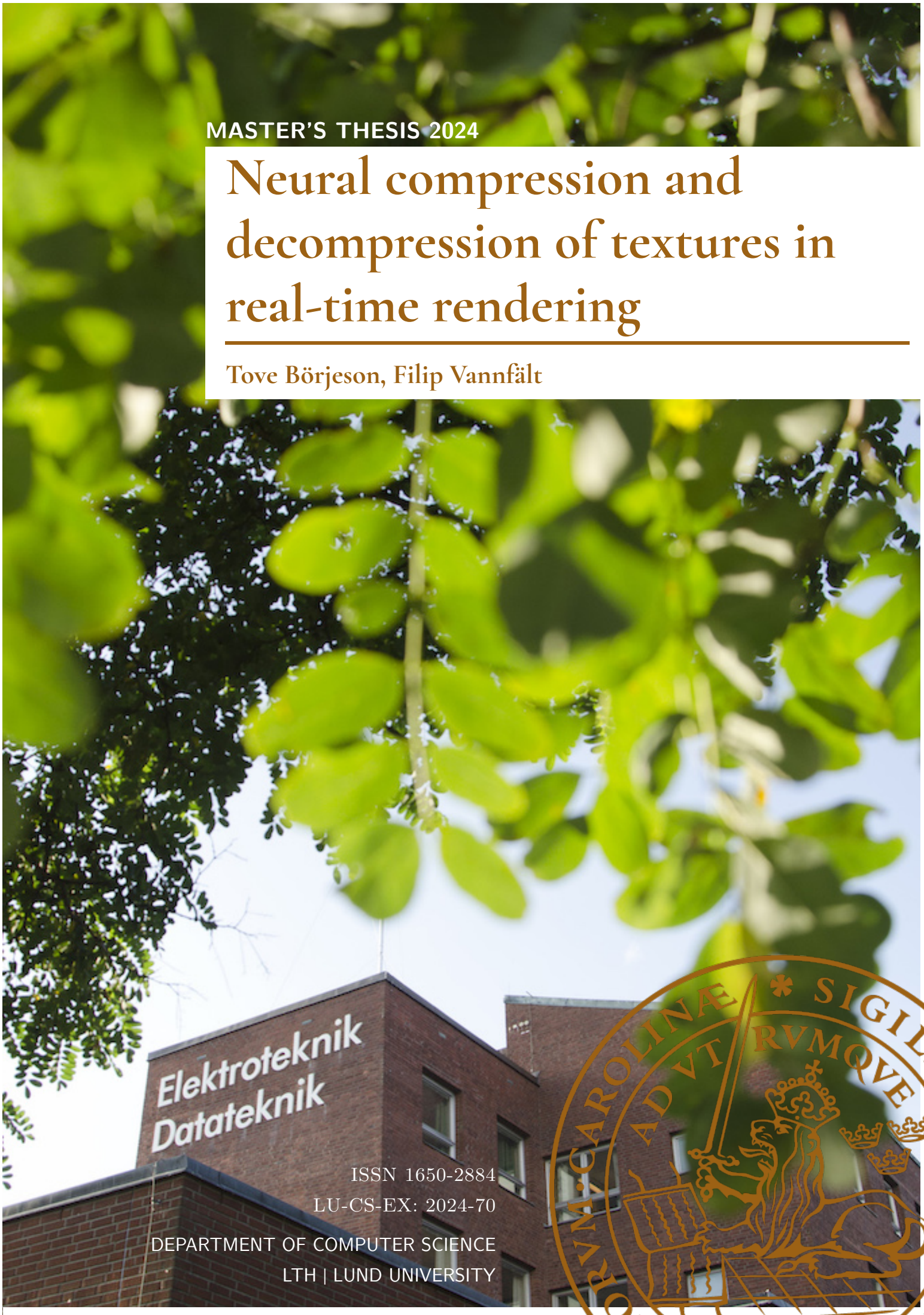


MASTER'S THESIS 2024

Neural compression and decompression of textures in real-time rendering

Tove Börjeson, Filip Vannfält

ISSN 1650-2884
LU-CS-EX: 2024-70

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-70

Neural compression and decompression of textures in real-time rendering

Neural komprimering och dekomprimering av texturer i realtidsrendrering

Tove Börjeson, Filip Vannfält

Neural compression and decompression of textures in real-time rendering

(A study of neural networks in computer graphics)

Tove Börjeson
to4335bo-s@student.lu.se

Filip Vannfält
fi3023va-s@student.lu.se

November 12, 2024

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

Supervisor: Michael Doggett, michael.doggett@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

The visual quality of real-time computer graphics has steadily increased over the past decades. However, this comes with higher storage requirements as a consequence. From this problem, the idea of representing textures with real-time inferred neural networks emerged.

This thesis explores the possibility to compress the storage of material textures by using a neural network to reconstruct these in real-time, instead of the industry standard of texture look-up. Several network models and architectures were explored, varying from simple coordinate-based models to neural feature-based models, to determine the viability of the implementation. These models and their results are compared based on the resulting image quality, the storage size of the model, and its respective inference time.

The results showed that there is potential for neural rendering as a method of image compression in real-time applications. However, it still has a long way to go to be considered as a replacement for current practices.

Keywords: neural network, computer graphics, texture compression, real-time rendering, neural rendering

Acknowledgements

We would like to thank our supervisor Michael Doggett for his unwavering support during this thesis. His insights and guidance was crucial, and ensured we stayed on course throughout the project. We would also like to thank Rikard Olajos for his help with the fragment shader inference. Finally, we would like to thank our friends and families for their support and encouragement, without which this thesis would not be possible.

Contents

Chapter 1
Introduction

In this section we give context to the thesis, define the research questions to be answered, and detail the scope of the research.

1.1 Background

The field of computer graphics has evolved steadily over time toward near-photorealistic quality. Both 2D and 3D visual effects in films and animations can produce stunning results with precise control over every minute detail in a rendered scene. In contrast, real-time rendering has long been struggling to approach the same levels of visual quality as present in non-real-time media. When a render has to be performed in a matter of milliseconds rather than minutes, hours, or even days for visual effects in other media, it is natural to expect a result of somewhat diminished quality.

A major question in real-time rendering is as follows: how much can one increase the visual quality at the cost of disk space, memory space, and rendering time? Increased detail in textures with higher resolutions is an easy way to increase visual quality, but requires an increasing amount of both disk space and GPU memory to use efficiently. In particular, completely filling up the GPU memory has a major negative impact on the general performance of the rendering. Data will need to be offloaded to the system memory and CPU-to-GPU transfers are considerably slower than in-memory accessing. For this reason, textures are often compressed so as to take as little space as possible when stored in the GPU memory.

Neural rendering is an emerging field of computer graphics where traditional methods of rendering images and video are combined with deep neural networks. By treating neural networks as a tool for approximating a function within some acceptable error, one can train neural networks to learn complex patterns that are directly used in a rendering pipeline. For example, the detailed surface of a 3D model or the way light is reflected from a surface can

be considered 'learnable', as in they both follow some underlying pattern or function. This technique could also be used to construct a 3D representation of an object from a set of 2D images from different angles.
This thesis aims to explore applications of neural rendering in texture compression, with the goal of reducing both disk storage and GPU memory usage by training a neural network to effectively represent a surface material.

1.2 Research Questions

The goal of this thesis is to investigate methods of image reconstruction using a neural network to be used in real-time rendering. Compared to Vaidyanathan et al [13], our method will use a simpler training and inference method, in order to make the technique more approachable for implementation and further studies. To properly investigate this, we pose the following research questions:

- RQ1: What reconstructed texture quality is achievable using a neural network?
- RQ2: What is the level of compression?
- RQ3: How fast is the inference in a real-time context?

1.2.1 Scope

In order to make this subject feasible for a master thesis, we will impose some constraints that limit the scope of this project to fit a 20-week timeline. Multiple texture channels are used to achieve the visual quality common in real-time rendered content today, such as normal maps, metallic maps, and roughness maps. Not all of these are required in order to produce feasible results for comparison, and thus we will only focus on albedo, normal, and specular maps.

There are many ways to achieve a reconstructed image using neural networks. We will focus on two models that we name Autodecoder and Fourier. These will be presented in Chapter 3.

We will not set any constraints on how long training of the neural network takes and therefore the practical training times are not in the scope of this thesis.

The model created was trained on RGB images and has not taken in to consideration for alpha channels. Even if images such as the specular texture are in greyscale, the textures were treated equally and the model was still trained the same as for the coloured ones.

The results will be compared to the original images the network was trained on. Current compression methods such as block compression BCn and their resulting compression rate and image quality will not be taken into consideration in this thesis.

1.3 Division of Work

During the project, there has been a division of focus regarding the implementation. Börjeson focused on the neural network specification and training in PyTorch, while Vannfält focused on the real-time inference and rendering of results. This is not to say that both parties were uninvolved in the full implementation – suggestions, ideas, and code were freely shared between the authors, and were vital to the final implementation upon which this thesis is based. Table 1.1 shows the party who wrote the majority of each section. Here, too, both worked and edited concurrently throughout the paper.

Section	Vannfält	Börjeson
1.1	X	
1.2	X	X
1.3	X	X
1.4	X	
1.5		X
2.1	X	
2.2	X	X
2.3	X	X
2.4	X	
3.1	X	X
3.2	X	
3.3	X	
3.4	X	X
4.1		X
4.2		X
5.1	X	
5.2	X	X
5.3	X	
5.4	X	X
5.5	X	X
6.1	X	X

Table 1.1: Thesis writing work division.

1.4 Related Work

In this section, we will detail existing techniques and methods that are relevant to our work. This thesis is based on several recent papers on the subject of neural rendering and techniques of image synthesis using neural networks.

1.4.1 Frequency encoding

Neural networks have been shown by Tancik et al to predictably fail to capture high-frequency data in a complex context – a concept referred to as *spectral bias* [10]. Particularly when map-

ping a coordinate to a matching output, such as a pixel in an image, the result will often be a blurry approximation of the image in that region. By transforming simple coordinate based features into a frequency space, they show that the spectral bias can be mitigated by mapping the input into a set of sinusoids of progressively higher frequency.

1.4.2 Autodecoder

Park et al introduced the *autodecoder* as an alternative to the autoencoder model for representation learning, and shows that encoder-less learning can be a viable alternative and a more efficient use of computational resources for coordinate-based problems [8]. This method is highly relevant for this thesis as it can be used to reduce the size of a network trained to solve similar problems.

1.4.3 Neural Textures

Thies et al introduces *neural textures* as a means to use trained latent codes in a neural rendering context, leveraging the fact that the trainable parameters can be saved in a grid-like pattern and used as a texture in a renderer [11]. They show that the latent code values can be directly integrated with the graphics pipeline, using the built-in texture filtering methods such as bilinear interpolation to sample the neural textures at intermediate positions. Their method of integrating latent codes in a rendering pipeline is employed in this thesis.

1.4.4 Neural Compression

Vaidyanathan et al showed that by combining the concepts of frequency encoding, autodecoders, and neural textures, it is possible to compress material textures by using layered latent codes to represent a texture set. By sharing features across mip levels, they present a method of significantly reducing storage size compared to a traditional mipmap chain [13]. This paper is the main inspiration for this master thesis and some of the methods introduced will be explored further below.

1.5 Outline

The following chapter will touch on the underlying theory behind neural networks, input processing, metrics evaluation and the rendering used in this thesis. In Chapter 3 details regarding the methodology used will be presented. In Chapter 4 the results from this thesis will be presented and in Chapter 5 these results will be discussed. In Chapter 6 the conclusions will be stated.

Chapter 2
Theory

In this chapter the theory behind neural networks and rendering will be explained. Here the ideas behind the input processing is presented.

2.1 Artificial Neural Network

Artificial neural networks are collections of matrix operations performed in sequence to mimic how biological neurons function in a brain. By connecting nodes – referred to as *neurons* – to each other a network is formed, where each connection path consist of a multiplication with a parameter called a *weight*. The neurons themselves are functions that sums the previous outputs multiplied by their corresponding weights, and then a single output is produced by passing the sum through an *activation function*. These neurons are organised in layers resulting in a collection of paths starting from the input layer, through a number of *hidden layers* whose output are not shown in computation, and a final output layer. A common example of a simple neural network is the *multilayer perceptron* or *MLP* for short, shown in Figure 2.1.

By modifying the parameters connecting each neuron in a process called *training*, these networks can be used to learn different kinds of patterns from a seemingly unrelated input. Training neural networks requires some metric that quantifies how close to the intended result a processed output is, so that the following adjustments of the parameters are proportional to how wrong a prediction is. This is provided by a *loss function* and what it represents can differ depending on what problem the network is trying to solve.

