (and its Gatys style descriptors) to the shader parameters. With small training samples, the network over-fitted, while with larger datasets it consistently converged to the average of all desired outputs. In one instance, the network did show some improvement, but the progress was so slow that training was halted after four days.

I conclude that the network architecture was not well-suited for the high level of abstraction in this problem. The mapping between the input image, the Gatys style descriptors, and the shader parameters is inherently abstract, and adding the multiplexing nodes only increased that complexity. Overcoming this would likely require a deeper network, more training iterations, and extensive parameter tuning. Unfortunately resource limitations prevented further exploration.

On the other hand, the work done on the super-shader has proven successful. With its 50 shader parameters, it can generate an impressive range of diverse materials. Combined with the *sub-shader* extraction script, this offers a straightforward way to create hundreds of thousands of small, understandable shaders that could serve as datasets, tools for artists, or even the foundation of a material database.

While potential implementation errors may have also contributed to the challenges, this project provided valuable insights. I gained a deeper understanding of neural networks, their limitations, and the intricacies of applying them in complex workflows. These lessons have already been beneficial for my professional development.

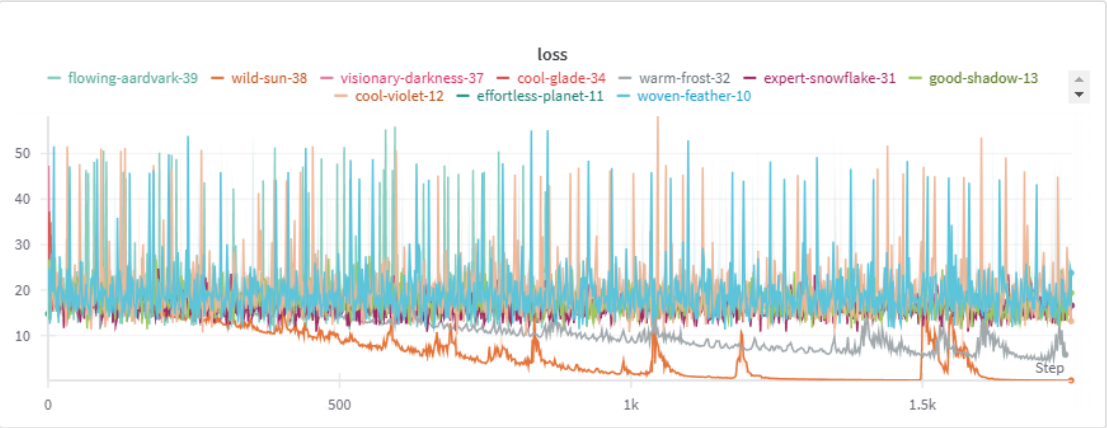


Figure 6.1: The loss graph of 40 different training runs shows the inability of the network to converge to the correct results, using different network and hyper parameter configurations. The two notable curves that seam to converge, show runs where I tested if the network is able to overfit given a very limited dataset.

6.1 Issues

The primary challenge of this project leading to its core objective staying unfulfilled was the incorrect estimation of its scope and complexity, compounded by a suboptimal allocation of time and resources. These factors severely limited the ability to adapt to unforeseen issues and iterate on the project.

The majority of time was spent developing the super-shader, generating and rendering the training data, and extracting Gatys style descriptors. As a result, insufficient time was dedicated to neural network research and development. This limitation was further exacerbated by the long training durations for each attempt and the inconsistent availability of the necessary hardware.

Another significant issue was the late realization, that my understanding of neural networks and their requirements was inadequate. This led to flawed ideas and incorrect estimations during the early stages. A key oversight was underestimating the challenges posed by the high dimensionality of the input data. The project revealed that a large part of successful neural network training lies in preprocessing and optimizing input data. Time would have been better spent exploring data reduction techniques such as principal component analysis (PCA) [Shl14], which could have been applied to the activation tensors of the VGG19 layers. Other potential improvements include using convolutional or grouping layers and more extensive use of methods like the RFR for dimensionality reduction.