2.1.1 Models

Different patterns and configurations in how the neurons are connected and what activation functions are used are referred to as neural network *models*. These can be tailored to solve a specific kind of problem depending on how the input data is to be processed. In addition to

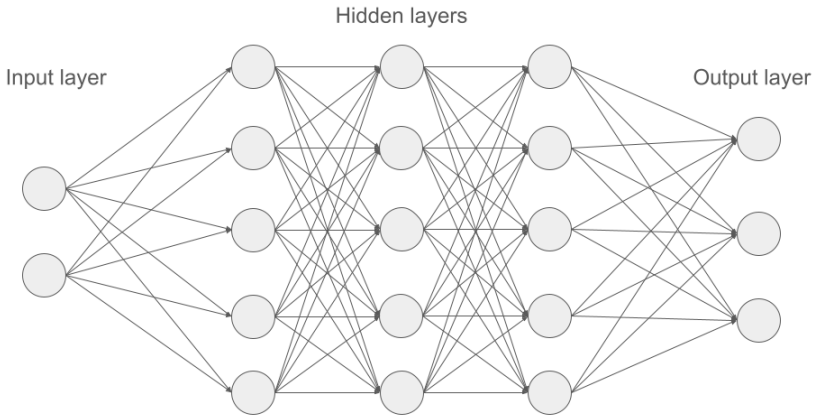


Figure 2.1: A MLP neural network consisting of fully-connected layers where the middle three layers are hidden.

the linear kind of model shown in Figure 2.1, two other models are relevant for this thesis and are detailed below.

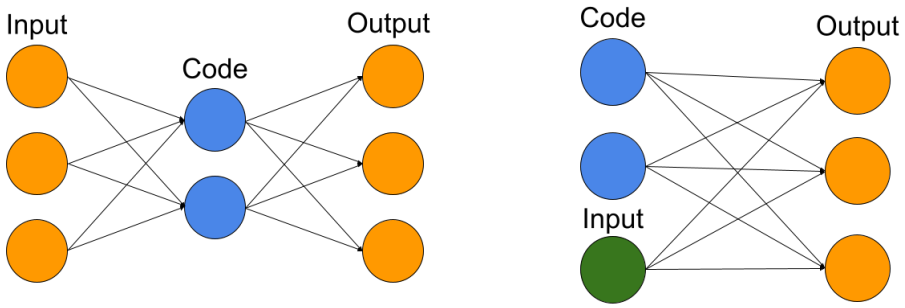
Autoencoder

An *autoencoder* is a style of network that is commonly used where one wants to find underlying features or patterns in the data and reconstruct the input from these patterns – one classic use case is denoising an image. It consists of two neural modules fitted together – starting with an encoder, which takes an input and passes it through a number of layers with decreasing sizes, resulting in a neural encoding of a desired size as shown in Figure 2.2a. This creates an informational bottleneck, forcing the encoder to only react to data that strongly influences the reconstruction. The encoding is then passed to a decoder part of the network, which feeds the code through a number of layers with sizes cascading upward until the desired output is acquired.

Autodecoder

Autoencoders are often used as tools for learning complex features and while efficient at reconstruction of partial data, they are not without drawbacks. If the encoding part of the model is viewed purely as an input processing function, multiple matrix multiplications can be seen as relatively computationally expensive.

Park et al shows that by substituting the encoder part of the network with a set of trainable latent codes corresponding to each input data point as shown in Figure 2.2b, one can leverage the training process itself to optimise the code through backpropagation [8]. Instead of just modifying the weights of each connection in the network during training, the *autodecoder* also modifies the latent code itself – the input code is thus trained to better represent underlying patterns at the same time as its interpretation by the network is trained. By separating the encoding from the neural network, this effectively creates a learnable representation that can be detached from the network inference process and stored separately.



(a) Autoencoder **(b)** Autodecoder

Figure 2.2: Simplified illustration showing differences between autoencoder and autodecoder.

2.1.2 Overfitting

When training neural network models, it is usually with the goal of solving a generalised problem from a set of examples. *Overfitting* is the concept of training a model to such a degree that it can only accurately reproduce the set of examples, and thus has lost the property to solve the generalised problem. This is often considered to be detrimental to the model, as it now has few uses outside of recreating data that is already known. An example of overfitting can be seen in Figure 2.3.

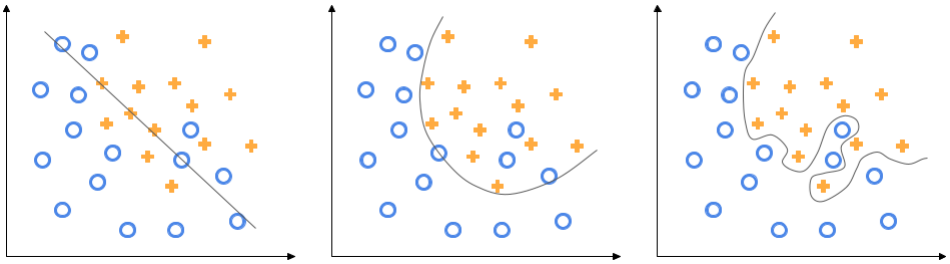


Figure 2.3: On the left a underfitting, middle normal-fitting and overfitting on the right.

2.1.3 Learning rate

How much the parameters in a neural network is changed after a training iteration is not only influenced by the error but also by a *hyperparameter*, or external variable affecting the whole network, called the *learning rate*. In most cases, the learning rate is a fixed value that is carefully chosen for a specific use case to find the global minimum error – too large of a learning rate may make the network completely overshoot a global minimum, while a too small learning rate may make the network get stuck in a local minimum. One common way of remedying this is to have a learning rate that varies over the course of the training, often resulting in a faster convergence and a lower final error.

2.1.4 Inference

When a network has been trained to an acceptable error margin, it is time to actually use it. *Inference* is the process of propagating an input through a neural network until an output is produced – in our use case, the intended output is the colour of a pixel at a specific spot in an image. As mentioned in the introduction to this section, a neural network is a sequence of matrix operations where each layer has its corresponding input, parameters, and output – the output of one layer becomes the input of the next layer, and the dimensions of the final output is specified by the problem that we are trying to solve. Equation 2.1 shows how an input matrix $\boldsymbol{X}$ is processed through each layer by being multiplied with the weight matrix $\boldsymbol{W}$ results in the output $\boldsymbol{Y}$ in Figure 2.1.

$$\begin{aligned} \boldsymbol{X}_{N\times D}, \quad \boldsymbol{W}_{D\times M} \\ \boldsymbol{X}\boldsymbol{W} = \boldsymbol{Y}_{N\times M} \end{aligned} \tag{2.1}$$

2.2 Image Quality Metrics

When discussing the subject of image quality in *lossy* compression, as in where some data is allowed to be lost, one usually points to different errors in the compressed image. In this section, we detail different metrics that can be used to quantify these errors to a measurable value.

There are two approaches to discuss errors in compressed images: the unreferenced approach, and the referenced approach. In an unreferenced context, where there is no ground truth to compare the compressed image to, these errors are seen by an observer as colour shifts, blurriness, grain, Gaussian noise and blocky artifacts [12]. The unreferenced approach is prone to subjective interpretations. In a referenced context where a ground truth is present, there are multiple ways to calculate mathematical metrics that quantify an error pertaining to differences between the compressed image and the true image.
For this reason, there have been multiple mathematical metrics developed to more accurately represent image distortions and how they impact the quality of an image.

2.2.1 Mean Square Error

The *mean square error*, also known as *MSE* or $\boldsymbol{L_2}$ *norm error*, is a pixel-by-pixel metric. The error is calculated using Formula 2.2 where the difference between the result and the reference value is squared. Their sum is then divided by the number of elements. By using the square of the Euclidean distance, large errors will have a proportionally larger impact than smaller errors, making it sensitive to outliers. MSE is an industry standard in error measurements.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \tilde{Y}_i)^2 \tag{2.2}$$

This, however, does not accurately reflect how the human eye perceives errors. The human visual system, shortened to *HVS*, is more sensitive for luminance and colour changes in

texture-less regions [15]. This can result in visual errors that are more or less prominent for the human eye. For example, a human would have trouble discerning the difference between two high-resolution images where one is shifted a single pixel to the right, while the per-pixel error would be enormous – resulting in a false negative pertaining to perceived quality. The MSE metric is indifferent to what kind of error is present – a wrong colour is treated the same as wrong luminance.

2.2.2 Peak Signal-to-Noise Ratio

Peak signal-to-noise ratio, *PSNR*, describes the ratio between a noise-corrupted signal and its original. As seen in Formula 2.3, the metric is based on the MSE where $\boldsymbol{MAX_I}$ is the max pixel value in a range dependant on what data type the image is coded in. Unlike MSE, PSNR is a logarithmic quantity and is thus measured in decibels – the higher a PSNR score, the closer the image is to the original [6].

$$PSNR = 20 \cdot log_{10}(MAX_I) - 10 \cdot log_{10}(MSE) \tag{2.3}$$

According to an analysis of digital image formats and their comparisons by Bull and Zhang, PSNR fits relatively well with subjective assessments despite not being based in human perception. Moreover, they detail that a typical score for lossy compressed images is between 30 to 50 dB. If it exceeds 40 dB, the image is considered very good while below 20 dB is normally considered unacceptable [3].

2.2.3 Structural Similarity

Per-pixel-error based metrics like MSE and PSNR have the fundamental problem of being based in a local context, in contrast being related to structural elements present in regions of an image. To truly give a metric of the human-perceived quality of an image, a different approach is needed.

Structural similarity index measure or *SSIM* is another mathematical metric that tries to encapsulate structural elements in addition to colour and luminance. This makes for a comparison based on certain features of an image rather than the absolute error for each pixel. These features are represented in the functions below, where l stands for luminance, c is for colour and s is for structural elements such as lines. The means and standard deviations in Formula 2.4 are calculated from a Gaussian filter [15]. Despite taking structural elements into account, it has been demonstrated to not fully represent a human perception in relation to patterns of differing luminance [7].

Multi-scale SSIM, often just shortened to *MS-SSIM*, is a continuation of the SSIM method seen in Equation 2.5. It takes in consideration the sensitivity of the human visual system to local structures and possible inconsistencies in the image by comparing the SSIM values of the image at different scales. With careful tuning, this method can outperform SSIM in perceived quality [14].

$$l(\boldsymbol{x}, \boldsymbol{y}) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

$$c(\boldsymbol{x}, \boldsymbol{y}) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

$$s(\boldsymbol{x}, \boldsymbol{y}) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

$$SSIM(\boldsymbol{x}, \boldsymbol{y}) = [l(\boldsymbol{x}, \boldsymbol{y})]^{\alpha} \cdot [c(\boldsymbol{x}, \boldsymbol{y})]^{\beta} \cdot [s(\boldsymbol{x}, \boldsymbol{y})]^{\gamma} \tag{2.4}$$

$$MSSIM(\boldsymbol{x}, \boldsymbol{y}) = [l_M(\boldsymbol{x}, \boldsymbol{y})]^{\alpha_M} \cdot \prod_{j=1}^{M} [c_j(\boldsymbol{x}, \boldsymbol{y})]^{\beta_j} [s_j(\boldsymbol{x}, \boldsymbol{y})]^{\gamma_j} \tag{2.5}$$

2.2.4 ꟻLIP

ꟻLIP is a difference evaluator created by developers from Nvidia, Lund University and Rochester Institute of Technology with the goal of creating a better metric for comparing rendered images with a ground truth [1]. The technique centers around spatial human perception and was developed based on the difference a human would perceive between two alternating pictures laying on top of each other. As shown in Figure 2.4, ꟻLIP has two pipelines that focus on colour and features, respectively. Andersson et al mentions that the combination of these two pipelines create a surprisingly powerful result in comparison to other metrics [1].

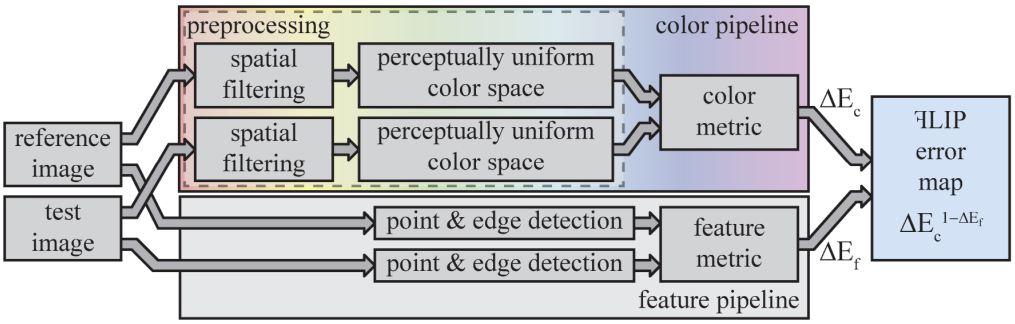


Figure 2.4: The ꟻLIP pipeline [1].

2.3 Fourier Mapping of Coordinate Input

As mentioned in Section 1.4.1, coordinate-based neural networks have a tendency to focus on low-frequency data in an otherwise high-frequency context. This is solved by Tancik et al by mapping the coordinates to the surface of a higher dimensional hypersphere with a randomly sampled set of sinusoids, seen in Equation 2.6 [10]. These sinusoids are also referred to as *Fourier features* in the context of machine learning [9].

The random sampling of the sinusoids are done with a Gaussian matrix denoted as $\boldsymbol{B}$ in Equation 2.7, where the dimension $\boldsymbol{m}$ determines the number of mappings each input coordinate gets. The standard deviation σ describes the frequency band from which the samples are taken, and determines the rate of convergence. These two parameters can be tuned to create

a sufficient approximation of a signal – if σ is too low the approximation will be smooth and lack details, and if too high it will be jagged and noisy.

$$\gamma(\boldsymbol{v}) = [cos(2\pi \boldsymbol{B}\boldsymbol{v}), sin(2\pi \boldsymbol{B}\boldsymbol{v})]^T \tag{2.6}$$

$$\boldsymbol{B} \in \mathbb{R}^{m \times d}, \mathcal{N}(0, \sigma^2) \tag{2.7}$$

2.4 Rendering

Rendering in computer graphics is the process of producing an image from 2D images or 3D models with a program called a *renderer*. Objects in a renderer are organised into graph data structures – also called *scenes* – where hierarchical relationships determine if objects are connected or not.

2.4.1 Rendering Materials

A *material* in computer graphics is a set of values that influence how light is affected when interacting with an object. A material can contain a number of different properties that represents the characteristics of a surface – colour, roughness, and metallic shine are a few examples. The parameters are often stored as one or more images where the specific values are represented by the RGBA values of the image. When discussing textures, it is often necessary to separate a pixel in a texture image from a pixel on the rendering result – for this reason, pixels in a texture are often called *texture elements* or *texels* to differentiate the two.

2.4.2 Deferred Rendering

The simplest method of rendering where for each pixel that is covered by some geometry, the contribution of each light in the scene is calculated individually. This, however, scales very poorly when the number of lights in a scene increases. *Deferred rendering* is a rendering technique that separates the geometry calculations from the light calculations into their own rendering processes, often called *render passes*, which enables one to only calculate illumination for pixels that are actually visible. By first rendering the geometry information into a *geometry buffer*, shortened to *G-buffer*, that information can then be extracted from the buffer at will during future render passes. A deferred renderer can have many passes, or simply one for geometry and one for lighting.

Chapter 3
Method

To answer our research questions the project was divided into three main parts. In the following order, they are to:

- Create a neural network that is trained to replicate textures.
- Use the parameters of the network to create a custom inference that can be used in the rendering pipeline.
- Create a deferred rendering pipeline incorporating the custom inference in real-time.

This chapter will describe our process of exploring different approaches to each subproblem, our motivations for the final methodology, as well as the method of evaluation.

3.1 Neural Network

The goal of the neural network is to replicate a set of textures from a single material given a corresponding input. Since the images all pertain to the same material, they all have common underlying structural elements such as lines and curves. By leveraging a neural network as a function approximator, we use Equation 3.1 to create a set of approximations g_M of the true colour values f_n at the corresponding texel location (u, v). The network will approximate a single material M, where O_M is the number of textures in the material.

$$g_M(u, v) \approx \{ f_n(u, v) \mid n \in O_M \} \tag{3.1}$$

3.1.1 Neural Network Requirements

In order to properly answer the research questions, we must consider the requirements that the neural network model should fulfill. One obvious requirement is the quality of the inferred textures – the model should reproduce the textures in a high enough detail in order

to substitute the original textures. In addition to this, the context of inference in a real-time renderer imposes a significant requirement on the model:
The rendered geometry is not constrained to a fully visible square – it can be unpredictably scaled, rotated, or cropped. Performance is a key factor and we do not want to compute the inference more than necessary, thus we require the model to infer the textures on a per-texel basis.
From these requirements, we define the following constraints for the models:

- *C1:* The model must be able to reproduce the textures in a sufficient quality.
- *C2:* The model must be able to infer on a texel-by-texel basis.

3.1.2 Motivation of Model Choices

Multiple models were examined and tested during the thesis, but not all of them were considered for the final implementation. A detailed description of the other models, their parameters, and their respective results can be seen in Appendix D. This section will go through our motivations for the chosen models and their benefits.

With a naïve model similar in structure to the network in Figure 2.1, it is possible to create a network that corresponds to the approximation g_M. Its strength lie in its simplicity, it does not rely on any additional input other than a single coordinate (u, v) – thus satisfying Constraint *C2*. The structure makes for an easy model to implement, both for the network and inference pass, and has a low memory footprint. The result seen in Appendix A, however, clearly displays the issue of spectral bias and is not detailed enough even after a large number of training epochs, leaving Constraint *C1* unsatisfied.

The idea of using a convolutional autoencoder model was briefly entertained, progressively reshaping a subregion of an image into some coded representation, and then extrapolating them to the resulting image. This, however, was ultimately abandoned due to not conforming to Constraint *C2* – the model had to process a region of coordinates, as opposed to a single coordinate, to infer a correct result.

The *Naïve model* can be improved with the Fourier mapping described in section 2.3. While this increases the input dimensions significantly, it also shifts the network bias from low-frequency features to higher-frequency for a more balanced image representation, solving the issue of spectral bias. A comparison of the unprocessed Naïve model and the *Fourier model* can be seen in Figure A.1. This model satisfies both *C1* and *C2*, and was thus deemed to be a reasonable candidate for the real-time inference.

The autodecoder concept explained in section 2.1.1 can also be combined with the Fourier input mapping to create a model that stores learned values in the trainable latent codes. This enables a smaller MLP network, provided that the latent code values are structured in a manner that enables unique results for each coordinate. The *Autodecoder model* is a modification of the Fourier model and by using the texel coordinates to sample from the latent codes, it satisfies both *C1* and *C2*.

Based on these motivations and the results seen from initial trials in Figure A.1, the chosen models are the Fourier model and the Autodecoder model.

3.1.3 Description of Chosen Models

The chosen models have some common characteristics, since they both are MLP networks made for the same purpose. This section will describe the commonalities, as well as the specific details of both models.

The problem that we are trying to solve involves recreating a single texture channel set of a material from positional input. This implies overfitting is not an issue – rather, it is the objective. For this reason, both models apply a steadily diminishing learning rate through cosine annealing, approaching zero as the training progresses. This ensures that smaller differences in colour can be captured accurately. Both models use ReLU as the activation function throughout the network, with the exception of a final Sigmoid activation to ensure the output is between 0 and 1. The models were trained in PyTorch and the code is available on GitHub[1]. After the training is complete, the network structure and parameters are exported as binary files and moved to the renderer.

The Fourier model starts by converting an input coordinate to a set of sinusoids according to Equation 2.6, resulting in a network input twice the size of the dimension m of the chosen Gaussian matrix. Following this, the Fourier mappings are given to a MLP similar in structure to the network in Figure 2.1, with an output that produces a set of RGB values. An illustration of the model can be seen in Figure 3.1a.

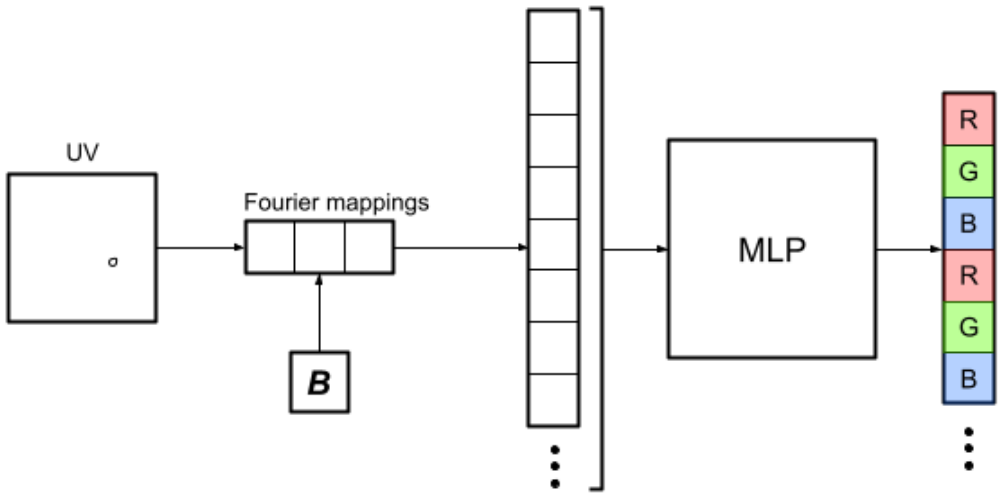
The Autodecoder model combines Fourier mapping and latent codes for the input. To capture both high- and low-frequency details we employ a similar method to Vaidyanathan et al, where two grids containing floating point values are created – one of high and one of low resolution, respectively. These grids are initialised with random values sampled from a normal distribution with a mean of 0.5 and standard deviation of 0.001 and 0.0001 for the high- and low-resolution grid, respectively. The mean and standard deviation values were selected through a simple trial, where the seemingly best values were picked from a few test training runs.

During training, the target texel coordinate (u, v) is scaled proportionally to the two grids and the closest four points are sampled from each of them. For the high-resolution grid, the four points are simply concatenated to the Fourier mapped input; for the low-resolution grid, the four points are bilinearly interpolated to a single value before concatenating. A simple illustration of the Autodecoder model can be seen in Figure 3.1b.

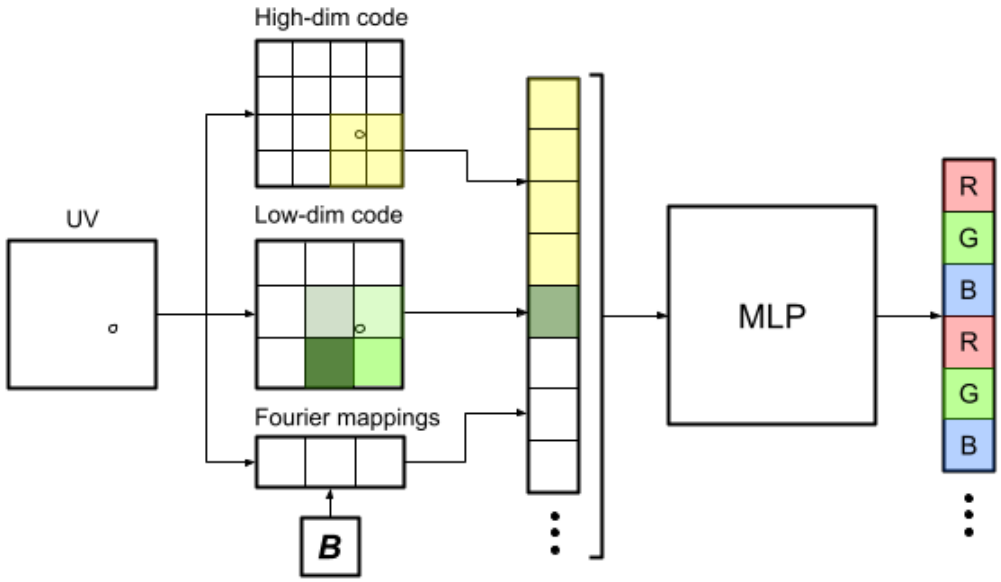
During backpropagation, the weights, biases, and grid values are updated. This allows the network to train the latent codes to represent more details of the image, at the same time as the neurons themselves get better at interpreting the latent codes. By storing learned data in the grids, the size of the network itself can be kept to a minimum.

[1] https://github.com/toveborjeson/exjobb

The Autodecoder model uses two different learning rates – one for the latent code values, and one for the weights and biases. For the last 10% of the epochs, the latent code values are frozen and the network only trains the weights and biases for the remaining training epochs.



(a) Illustration of the Fourier model.



(b) Illustration of the Autodecoder model.

Figure 3.1: Simplified illustrations of the chosen models.

3.2 Inference

As stated in Section 2.1.4, the inference of a linear neural network is a series of matrix multiplications. It is very common to use the GPU for these operations and in this section we will detail methods of leveraging the GPU to perform the inference inside an OpenGL renderer.

Multiple methods for real-time inference were investigated during the project, but we ultimately chose to implement two inference methods for the final methodology. Details about the other investigated inference methods can be found in Appendix C.