Additionally, paths to reduce abstraction could have been explored. For example, rather than comparing shader parameters directly, frames could have been rendered during training, and style descriptors of the renders could have been compared to the inputs. This approach, similar to the method employed by Liang et al. in MATch [Shi+20], might

have produced more meaningful loss calculations and improved results. Another way to reduce the level of abstraction would have been a different handling of shader parameters. Even without removing multiplexers, unique value parameter to node function mapping could have simplified the estimator problem (more on this in Section 6.1.3).

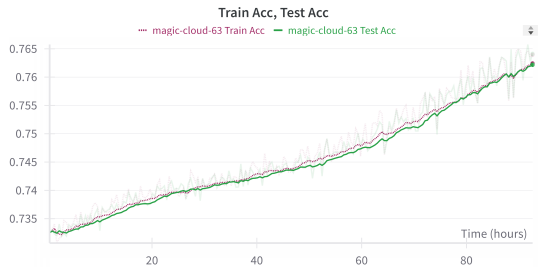
An early pivot to a system that is able to produce a vast catalogue of unique but simple Blender shaders, searchable via style descriptors, could have lead to a more positive and productive outcome as well.

In hindsight, addressing these issues earlier in the project would have significantly improved the outcomes. The experience gained has highlighted the importance of careful planning, prioritizing foundational preprocessing techniques, and iterating quickly to adapt to challenges.

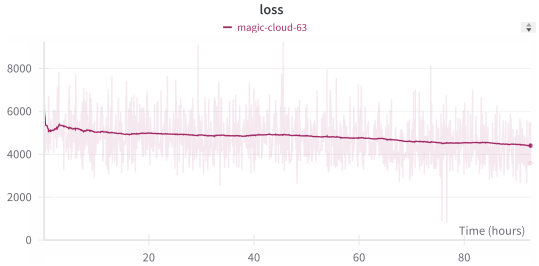
6.1.1 Non-Convergence of Estimator Network

The vast majority of training runs converged rapidly to non-functional solutions, where all outputs were identical and equal to the average of the desired output values, regardless of input. To address this, I implemented MSLE loss, which assigns more weight to small discrepancies in the desired values. Using the network configuration shown in Table 5.1 and a fixed learning rate of 0.000001, I observed some initial signs of meaningful learning. However, after nearly four days of training (3 days and 20 hours), the accuracy improved by only 0.03%, as shown in Figure 6.2. While the graph indicated that training had not plateaued, the rate of improvement was deemed too slow given the available resources, and the run was cancelled.

Subsequent attempts to reproduce meaningful convergence were unsuccessful as can be seen in Figure 6.1, even with a much larger and deeper network. Unfortunately, due to resource constraints, I was unable to perform any parameter tuning during data generation, or training. In a last-ditch effort, I reduced the input size significantly using RFR as described in Section 5.3.1. This enabled a much deeper and wider network, but also lead to no functional convergence.



(a) Training and test accuracy. Note that the y-axis does not start at zero.



(b) Loss during training.

Figure 6.2: After nearly four days, the graphs show minimal improvement in accuracy and loss. The training was canceled due to slow progression.

Additionally, since the training set was custom-generated for this project, no comparative evaluation with other methods or approaches was possible.

6.1.2 Network Choice and High Level of Abstraction

As previously discussed, the relationship between the input image and the output shader parameters is highly abstract. This abstraction is compounded by intermediate steps, such as the Gatys style descriptor and the super-shader, which remain entirely unknown to the estimator network. For the network to effectively learn this connection, it would likely need to be both very deep and, given the high dimensionality of the input data, also quite wide. Supporting such a configuration would require an enormous number of parameters. For instance, with 30,000 input nodes, 10,000 nodes per hidden layer, and three hidden layers, approximately $600 million$ weights would need to be trained—a scale far beyond what consumer hardware can handle.