3.2.1 Fragment Shader

Since the inference operation is performed on a per-texel basis, using a fragment shader is a very logical option – it requires very little implementation overhead compared to using a CUDA kernel or any other API. The network parameters and Gaussian matrix $\boldsymbol{B}$ can be provided to the shader by using a general storage buffer, in OpenGL 4.3 and onward this is called a *Shader Storage Buffer* or *SSB* and in practice it is similar to how a Uniform Buffer is used. The inference itself is a simple element-wise matrix multiplication implemented with nested for-loops, where the UV-coordinates of the model serve as the base input to be processed before being passed through the network. The shader is only executed for each pixel covered by the target geometry, which means there is no need to cull irrelevant information.

3.2.2 CUDA Kernel

Fragment shaders are primarily used for simpler operations and not matrix multiplications with shared data. By having greater control over the scheduling, execution, and memory use of the GPU we can handle the inference more efficiently. We implemented the same matrix multiplication as in the fragment shader in a custom CUDA kernel, which allows us greater control over the inference. The *CUDA-OpenGL interop* API makes it possible to interact with OpenGL textures and buffers in a CUDA context, which we used to sample the UV-mapping input as well as write to the inferred texture output.

Compared to using the fragment shader approach where the relevant pixels are found with rasterisation, the CUDA kernel has no prior knowledge of relevant pixels – thus, the kernel has a single thread for each pixel on the screen and relevant pixels are manually sorted out. Since the target scene contains nothing else but the relevant objects, it is sufficient to check if the alpha channel of the frame buffer is larger than zero.

3.2.3 Model Specifics

The neural network models are based on the same principle, but differ slightly in what data needs to be provided for their respective input. The common requirements are the weights and biases of the neural network, the Gaussian matrix, as well as the UV-mappings of the

target 3D model – below is a description in how the models differ in relation to the additional inputs required.

Fourier Model

As detailed in Section 2.3, the Fourier model requires the input to be extruded to a certain number of sinusoidal functions before inference can be performed. This requires the same random Gaussian matrix $\boldsymbol{B}$ as present during training, which is exported from PyTorch in tandem with the network parameters. Just before the actual inference, the texture coordinates (u, v) present in the 3D model are extruded to the actual values as shown in Equation 2.6. After this, the input is passed to the network as usual.

Autodecoder Model

The input to the Autodecoder model consist of both Fourier mappings and the concatenated samples from the grids described in Section 3.1.3. By leveraging the grid structure of the latent codes, they can be re-interpreted as two OpenGL textures. For the fragment shader inference specifically, this has the effect of making the latent code values accessible using the OpenGL samplers. For the low-frequency code, the built-in bilinear interpolation sampler *GL_LINEAR* is used; while for the high-frequency code a simple closest texel fetch, *GL_NEAREST* sampler, is used.

For the CUDA inference, they are treated as sets of 32-bit floats, where the grid structure is flattened to a single dimension. There is no implicit way to sample with bilinear interpolation as in the shader, thus a custom bilinear interpolation is used in the same manner as during training.

3.3 Rendering

In this section, details regarding the integration of the inference methods into a deferred rendering pipeline are provided. We make use of the *bonobo* framework from the courses EDAN35 and EDAF80, with some slight modifications to accommodate the model inference[2]. Before the rendering loop is started, the model parameters and additional data is parsed from the binary files and bound to their respective locations.

Scene

The rendering scene is a simple quad created in Blender, with the *sponza_details* textures bound to the corresponding material channel. This creates a direct baseline for comparison with the inferred textures, which can be toggled in the GUI. An example of the renderer in use regular texture mapping can be seen in Figure 3.2.

[2]https://github.com/LUGGPublic/CG_Labs

Geometry Buffer Split

In a regular deferred renderer, the G-buffer contains all geometric data as well as the provided material textures of a 3D model. In our case, we need the UV-coordinates specifically as input to the networks, and from those create the material textures in real-time. As can be seen in Figure 3.3, we separate the UV-coordinate fetching from the geometry pass to an inference pre-pass, and either pass it to CUDA or to the inference fragment shader depending on the desired inference method. The UV-mappings are also provided to the regular geometry pass in tandem with the inferred textures.

After inference, the material textures are constructed and written to a framebuffer – meaning that the geometry render pass was also modified to accommodate for both sampling a regular texture input and sampling from a framebuffer. After this there are no substantial differences in how the rendering is performed compared to a regular deferred renderer – most of the complexity, both computational and in implementation, is located in the inference itself.

3.4 Evaluation

To answer the research questions we need to discern the way we evaluate the texture quality, memory allocation size and inference time. This section will describe the method of obtaining the metrics needed for these aspects.

3.4.1 Texture Quality

To answer RQ1 we also need to establish a standard of image quality that can be compared between the different models. Deciding on a measurable metric for the inferred texture quality is a non-trivial task due to the inherent subjectiveness of the human visual system. For this reason, we will use all of the metrics detailed in Section 2.2 since each metric is sensitive to different kinds of errors. This is done in order to represent the image quality as broadly

Figure 3.2: Example of the rendered scene with regular texture mapping.

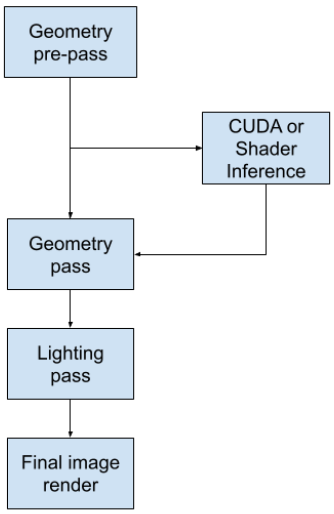


Figure 3.3: Flowchart illustrating the modified deferred renderer

as possible. These metrics will be applied to the different tests to determine the optimal specifications for the final models, which will be compared. The tests include the amounts of neurons and the use of latent code in the Autodecoder model and Fourier mappings.

3.4.2 Storage Size

To answer RQ2, a measurement of the storage size for each model is required. In this section we will establish how the size of each model is calculated. Note that the equations calculate a purely logical size S expressed in bytes – the true size on a disk depends on the block size of the file system. All parameters and representations are saved as 32-bit floating point values, meaning a single parameter is four bytes in total.

Equation 3.2 and Equation 3.3 shows the logical size calculations for each model. In the equations, N represents the number of neurons, L the number of hidden layers, and O_M the number of texture channels in the material. The matrix dimensions m and d comes from the dimensions of the specific Gaussian matrix $\boldsymbol{B}$ chosen for the model. The total parameter size P varies between the models due to the different inputs provided, thus they are denoted P_f and P_a for the Fourier and Autodecoder model, respectively.

Additionally, a description of the model hyperparameters are also exported. This includes the number of hidden layers and neurons, and the respective input and output numbers it is configured for. These descriptive values are represented as four 32-bit integers, added to the total sizes in equations 3.2 and 3.3.

$$\begin{aligned} P_f &= N \cdot ((2 \cdot m) + L \cdot N + 3 \cdot O_M + L + 1) + 3 \cdot O_M \\ G &= m \cdot d \\ S_f &= 4 \cdot (P_f + G + 4) \end{aligned} \tag{3.2}$$

The Autodecoder model uses the latent code grids as additional parameters, whose proportions are identical to the width and height of the material textures in question. These are represented as $\boldsymbol{C}_h$ and $\boldsymbol{C}_l$ in Equation 3.3 for the high- and low-dimension code, respectively.

$$\begin{aligned} P_a &= N \cdot ((2 \cdot m + 5) + L \cdot N + 3 \cdot O_M + L + 1) + 3 \cdot O_M \\ C &= C_{lx} \cdot C_{ly} + C_{hx} \cdot C_{hy} \\ S_a &= 4 \cdot (P_a + G + C + 4) \end{aligned} \tag{3.3}$$

3.4.3 Measuring Inference Time

To answer RQ3 we need to measure the time it takes for the inference to be performed. OpenGL provides this functionality for renderers through the *GL_TIME_ELAPSED* query, which can measure the elapsed time between the start and end of a designated section of the rendering code. To mitigate outliers, the mean value of 50 frame time samples are considered to be representative of the inference time.

Since the inference is done per-pixel, the performance of the renderer is directly tied to the number of pixels to render – therefore, the rendering program is run in a window of a fixed size of 1600 by 900 pixels to provide a stable baseline for comparison. Inference time measurements are taken from the worst-case scenario where the whole rendering screen is covered by the target geometry, resulting in an inference time measurement on all 144 000 pixels executed in parallel.

Chapter 4
Results

In this chapter we will present the results according to the evaluation methods detailed in Section 3.4. The two models that could produce a comparable result were the Fourier and Autodecoder. Figure 4.1 shows the best configurations for these and their results as the images from PyTorch. The ꟻLIP error map is also included to showcase the problem areas for the individual models in the form of a heatmap as well the ground truth image. Here we can see that it is mainly in the transitions and edges between the larger structures that the errors occur.

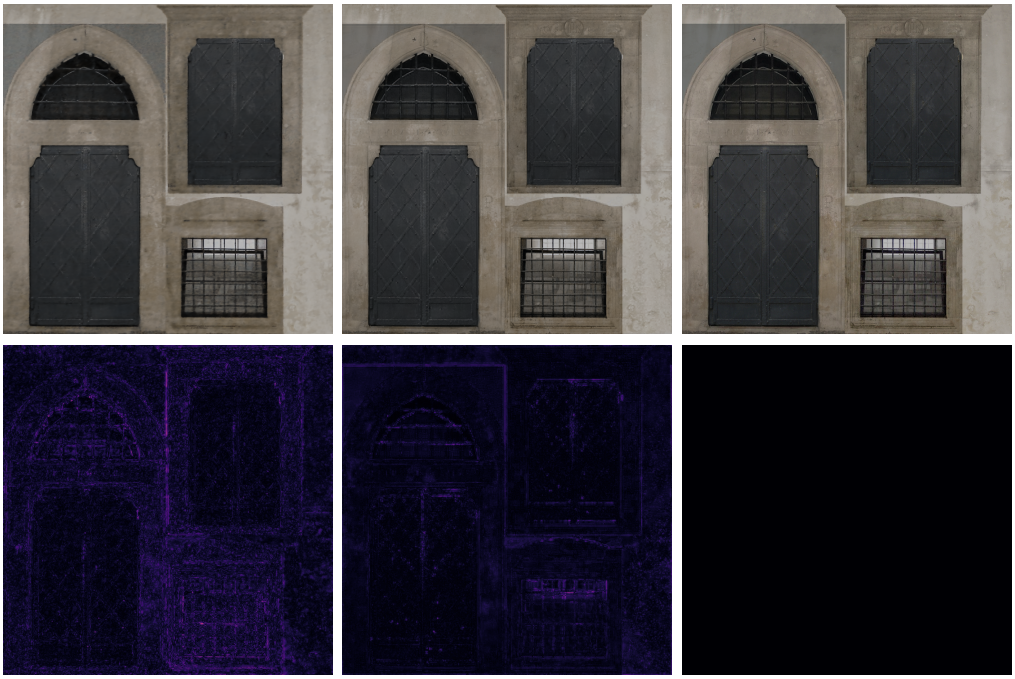
The following results were produced by a computer equipped with a NVIDIA GeForce RTX 4060Ti 16GB GPU, as well as a 13th Gen Intel Core i7-13700KF 3.40 GHz CPU. The training and rendering were performed on Windows where the former was done on the *Windows Subsystem for Linux*.

All networks were trained for 6000 epochs and compared to the *sponza_details* image set present in the *CG_Labs* repository[1]. An example of the neural network models in comparison to the texture set images can be seen in Appendices A and B. The texture set has a combined storage size of 3 246 278 bytes and the mean render time with a regular texture lookup is 0.477440 milliseconds. The code for the renderer is published on GitHub[2].

4.1 Fourier Model

For all Fourier models we use an initial learning rate of 0.005, and $\sigma = 10.0$ was found to be a suitable Gaussian standard deviation. The results from the tests run on the Fourier model are listed below.

[1] https://github.com/LUGGPublic/CG_Labs
[2] https://github.com/filvan1/neural_compression



(a) Fourier model (b) Autodecoder model (c) Ground truth

Figure 4.1: Example result of models in PyTorch with corresponding ꟻLIP heatmaps.

4.1.1 Gaussian Dimensions

Table 4.1 shows how image quality metrics change as the Gaussian matrix dimension increases in a Fourier model with 402 neurons. According to the acceptable limit of PSNR mentioned in Section 2.2.2 all dimensions of the Fourier matrix produce results that exceeds it. The normal texture channel is the least affected of changes in the dimension. All the metrics are in agreeance that the 12×2 were the most accurate and is marked in bold in the table. The only real outlier is the 4×2 dimension. It had overlaying artefacts similar to aliasing throughout all the channels that resulted in the poor performance. The nature of errors of the normal channel resulted in that this artefact were not as deviant as the ones in the diffuse and specular. This is because the normal defaults to produce an almost entirely monotone purple image that is more akin to the ground truth than if the others did the same.

Table 4.1: Table showing the Gaussian dimension to image quality and size of a Fourier model with 402 neurons in four layers.

	Image Quality											
***dim*(*B*)**	**100·MSE (↓)**			**PSNR (↑)**			**1-MSSIM (↓)**			**ꟻLIP (↓)**		
	D	N	S	D	N	S	D	N	S	D	N	S
4×2	0.617	0.0804	0.772	22.10	30.95	21.13	0.279	0.0883	0.337	0.229	0.0867	0.230
8×2	0.0712	0.0796	0.180	31.48	30.99	27.44	0.0881	0.0870	0.131	0.0987	0.0890	0.103
12×2	**0.0419**	**0.0638**	**0.116**	**33.78**	**31.95**	**29.35**	**0.0530**	**0.0780**	**0.0818**	**0.0758**	**0.0840**	**0.0794**
16×2	0.0431	0.0693	0.118	33.65	31.59	29.29	0.0537	0.0823	0.0884	0.0762	0.0848	0.0809
20×2	0.0529	0.0761	0.153	32.76	31.19	28.16	0.0661	0.0870	0.111	0.0854	0.0881	0.0933

In Table 4.2 we can see the impact of the Gaussian matrix dimensions on the inference times of a Fourier model. We chose 256 neurons for this test while 402 had problems with stopping the rendering because of too long inference times. This impeded the ability to run extensive testing.

Table 4.2: Comparison of inference time between Gaussian matrix dimensions for a Fourier model with 256 neurons, four layers.

$dim(\boldsymbol{B})$	**Shader inference** (ms)			**CUDA inference** (ms)		
	Inference	Rendering	Total	Inference	Rendering	Total
4×2	1325.266235	1.63562	1326.901855	538.960754	0.4917	539.452454
8×2	1330.887207	2.026489	1332.913696	567.286743	0.412537	567.699280
12×2	1348.277954	1.598633	1349.876587	536.417969	0.511902	536.929871
16×2	1066.792847	1.770019	1068.562866	533.542725	0.450805	533.993530
20×2	1348.094238	0.972046	1349.066284	532.615234	0.43097	533.046204

4.1.2 Number of Neurons

Table 4.3 shows how image quality metrics change as the number of neurons increases in a Fourier model with four layers, $dim(\boldsymbol{B}) = 16 \times 2$. The largest number of neurons in this configuration that is still considered a compression is 402. It is also the highest image quality recorded for this test in consideration of all the different metrics. This is of specific interest and is marked bold in the following tables.

Table 4.3: Table showing the number of neurons to image quality and size of specific Fourier.

	Image Quality												**Model size (B)**
Neurons	**100·MSE (↓)**			**PSNR (↑)**			**1-MSSIM (↓)**			**ꟻLIP (↓)**			
	D	N	S	D	N	S	D	N	S	D	N	S	
32	0.833	0.0854	1.790	20.79	30.68	17.47	0.505	0.0925	0.590	0.294	0.0966	0.337	22 452
64	0.360	0.0833	0.698	24.44	30.79	21.56	0.343	0.0900	0.454	0.201	0.0905	0.216	77 492
128	0.196	0.0827	0.443	27.08	30.82	23.53	0.227	0.0891	0.306	0.152	0.0888	0.161	285 876
256	0.125	0.0825	0.328	29.02	30.84	24.84	0.128	0.0889	0.211	0.116	0.0888	0.134	1 095 860
402	**0.0431**	**0.0693**	**0.118**	**33.65**	**31.59**	**29.29**	**0.0537**	**0.0823**	**0.0884**	**0.0762**	**0.0848**	**0.0809**	**2 659 812**
512	0.104	0.0821	0.292	29.82	30.86	25.35	0.0993	0.0887	0.173	0.102	0.0889	0.120	4 288 692

Table 4.4 shows the inference times for each tested neuron amount with the configuration $dim(\boldsymbol{B}) = 16 \times 2,\ \sigma = 10.0$. Note that we could not measure the shader inference for 402 neurons due to the renderer crashing when this was attempted. A benchmark that is often used for usable rendering of images is 16.67 ms, or 60 frames per second, so 32 neurons is the only one that could meet this limit. This benchmark is further discussed in Section 5.3.

Table 4.4: Comparison between rendering methods for the Fourier model with the configuration $dim(\boldsymbol{B}) = 16 \times 2,\ \sigma = 10.0$.

	Shader inference (ms)			**CUDA inference** (ms)		
Neurons	Inference	Rendering	Total	Inference	Rendering	Total
32	6.833193	0.595167	7.428360	7.324630	0.23371	7.558340
64	63.308800	0.924659	64.233459	21.964720	0.252321	22.217041
128	239.674225	1.199859	240.874084	76.610214	0.281151	76.891365
256	1065.924194	1.23816	1067.162354	528.694214	0.481506	529.175720
402	-	-	-	**8207.384766**	**1.044922**	**8208.429688**

4.2 Autodecoder Model

For the Autodecoder model we found that a Gaussian standard deviation of $\sigma = 5.0$ was the most suitable. Additionally, we initialised the latent codes with values sampled from a normal distribution with mean $\mu = 0.5$ and standard deviation $\sigma = 0.001$. For all networks trained, the learning rate for the weights and biases had an initial learning rate of 0.005, while the latent code values had an initial learning rate of 0.1.

4.2.1 Gaussian Dimensions

The impact of the Gaussian matrix dimensions were not as significant concerning image quality in the Autodecoder model as in the Fourier model. This can be seen in Table 4.5. Which dimension produced the highest image quality is wildly spread across the dimensions and the best values are marked in bold.

Table 4.5: Table showing the Gaussian dimension to image quality and size of a Autodecoder model with 64 neurons, two layers, 768 high-dimensional code, and 256 low-dimensional code.

	Image Quality											
$dim(\boldsymbol{B})$	**1000·MSE (↓)**			**PSNR (↑)**			**1-MSSIM (↓)**			**ꟻLIP (↓)**		
	D	N	S	D	N	S	D	N	S	D	N	S
4×2	0.126	0.185	0.231	38.99	37.33	36.36	0.00877	0.0199	0.00510	0.0435	**0.0541**	0.0288
8×2	0.131	0.205	0.252	38.83	36.89	35.98	0.00893	0.0207	0.00536	0.0436	0.0563	0.0295
12×2	0.127	**0.183**	0.233	38.96	**37.38**	36.32	0.00870	0.0201	0.00507	**0.0432**	0.0566	0.0290
16×2	0.119	0.192	0.237	39.23	37.16	36.25	0.00894	0.0221	0.00497	0.0436	0.0586	0.0289
20×2	0.143	0.197	**0.228**	38.46	37.07	**36.42**	0.00870	**0.0192**	0.00516	0.0438	0.0562	**0.0287**
32×2	**0.115**	0.184	0.235	**39.38**	37.35	36.29	**0.00869**	0.0211	**0.00494**	**0.0432**	0.0579	0.0292

We further see that in Table 4.6 that the Gaussian dimensions have a proportionally smaller but still significant impact on the inference times of a similar Autodecoder model. The CUDA inference all has times measured under the 16.67 ms benchmark while the shader measurements are all over.

Table 4.6: Comparison of inference time between Gaussian matrix dimensions for an Autodecoder model with 64 neurons, two layers, 768 high-dimensional code, and 256 low-dimensional code.

$dim(\boldsymbol{B})$	**Shader inference** (ms)			**CUDA inference** (ms)		
	Inference	Rendering	Total	Inference	Rendering	Total
4×2	32.432266	0.711895	33.144161	11.895890	0.28851	12.184400
8×2	32.068565	0.811051	32.879616	12.174150	0.292912	12.467062
12×2	31.801262	0.78154	32.582802	13.003284	0.288956	13.292240
16×2	27.983953	0.785206	28.769159	13.568225	0.321938	13.890163
20×2	28.870411	0.789452	29.659863	14.061773	0.341727	14.403500

4.2.2 Number of Neurons

Table 4.7 shows that the increase of number of neurons generates albeit a small improvement, but does not ultimately significantly impact the image quality of the Autodecoder when the latent code values are fixed. 1024 was tried for the Autodecoder but this was not possible to train with our current implementation due to lack of GPU memory. From the table we can see that it is only the 512 neurons that exceeds the compression mark of 3.2 MB.

Table 4.7: Metrics for the number of neurons spanning from 64 to 1024 for an autodecoder with configuration $dim(\boldsymbol{B}) = 8 \times 2$, high resolution code dimension 768, low resolution code dimension 256. D stands for diffuse texture channel, N for the normal and S for specular.

Neurons	**Image Quality**												**Model size (B)**
	1000·MSE (↓)			**PSNR (↑)**			**1-MSSIM (↓)**			**ꟻLIP (↓)**			
	D	N	S	D	N	S	D	N	S	D	N	S	
64	0.131	0.205	0.252	38.83	36.89	35.98	0.00893	0.0207	0.00536	0.0436	0.0563	0.0295	2 662 788
128	0.129	0.201	0.251	38.88	36.96	36.00	0.00876	0.01917	0.00531	0.0433	0.0565	0.0298	2 769 540
256	0.114	0.156	0.252	39.44	38.06	35.98	0.00814	0.01879	0.00496	0.0409	0.0553	0.0295	3 179 652
512	0.101	0.144	0.220	39.94	38.40	36.58	0.00743	0.01704	0.00453	0.0393	0.0527	0.0280	4 786 308

4.2.3 Latent Code Dimensions

Table 4.8 shows how image quality metrics and size changes with the latent code dimensions in an Autodecoder model with $dim(\boldsymbol{B}) = 16 \times 2$, 64 neurons and two layers.

From this we measure the inference times of a subset of the valid code dimensions. Of certain interest is the 768x256 configuration, as it is the highest quality that is still a compression. This configuration is marked as bold in Table 4.9.

Table 4.8: Table showing how latent code dim influences image quality and storage size. Metrics for the latent code grid dimensions spanning from 128/32 to 1024/768 for an autodecoder with configuration 64 neurons, $dim(\boldsymbol{B}) = 8 \times 2$ and σ = 5.0. D stands for diffuse texture channel, N for the normal and S for specular.

Latent Code Grids		Image Quality												Model size (B)
		1000·MSE (↓)			PSNR (↑)			1-MSSIM (↓)			FLIP (↓)			
High res	Low res	D	N	S	D	N	S	D	N	S	D	N	S	
128														
	32	0.987	0.787	2.70	30.06	31.04	25.69	0.0977	0.0828	0.161	0.0985	0.0871	0.115	110 980
	64	0.858	0.772	2.37	30.66	31.13	26.26	0.0867	0.0819	0.140	0.0945	0.0882	0.107	123 268
256														
	64	0.555	0.675	1.60	32.56	31.71	27.95	0.0448	0.0691	0.0637	0.0733	0.0807	0.0767	319 876
	128	0.468	0.588	1.35	33.30	32.30	28.70	0.0380	0.0723	0.0541	0.0688	0.0825	0.0706	369 028
512														
	64	0.318	0.372	0.811	34.98	34.29	30.91	0.0156	0.0307	0.0122	0.0531	0.0614	0.0427	1 106 308
	128	0.274	0.350	0.683	35.62	34.56	31.66	0.0141	0.0269	0.0119	0.0496	0.0594	0.0409	1 155 460
	256	0.211	0.369	0.509	36.75	34.33	32.93	0.0124	0.0282	0.00999	0.0473	0.0626	0.0374	1 352 068
768														
	64	0.174	0.208	0.417	37.59	36.83	33.80	0.00920	0.0196	0.00568	0.0445	0.0536	0.0323	2 417 028
	128	0.158	0.227	0.353	38.00	36.44	34.52	0.0101	0.0213	0.00660	0.0468	0.0579	0.0335	2 466 180
	256	**0.131**	**0.205**	**0.252**	**38.83**	**36.89**	**35.98**	**0.00893**	**0.0207**	**0.00536**	**0.0436**	**0.0563**	**0.0295**	**2 662 788**
	512	0.0799	0.156	0.0909	40.97	38.08	40.41	0.00622	0.0161	0.00213	0.0397	0.0520	0.0238	3 449 220
1024														
	64	0.0651	0.0817	0.0405	41.86	40.88	43.93	0.00774	0.0110	0.00255	0.0424	0.0477	0.0250	4 252 036
	128	0.0577	0.0926	0.0454	42.39	40.33	43.43	0.00536	0.0120	0.00159	0.0378	0.0489	0.0229	4 301 188
	256	0.0496	0.0694	0.0376	43.05	41.59	44.25	0.00501	0.00841	0.00218	0.0367	0.0446	0.0230	4 497 796
	512	0.0419	0.0549	0.0250	43.77	42.60	46.01	0.00394	0.00663	0.000942	0.0343	0.0406	0.0218	5 284 228
	768	0.0335	0.0338	0.0113	44.75	44.72	49.46	0.00269	0.00268	0.000391	0.0329	0.0391	0.0217	6 594 948

Table 4.9: Comparison between rendering methods for a 64-neuron, 2-layer Autodecoder model with the configuration $dim(\boldsymbol{B}) = 8 \times 2$, $\sigma = 5.0$.