Despite these complexities, the most basic fully connected ANN was chosen as the *parameter predictor* network. This network was fed unfiltered and unprocessed Gatys style descriptor data, consisting of $304,416$ float values per input. With this many nodes in the input layer, the network's width and depth were severely constrained by hardware limitations. Most training attempts used a network with only four fully connected layers (see Table 5.1a). Initially, the primary limitation was assumed to be insufficient hardware, but it later became evident that preprocessing steps to reduce input dimensionality were crucial for successful training.

When RFR was eventually employed to reduce the input nodes to just $2,500$, the network still failed to learn and did not converge. This suggested that the underlying architecture and preprocessing were not well-suited to the complexity of the task.

Further reflections revealed that the Gatys style descriptor could be restructured into a 2D matrix, potentially increasing the informational richness of the input while making it more amenable to analysis using a CNN. By employing a CNN as an encoder network to process this matrix, the dimensionality of the input to the subsequent ANN could be significantly reduced. This hybrid architecture would allow the ANN to operate on far fewer inputs, enabling deeper and more complex networks within the constraints of consumer hardware.

Such a design, resembling common image classifier architectures, could better capture the abstract relationship between style descriptors and shader parameters. It would potentially address the system's limitations without requiring more powerful hardware, providing a more effective solution to the challenges faced in this project.

6.1.3 Multiplexing *Super-Shader* Parameters

The *super-shader* relies on two types of parameters: multiplexing parameters and value parameters. Multiplexing parameters control the branching within the shader by switching between different functions or operations. Value parameters are mapped internally to

suitable ranges and fed into the shader, where their role depends on the current state of the multiplexing parameters.

The integration of multiplexers into the super-shader significantly increases both the abstraction level and the complexity of the parameter space. Multiplexers serve as discrete switches controlled by parameters (aptly named multiplexing parameters) that determine which subsets of nodes and parameters are activated. This results in two primary challenges:

Non-Differentiability

Multiplexers inherently introduce non-differentiable behaviour into the shader. During training, gradient-based methods rely on smooth transitions in the output, given smooth transitions in the input. But the discrete nature of multiplexers creates abrupt changes in the network's output. For example, a slight change in a multiplexing parameter might redirect the flow of information, causing a sudden jump in the shader's behaviour. This disrupts the gradient flow and makes it exceedingly difficult for the network to converge.

Multi-Modal Parameter Mapping

The multiplexers not only switch between different shader subcomponents but also redefine the role of various parameters based on the active branch. Consequently, the same parameter may control entirely different aspects of the shader depending on the current multiplexer configuration. This leads to a highly non-linear and non-continuous mapping between the input (Gatys style descriptors) and the output (shader parameters). The network is then faced with the challenge of figuring out the role of specific value parameters in the context of the set multiplexing parameters. The resulting combinatorial explosion in possible configurations can cause the network to converge to an average solution rather than specializing in the appropriate modes.

In summary, the multiplexing mechanism adds a layer of conditional abstraction that complicates the learning process. Not only does it break the smooth differentiability required for effective gradient descent, but it also imposes a multi-modal structure on the parameter space, demanding a deeper, more sophisticated network architecture and more advanced training strategies. These challenges, combined with resource constraints, likely contributed to the observed inability to achieve meaningful learning outcomes in this work.

Removing multiplexing parameters entirely would be quite difficult. However, by increasing the number of parameters and mapping them to the same node inputs, regardless of whether a node is currently activated by a multiplexer, the additional complexity introduced by the multi-modal parameter mapping could be reduced.

Liang Shi et al.'s approach in MATch [Shi+20] avoids these challenges entirely. Instead of relying on non-differentiable parameters, their method fixes such parameters to predefined values and employs separate estimator networks for each shader configuration. This

not only simplifies the task for their networks but also enables them to back propagate through their shader graphs. By iteratively refining their results as a post-processing step, they achieve greater accuracy and flexibility in parameter prediction.