Latent Code Grids	Shader inference (ms)			CUDA inference (ms)		
High, Low	Inference	Rendering	Total	Inference	Rendering	Total
256, 128	32.013371	0.814331	32.827702	12.142941	0.264019	12.406960
512, 256	32.013966	0.830654	32.844620	12.143495	0.402505	12.546000
768, **256**	**32.068565**	**0.811051**	**32.879616**	**12.174150**	**0.292912**	**12.467062**
768, 512	31.934525	0.788936	32.723461	12.150865	0.270395	12.421260
1024, 768	31.997766	0.801779	32.799545	12.212222	0.298959	12.511181

Chapter 5

Discussion

In this chapter we will discuss the results presented in Chapter 4 and determine the best model based on our research questions.

5.1 RQ1: Reconstructed Image Quality

What reconstructed texture quality is achievable using a neural network?

To answer this question we have to decide what we define as high quality and what metric we will prioritise highest. In Chapter 2 we bring up both non-reference and reference based metrics. There are a lot of reconstructed image results that looked good and alike so that is why we pressed on the reference based metrics to get some level of objectivity. During the tests we saw that the ꟻLIP and MSE usually trends the same where as MSSIM had some differences. PSNR is based on MSE so it naturally follows its peaks and valleys. We saw that MSSIM values structures highest and leaves colour second.

The best image quality that we could measure was for the Autodecoder model with resolution grids 1024/768 with the mean of the three texture channels being MSE: $2.62 \cdot 10^{-5}$, PSNR: 46.31, 1-MSSIM: 0.00192, ꟻLIP: 0.0313. The models in Table 5.1 are both the best performing of their kind while also having a level of compression. Among the two, the Autodecoder with resolution grids 768/256 still dominates while having 7 times the error of the 1024/768. Subjectively we visually deemed it as good and when looking at the PSNR score the 768/256 still ends up in the acceptable range.

Table 5.1: Comparison of the mean metrics of the three texture channels between the two discrete models, Fourier and Autodecoder, with their corresponding best configurations.

Model	Mean Image Quality				Network Size	Inference Time
	100·MSE	PSNR	1-MSSIM	ꟻLIP		
Fourier	0.0739	31.70	0.0710	0.0798	2 646 916	536.929871
Autodecoder	0.0181	37.56	0.0112	0.0421	2 660 708	11.895890

There are some factors that we have seen contribute strongly to the quality of an image. Below some of these will be discussed.

5.1.1 Fourier Mapping

We see that the Fourier mapping of the UV-coordinate input increases the image quality for both models, but the specific impact varies between the two. For the Fourier model we saw that this parameter could improve the final result significantly. In Table 4.1 we can see that the largest improvement occurred in the lower ranges, and converged around 12 and 16. All tests concluded that one dimension was superior to the others.

We did not see a similar steep improvement for the Autodecoder model between the dimensions, although excluding the Fourier mappings altogether resulted in a worse result. Because of this, we decided to keep the Fourier mapping in the model. There was not a clear leader when comparing the measurements in Table 4.5 and the value variation was low, thus we chose one that seemed fine at that moment to continue testing with. It was not until later when conducting further tests of the dimensions that 4×2 and 32×2 were discovered as a slightly better option in regards to the mean. At that point, we did not have time to render and print the results again – resulting in the rendered images in Appendix B to be of the dimension 16×2. The metrics in Table 5.1 are of the 4×2 due to its superior inference time compared to 32×2.

Regarding the standard deviation, its main contribution was to fine-tune the grain and blur in the image. We did a small venture and tried different values. Afterwards we set the Autodecoder to 5.0 and the Fourier to 10.0. These configurations were not optimised to their full extent but were deemed good enough when we did not discern any apparent problems. This could mean we hit a local extremum, but further tests were not made.

5.1.2 Number of Neurons

A parameter that had a significantly sizeable impact on the image quality was the number of neurons in each hidden layer of any model. With lower amounts of neurons, the image become blurry and smoothed out while increasing the number resulted in distinguished details and colours. This parameter sets a kind of ceiling for the image quality. While the different models require different amounts of neurons, we see that issues with image quality can quickly be remedied by increasing this parameter. It is a quick-and-easy fix but larger amounts may severely impact the other aspects negatively, as shown in the following Section

5.2.

The Fourier model show the most improvement regarding the increase of the number of neurons. Between 32 to 402 there is a decrease by roughly 14 and 18 times for the MSE, seen in Table 4.3. The only texture channel that did not see this improvement trend is the normal and this is because of the nature of this texture as mentioned earlier. As it is mainly one solid colour without major detailing, it will skew the comparison somewhat. This phenomenon is most prevalent for the Fourier model as it struggles the most with details at lower neuron amounts. The normal of the Autodecoder model follows the same improvement trend as the other channels in Table 4.7 as it already produced sharp images at the lowest measured value 64 neurons.

5.1.3 Use of Latent Code

During our tests we saw that the use of latent code in the Autodecoder network resulted in the capture of more details as the grids increased in size. This factor in the network follows the same observed rule where more and bigger is better when optimising for the image quality. In Table 4.8 we see that the larger grid dimensions produces better results. The combinations containing the high-resolution grid 1024 all has PSNR scores over 40 dB and even 49.46 dB, which is considered high. The changes between the high-resolution grid are the ones that yield the largest increase of image quality. Even so, the increase of the low resolution grids still give significant improvement.

5.2 RQ2: Level of Compression

What is the level of compression?

When evaluating the level of compression, we compare the total amount of memory storage of the render material – that being the three texture channels – to the total of all the output files of a neural network model. We use a single network to learn the three different textures at once – this has a negligible impact on the network size itself, as seen with the O_M factor in equations 3.2 and 3.3. This is a deciding factor in compressing the images, and the more output images are approximated by the network, the better the compression factor will be. While we see that it is not viable for compressing a single image, it might be able to compress multiple texture channels for one material. When adding more texture channels we saw a slight dip in image quality but this could be compensated with a longer training time to achieve the same result.

The output files of the model contain the binary representations of all neural network weights and biases, the Gaussian matrix $\boldsymbol{B}$, and in the case of the Autodecoder model, the latent code values. The input images amount to 3 246 278 bytes in total and any value below this is considered a compression. Considering compression without putting it in relation to other aspects of the result is not very productive – otherwise one could simply conclude that a smaller network is better. In general, we can see a trend of larger storage size indicating a

higher quality result for the texture. In this section, we detail the factors that we have seen contribute strongly to the level of compression.

5.2.1 Number of Neurons

One of the most influential parameters for the network size is the number of neurons, as this value becomes a quadratic term in equations 3.2 and 3.3. This implies that keeping the number of neurons low is the most effective way of reducing the required storage size.

In Tables 4.3 and 4.7 we compare the number of neurons to the resulting image quality and size of each respective model. In the case of the Fourier model, we can see that a higher number of neurons indicate a higher quality reconstruction – this is less evident for the Autodecoder model, where gains are relatively minor in comparison. This indicates that the quality-to-size ratio of the Autodecoder is not mainly influenced by the number of neurons.

5.2.2 Latent Code Dimensions

As the number of neurons in an Autodecoder model only increased the storage size dramatically without affecting the image quality significantly, we now analyse the impact of the latent code values. We can see in Table 4.8 that there is a positive correlation between the dimensions of the latent code grids compared to the image quality. Furthermore, the table also illustrates the resulting storage size required for a specific configuration. As the latent codes are represented as square textures, an increase in dimension results in a quadratic growth of the number of values. In particular, we can compare the image quality metrics to results of a similar model size in Table 4.7 to see that latent codes of larger dimensions are more important than the number of neurons for the Autodecoder model in both storage and quality.

5.3 RQ3: Real-time inference

How fast is the inference in a real-time context?

In the context of video games, a frame latency of 16.67 milliseconds or 60 frames per second is considered good while frame times above 100 ms has been found to be almost unplayable in a study by Claypool et al [4]. Furthermore, they show that frame rate has a high impact on the perceived experience and performance of players in computer games – especially if the frame rate varies greatly. Many applications therefore aim for a stable 60 frames per seconds at all times, resulting in a single render pass to have a budget of a few milliseconds or less. In this section, we analyse the impact that the different model configurations on the total render time.

5.3.1 Network Configurations

The inference time for both models vary between the different configurations. From Table 4.4 we can see that the inference time increases dramatically when the number of neurons in a four-layer network is expanded – in the case of 402 neurons per layer it increased to such

an extent that the renderer itself crashed. Due to the basic MLP structure that both models are based upon, we can assume that the inference times for a two-layer Autodecoder model grows in a similar fashion when the number of neurons increases. Since the implementation consists of nested for-loops, this result is not very surprising.

The impact of the Gaussian matrix dimensions on the inference time is still largely bound by the corresponding number of neurons in the network. In Tables 4.2 and 4.6 we see that for both models, the Gaussian matrix does still impact the performance in a small albeit significant manner. This is in accordance to the structure of the MLP, where each additional input neuron results in a new set of weights equal to the number of neurons in the subsequent hidden layer. One interesting thing to note in the tables is that the shader inference becomes quicker as the Gaussian matrix dimensions approaches 16×2. We are unsure why this is the case – we theorise that the resulting 32 floating point values aligns with some cache size in the OpenGL implementation, making it easier to share them between the pixel shader accesses.

The latent code dimensions of the Autodecoder model has no discernible impact on the resulting inference time, as we can see by comparing the inference times in Table 4.9 with the corresponding image quality metrics and storage size in Table 4.8. The variance is so small that it could even depend on the momentary state of the computer or chance. This can be very advantageous when deciding on a configuration of the model – image quality and storage size are then decided by the code dimensions, while the inference time remains largely similar. This result further strengthens the Autodecoder model as a preferred method of using neural networks for real-time texture inference.

5.3.2 Inference Method

From both Table 4.4 and Table 4.9 we can see that using CUDA for the real-time inference generally results in the quicker inference times. Before implementing the CUDA kernel, we were in doubt that the gains of using a CUDA kernel would be offset by the cost of switching context on the GPU from OpenGL and back. Looking at the top row of Table 4.4, we can see the only case where the shader inference was quicker than the CUDA inference – we hypothesise that this is either due to the overhead for launching a CUDA kernel, or due to the OpenGL-CUDA context switch.

As mentioned in Section 3.2, we use a very naïve approach to the matrix multiplications required for the inference with nested for-loops. This is considered to be a very slow method of matrix multiplication with the GPU, and does not leverage the memory structure of the GPU properly – comparisons have indicated that a naïve for-loop implementation is orders of magnitude slower than a cuBLAS implementation [2].

5.4 Practical Viability

We found it difficult to determine one optimal model, as one must weigh many equally important parameters and aspects that are deeply entwined. The inference time has to be as low as possible, while there has to be a compression present to even be considered a beneficial

method in the use case of video games and other real-time applications. This, in combination with the need to have a fundamental level of image quality, results in multiple simultaneous priorities.

In short, from the results in the previous chapter we have concluded that the Autodecoder with latent code dimensions of 768 high- and 256 low-resolution to be the most viable candidate. This model and its parameters sum to a total of 2 662 788 bytes – a reduction of 17.97% compared to the material texture set size.

The Fourier model has advantages, particularly its relative simplicity implementation-wise. However, the strong correlation between the number of neurons to the image quality means that one can not improve the model without also impacting the size and inference time – thus rendering it unsuitable for a real-time context.

The methods we present are not in and of themselves a substitute for regular deferred rendering – but we do see potential in using neural features to create textures in real-time to save storage space.

5.5 Limitations

During our thesis we encountered some limitations that could possibly impact the final results. Below the main ones are discussed.

5.5.1 Test Case Selection

Due to the number of possible cases when testing every parameter, loss function, and model, the sheer amount of data and time required to process it made it nearly impossible to test everything. Furthermore, the implementation of the inference requires a small amount of hard-coded values related to the allocation of memory on the GPU – in the current version, this can not be automated and was thus done manually for each measurement. This resulted in fewer measurements for the inference due to time constraints. After seeing trends from initial sweeping trials, we chose the models we believed would perform best at the moment while being balanced representation of all the different parameters. The test cases themselves were judged based on their ability to produce contributory results as well as interesting discussion points.

5.5.2 Rendering Artefacts

The real-time inference results from the Autodecoder model contain artefacts such as bright spots, colour smearing, and blockiness compared to the PyTorch output – this can be seen in Appendix B. We theorise this is due to differences in bilinear interpolation between the different platforms, but we did not investigate this further due to time constraints. For this reason, our results of model inference time are not considered to be strictly adherent to their respective image quality – this is due to the real-time inference not producing the same

output as in PyTorch. However, we do consider the measurements to be indicative of how the network inference time changes with respect to quality and storage size.

Chapter 6
Conclusion

To summarise, we have investigated the use of neural networks in inference of textures in real-time – particularly how their configurations and parameters impact the resulting image quality, storage size, and inference time. We established two neural network models and their architectures, how they differ, and how their configurations can shape the results. What model configuration is most suitable for a real-time application is dependent on the intended purpose. Overall, it is a compromise between image quality, storage space, and inference time – while it is possible to tune the networks to favour a specific target, we do not consider the methods proposed to be a substitute of regular texture lookup rendering. For this to be viable, the networks would need to be smaller, faster, and the inference result more detailed – still, we see potential for future iteration and development.

6.1 Future Work

In this thesis we have explored a specific application of neural rendering, but the results are far from practical in and of themselves. For this reason, we consider the methods and implementations described as a starting point for future studies – below we detail viable avenues for this, both regarding questions and methods that we did not investigate, as well as areas of our implementation that can be improved.

6.1.1 Further Studies

Different Textures and Mipmap Levels

As mentioned in Section 1.2.1, we focused solely on albedo, normal, and specular maps. Further studies might find that the network models are more suitable for other kinds of textures. Furthermore, Vaidyanathan et al compares outputs at different mipmap levels while in this

thesis we only investigated a single level – it might be possible to have the level of detail gradually decrease through a diminishing Gaussian matrix depending on distance to the object.

Multi-material Network

In this thesis we test a single network corresponding to a single material. In real-time rendering applications such as video games, there can be hundreds of materials present in a single scene. There might be ways to train a neural network to represent not only a single material, but to train a single network for an entire scene.

Individual objects may be differentiated by some additional data provided to the network; for example, in the CUDA inference we manually sort out irrelevant texels from the framebuffer by looking at the blue channel – this method could be used to not only sort out irrelevant texels, but also to assign a unique identification number to a certain material.

Comparison with BCn compression

As detailed in Section 1.2.1, we do not compare this method to other standard texture compression techniques, such as BCn. A future iteration on this thesis could evaluate how the different approaches vary in texture reconstruction quality, memory bandwidth, storage space, and frame time. This is particularly interesting for applications with less powerful hardware where GPU memory is limited.

6.1.2 Neural Network Implementation

There were several concepts and implementation details in the neural network that we either discarded or never implemented due to time constraints. Below we detail the implementations and concepts that would be suitable for further study regarding the structure of the neural network itself.

Floating Point Representations

All parameters, latent codes, and Gaussian matrices were saved as 32-bit floating point representations. This is for most cases unnecessarily large and potentially makes the storage sizes much larger than is required for the model. A future implementation could try training the networks with parameters stored as 16-bit or even 8-bit floating point formats to reduce the file size even more. The excess storage could for example be used for latent codes of larger dimensions, in order to increase the quality of the results.

Adding Uniform Noise to Latent Codes

In their proposed method, Vaidyanathan et al simulates quantisation of the latent code values by adding uniform noise according to the number of desired quantisation levels. This was briefly experimented with during the thesis, but not investigated thoroughly enough to discern the impact of different amounts of noise.

Experimenting with Grid Alignment

One noticeable artefact in the trained images are faint lines across the edges of the images. We theorise that these lines stem partly from a data loss close to the edges where proportional scaling is used, partly from that the grids happen to align around the edges, making interpolated data very similar. This was supported by the fact that model loss decreased slightly when changing the scaling to be half a pixel larger than it actually was. Future experiments in combating grid alignment could entail a combination of different scaling and uniform shifts.

Distribution of Latent Code Values

The initial latent code values were randomly sampled from a normal distribution with fixed means and standard deviations – our mean and standard deviations were selected through a simple trial, but were not thoroughly analysed. Later trials found that different values of the standard deviation could impact the convergence time for the network training more than previously thought. If quick training times are of specific interest, these normal distribution parameters could be investigated further.

Random Cropping

During training, this implementation trains on all texel coordinates for a material set at the same time. This, however, results in backpropagation focusing on areas of high complexity overall in the textures, leaving some areas largely untouched after the first set of training epochs. A future iteration could therefore improve the inferred image quality by using a set of random croppings of a texture instead of the whole texture at once.

Triangle Wave Mapping

Müller et al proposed triangle waves as an alternative to the sinusoid functions in the Fourier frequency mapping [5]. This method had a resulting 0.25 ms reduction in frame time for no noticeable reduction in quality. Vaidyanathan et al used this method as well in their implementation. A future iteration on this thesis subject could investigate different positional encodings that still mitigates the spectral bias of MLP networks.

6.1.3 Inference Implementation

CUDA Inference

The CUDA inference implementation used in this thesis is a simple matrix multiplication performed with for-loops. This is a largely unoptimised way of performing inference. Approaches using cuBLAS and CUTLASS were explored, but eventually abandoned due to time constraints. The variable input size means that there must be some pre-processing before the inference is performed with these libraries, otherwise the inference would require to be calculated for the whole screen at all times, regardless of geometry coverage. A future implementation could aggregate relevant UV-coordinates into a separate matrix, on which the inference is performed, and then written back to their respective frame buffer location. More on these methods can be found in Appendix C.

References

[1] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020.

[2] Simon Boehm. How to optimize a cuda matmul kernel for cublas-like performance: a worklog, 2022.

[3] David R. Bull and Fan Zhang. Chapter 4 - digital picture formats and representations. In David R. Bull and Fan Zhang, editors, *Intelligent Image and Video Compression (Second Edition)*, pages 107–142. Academic Press, Oxford, second edition edition, 2021.

[4] Shengmei Liu, Atsuo Kuwahara, James J Scovell, and Mark Claypool. The effects of frame rate variation on game player quality of experience. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *ACM Trans. Graph.*, 40(4), jul 2021.

[6] Manasa Nadipally. Chapter 2 - optimization of methods for image-texture segmentation using ant colony optimization. In D. Jude Hemanth, Deepak Gupta, and Valentina Emilia Balas, editors, *Intelligent Data Analysis for Biomedical Applications*, Intelligent Data-Centric Systems, pages 21–47. Academic Press, 2019.

[7] Jim Nilsson and Tomas Akenine-Möller. Understanding ssim, 2020.

[8] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation, 2019.

[9] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[10] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains, 2020.

[11] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *CoRR*, abs/1904.12356, 2019.

[12] Kimhan Thung and Paramesran Raveendran. A survey of image quality measures. In *2009 International Conference for Technical Postgraduates (TECHPOS)*, pages 1 – 4, 01 2010.

[13] Karthik Vaidyanathan, Marco Salvi, Bartlomiej Wronski, Tomas Akenine-Möller, Pontus Ebelin, and Aaron Lefohn. Random-Access Neural Compression of Material Textures. In *Proceedings of SIGGRAPH*, 2023.

[14] Zhou Wang, Eero Simoncelli, and Alan Bovik. Multi-scale structural similarity for image quality assessment. *Proceedings of the IEEE Asilomar Conference Signals, Systems and Computers*, 02 2004.

[15] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for neural networks for image processing, 2018.

Appendices

Appendix A
Network result images

(a) Naïve (b) Autodecoder (c) Fourier (d) Ground truth

Figure A.1: Comparisons of all material channels in the *sponza_details* material for each network model. From top to bottom: diffuse, normal, and specular maps.

Appendix B
Rendered images

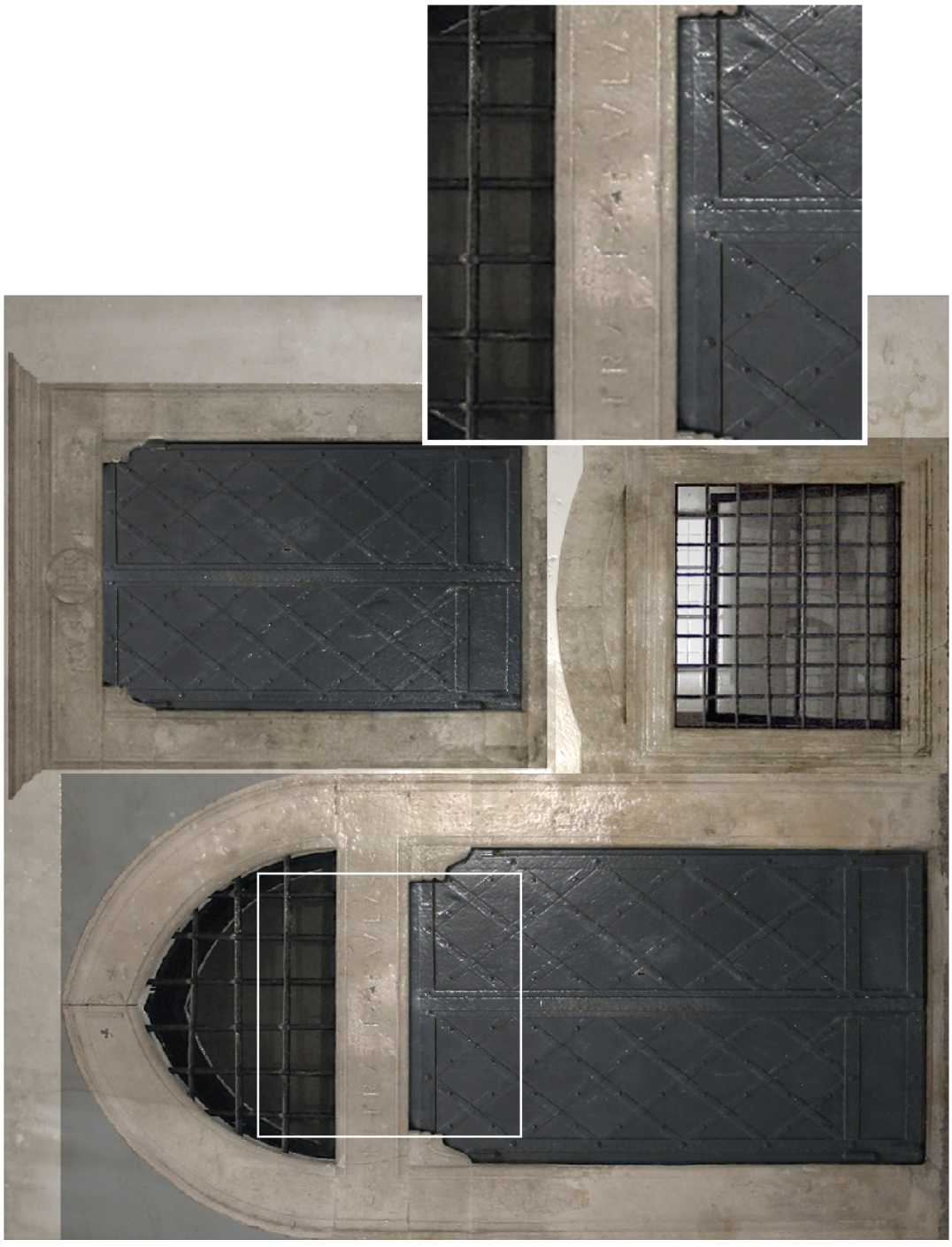


Figure B.1: Render with regular *sponza_details* diffuse texture.