6.2 Successes

Despite the challenges faced in training the neural network, the development of the *super-shader* and *sub-shader* extractor stands out as an achievement. In particular, these components demonstrate that a single, unified shader can be highly versatile and practical. Key successes include:

- **Versatile Shader Generation:** The *super-shader*, controlled by 50 adjustable parameters, can produce a wide range of diverse materials. This confirms that complex material properties can be encapsulated within a single shader framework.
- **Efficient Simplification:** The *sub-shader* extraction script successfully reduces the parametrised complex shader graph from over 500 nodes to around 30 nodes. This simplification not only improves usability and manageability for artists but also reduces rendering times significantly.
- **Dataset and Catalogue Creation:** Together, the *super-shader* and *sub-shader* extraction process enable the automated generation of extensive shader datasets. These datasets can serve as a foundation for further academic studies in material synthesis and provide a space-efficient Blender material catalogue comprised of random, yet comprehensible, shaders.
- **Commercial and Practical Potential:** Beyond research, the methods developed here could be adapted into commercial tools, offering 3D artists a flexible and customizable approach to material creation.

These successes lay the groundwork for future work, suggesting that a unified procedural approach to shader generation holds promise for both academic exploration and practical application.

6.3 Future Work

This project, while yielding limited positive results, provided valuable insights and identified key areas for improvement in shader generation workflows. The knowledge gained should assist future research and experimentation. To support these efforts, the dataset generation code, which leverages Blender's free and open-source platform, is accessible on GitHub [Win25]. Additionally, the Gram style descriptor code, RFR, and network training scripts, complete with a Docker environment for easy setup, is available in the same repository.

6.3.1 Super-Shader

The *super-shader* is effective in generating diverse materials but presents challenges for parameter prediction. A promising approach to address this involves splitting the *super-shader* into multiple differentiable shaders. This can be achieved by iterating through all combinations of multiplexing parameters and using the optimization script to prune unused nodes for each combination. The result would be a collection of simpler, differentiable shaders with relatable value parameters. Everything except the iteration through combinations is already possible with the provided code.

A method similar to the approach used in MATch [Shi+20] could then be employed to select the appropriate shader for a given task. This modular strategy also enables adding new shaders to the pipeline without requiring retraining.

6.3.2 Style Descriptor Handling

The current implementation calculates five independent Gram matrices from the feature activations of VGG19's convolutional blocks as detailed in Section 5.2.2. These matrices are flattened and concatenated into a single 1D vector fed into the decoder network. However, this method has several limitations:

- Relationships between feature activations across different layers are ignored.
- Flattening Gram matrices into a 1D vector results in the loss of spatial structure.

An alternative approach would be to calculate a single, large Gram matrix from all feature activations combined. This matrix would capture both intra- and inter-layer relationships, providing richer information for parameter prediction. While this approach would increase the size of the style descriptor, the use of an appropriate encoder network could mitigate the computational challenges (see 6.3.3).

6.3.3 Network Choice

CNNs are well-suited for tasks involving multidimensional data with spatial relationships, making them a promising choice for this project. By reshaping the input data into a multidimensional field, a CNN encoder could be used to reduce the dimensionality of the input before passing it to a *parameter predictor* decoder. This approach would allow for deeper and more complex networks without exceeding the VRAM limitations of consumer hardware.

Two potential methods for reshaping the input data include:

- Stacking Gram matrices from different layers into a multichannel input. Padding or upscaling may be required due to differing resolutions.

- Concatenating feature activations from VGG19's convolutional blocks into a single vector and computing one large Gram matrix that includes inter-layer relationships.

Alternatively, separate CNNs could be trained for each Gram matrix and their combined outputs fed into the decoder network. This approach offers flexibility but may increase computational complexity.

6.3.4 Data Augmentation

No data augmentation was performed in this project, which likely limited the network's ability to generalize. Since the dataset is fully generated, implementing augmentation techniques is straightforward. Potential methods include:

- Translating UV coordinates at render time, altering the pixel values of the input images without changing the desired shader parameters.
- Moving the light source to generate multiple images with varying lighting conditions for the same shader settings, improving robustness to different lighting scenarios.
- Adjusting light colour or white balance to enhance abstraction and improve performance on varying input conditions.
- Slightly changing the render angle to introduce variability while maintaining the same output parameters.

6.3.5 Domain Shift

Once successful training has been achieved, addressing domain shift will be crucial for practical applications. Differences in input data from new renders or photographs can significantly degrade network performance. To mitigate this, datasets collected from diverse domains can be used to augment training. Additionally, introducing a domain-identifying output parameter and penalizing its accuracy in the loss function can encourage the network to become domain-invariant, as suggested by Heinze-Deml et al. [HM21].

List of Figures

Acronyms

ANN Artificial Neural Network. 19, 20, 38

CNN Convolutional Neural Network. vii, ix, 6, 19–24, 38, 41–43

MLP Multi Layer Perceptron. vii, ix, 13, 25

MSE Mean Squared Error. 32

MSLE Mean Squared Logarithmic Error. 32

PBR Physics Based Rendering. 2, 8, 9

PCA Principal Component Analysis. 32

RFR Random Forest Regressor. 1, 3, 32, 33, 36–38, 40

SVBRDF Specially Varyin Bidirectional Reflectance Distribution Function. 8–10, 25

Bibliography

[AAL16] Miika Aittala, Timo Aila, and Jaakko Lehtinen. "Reflectance modeling by neural texture synthesis". In: *ACM Transactions on Graphics (ToG)* 35.4 (2016), pp. 1–13.

[App] Apple. "Blurring an image". In: (). [Online; accessed February 25, 2024]. URL: https://developer.apple.com/documentation/accelerate/blurring_an_image.

[AWL+15] Miika Aittala, Tim Weyrich, Jaakko Lehtinen, et al. "Two-shot SVBRDF capture for stationary materials." In: *ACM Trans. Graph.* 34.4 (2015), pp. 110–1.

[Bro19] Jason Brownlee. *A Gentle Introduction to Pooling Layers for Convolutional Neural Networks*. Adapted from: Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning (Adaptive Computation and Machine Learning series)*, The MIT Press, 2016. and Francois Chollet, *Deep Learning with Python*, Manning, 2017. 2019. URL: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/.

[Che+21] Leiyu Chen et al. "Review of Image Classification Algorithms Based on Convolutional Neural Networks". In: *Remote Sensing* 13.22 (2021). ISSN: 2072-4292. DOI: 10.3390/rs13224712. URL: https://www.mdpi.com/2072-4292/13/22/4712.

[Cho17] Francois Chollet. *Deep Learning with Python*. Manning, 2017.

[EF01] Alexei A Efros and William T Freeman. "Image quilting for texture synthesis and transfer". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, pp. 341–346.

[EL99] Alexei A Efros and Thomas K Leung. "Texture synthesis by non-parametric sampling". In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. IEEE. 1999, pp. 1033–1038.

[GEB15a] Leon Gatys, Alexander S Ecker, and Matthias Bethge. "Texture synthesis using convolutional neural networks". In: *Advances in neural information processing systems* 28 (2015).

[GEB15b] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. "A neural algorithm of artistic style". In: *arXiv preprint arXiv:1508.06576* (2015).

[GEB16] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. “Image style transfer using convolutional neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2414–2423.

[Guo+20] Yu Guo et al. “A bayesian inference framework for procedural material parameter estimation”. In: *Computer Graphics Forum*. Vol. 39. 7. Wiley Online Library. 2020, pp. 255–266.

[HB95] David J Heeger and James R Bergen. “Pyramid-based texture analysis/synthesis”. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 1995, pp. 229–238.

[HM21] Christina Heinze-Deml and Nicolai Meinshausen. “Conditional variance penalties and domain shift robustness”. In: *Machine Learning* 110.2 (2021), pp. 303–348. ISSN: 1573-0565. DOI: 10.1007/s10994-020-05924-1. URL: https://doi.org/10.1007/s10994-020-05924-1.

[Hu+22] Yiwei Hu et al. “An Inverse Procedural Modeling Pipeline for SVBRDF Maps”. In: *ACM Trans. Graph.* 41.2 (Jan. 2022). ISSN: 0730-0301. DOI: 10.1145/3502431. URL: https://doi.org/10.1145/3502431.

[Ian16] Aaron Courville Ian Goodfellow Yoshua Bengio. *Deep Learning (Adaptive Computation and Machine Learning series)*. The MIT Press, 2016.

[Jul62] Bela Julesz. “Visual pattern discrimination”. In: *IRE transactions on Information Theory* 8.2 (1962), pp. 84–92.

[Kaj86] James T. Kajiya. “The Rendering Equation”. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi.org/10.1145/15886.15902.

[Kum20] Abhishek Kumar. “PBR Texturing vs. Traditional Texturing”. In: *Beginning PBR Texturing: Learn Physically Based Rendering with Allegorithmic’s Substance Painter*. Berkeley, CA: Apress, 2020, pp. 43–46. ISBN: 978-1-4842-5899-6. DOI: 10.1007/978-1-4842-5899-6_5. URL: https://doi.org/10.1007/978-1-4842-5899-6_5.

[Li+17] Yanghao Li et al. “Demystifying neural style transfer”. In: *arXiv preprint arXiv:1701.01036* (2017).

[Mae+24] Arman Maesumi et al. “One Noise to Rule Them All: Learning a Unified Model of Spatially-Varying Noise Patterns”. In: (2024).

[PS00] Javier Portilla and Eero P Simoncelli. “A parametric texture model based on joint statistics of complex wavelet coefficients”. In: *International journal of computer vision* 40.1 (2000), pp. 49–70.

[Res24] ResearchGate. *Artwork Style Transfer Model using Deep Learning Approach - Scientific Figure*. https://www.researchgate.net/figure/The-Gram-Matrix-is-created-from-a-target-image-and-a-reference-image_fig4_356667127. [accessed 3 Mar, 2024]. 2024.

[sci25a] scikit-learn developers. *sklearn.ensemble.RandomForestRegressor – scikit-learn 1.0.2 documentation*. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html. Accessed: 15 February 2025. 2025.

[sci25b] scikit-learn developers. *sklearn.multioutput.MultiOutputRegressor – scikit-learn 1.0.2 documentation*. https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputRegressor.html. Accessed: 15 February 2025. 2025.

[SF95] Eero P Simoncelli and William T Freeman. "The steerable pyramid: A flexible architecture for multi-scale derivative computation". In: *Proceedings., International Conference on Image Processing*. Vol. 3. IEEE. 1995, pp. 444–447.

[Shi+20] Liang Shi et al. "Match: differentiable material graphs for procedural material capture". In: (2020).

[Shl14] Jonathon Shlens. *A Tutorial on Principal Component Analysis*. 2014. arXiv: 1404.1100 [cs.LG]. URL: https://arxiv.org/abs/1404.1100.

[SZ14] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[VG97] Eric Veach and Leonidas J Guibas. "Metropolis light transport". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 65–76.

[Win25] Marcel Winklmüller. *Inverse Material Synthesis via Sub-Shader Extraction*. Accessed: 2025-03-19. 2025. URL: https://github.com/Liquidmasl/Inverse-Material-Synthesis-via-Sub-Shader-Extraction.

[WL00] Li-Yi Wei and Marc Levoy. "Fast texture synthesis using tree-structured vector quantization". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 479–488.