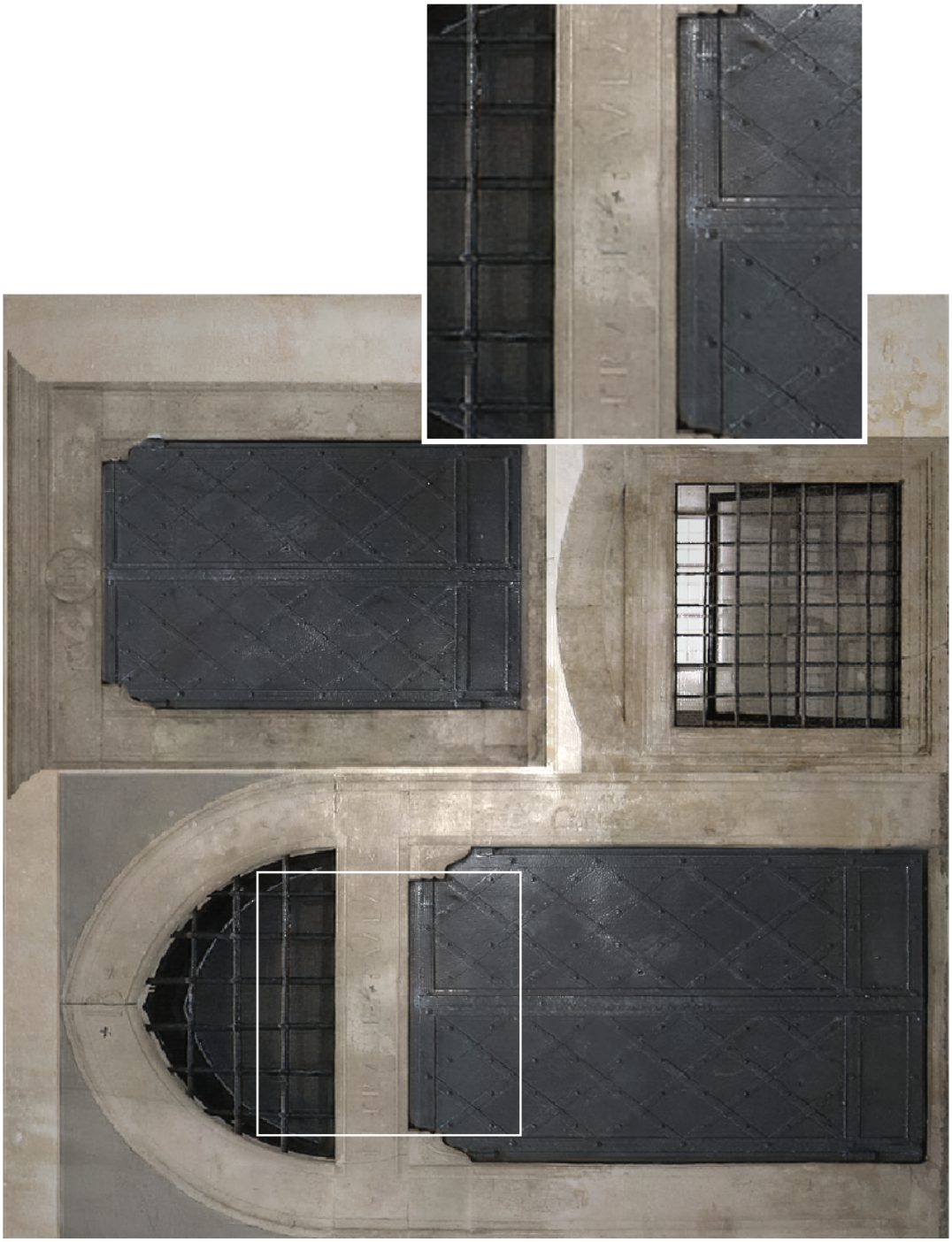


Figure B.2: Render with the overall best Autodecoder model, fragment shader inference.

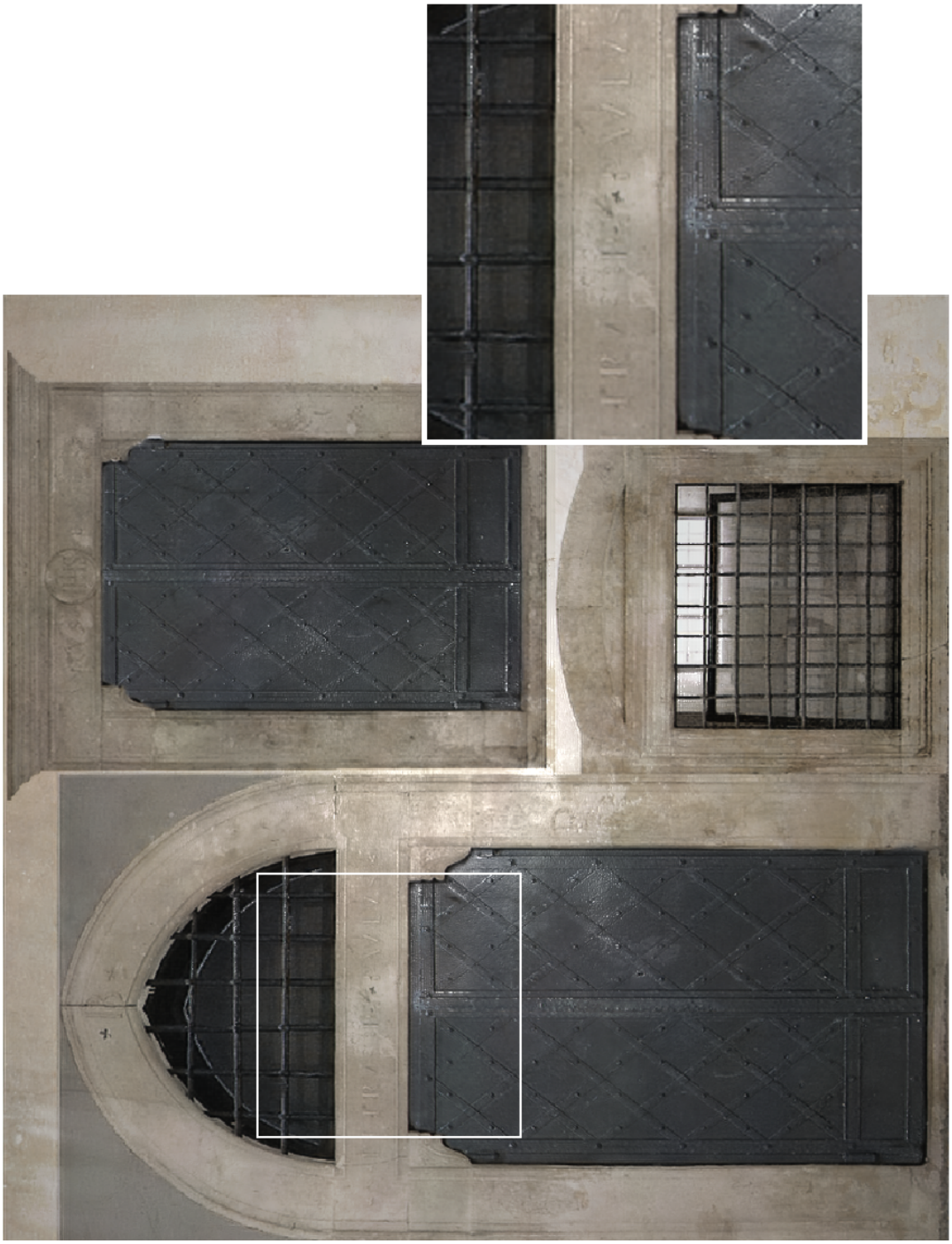
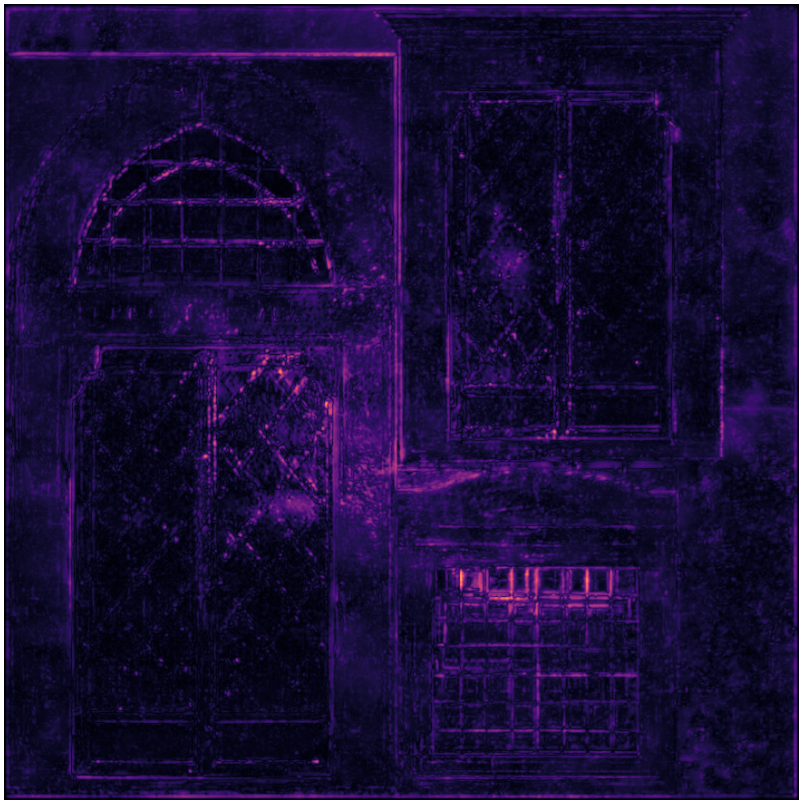
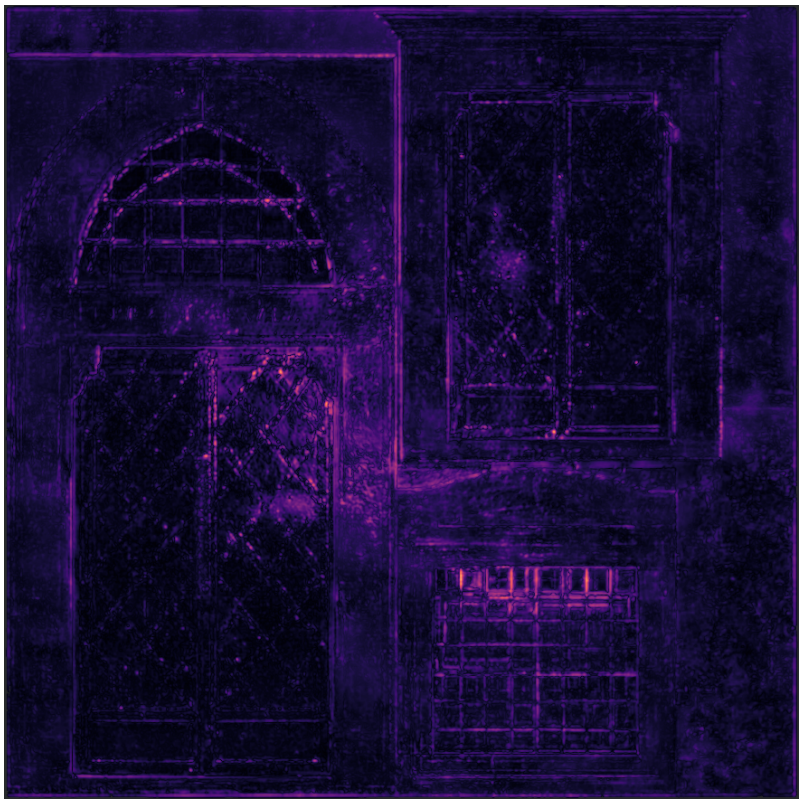


Figure B.3: Render with the overall best Autodecoder model, CUDA inference.

Figure B.4: Render with the overall best Fourier model, CUDA inference.



(a) ꟻLIP heat map of Autodecoder shader inference compared to regular texture render.



(b) ꟻLIP heat map of Autodecoder CUDA inference compared to regular texture render.

Figure B.5: ꟻLIP heatmap comparisons of renders.

Appendix C
Investigation of inference methods

The methods mentioned in Chapter 3 were fully implemented and tested – we also investigated other solutions to the inference problem, but did not implement them fully due to time constraints. Following are descriptions of what other methods we tried, their potential benefits, and the issues we encountered.

C.1 TensorRT

TensorRT is a deep-learning inference API developed by NVIDIA which optimises a trained network and creates its own runtime for execution. We experimented with exporting our trained model to a *.onnx* file and constructing a TensorRT runtime engine from it, but encountered problems due to the large potential variation in input size to the network. It is possible to define a variational network input size with batching in TensorRT, which means that one could adapt the batch size dynamically. The optimisation process ran out of memory when testing a single batch for the whole inference, so one would have to find a batch size which balances throughput with memory footprint.

A single TensorRT inference call schedules the inference task on the GPU, and multiple batches would then require synchronisation to prevent the CPU from writing the output before the process is finished. A potential implementation would be to fix the batch size to a smaller amount, and after synchronisation stitching together the results to the final framebuffer. This method would require manual culling of the non-relevant texels through a CUDA kernel in addition to the TensorRT inference, as well as a CUDA kernel to convert the inputs to the Fourier mappings. The potential gains by using this method is unclear, as the optimisations very much depend on the structure of the neural network itself – a simple MLP such as the one we use might not be able to change much without impacting the resulting image.

C.2 cuBLAS

NVIDIA cuBLAS is a linear algebra API leveraging CUDA to perform very fast vector and matrix operations. We investigated using this API for the inference multiplications as it has a much larger potential throughput than a naïve looping matrix multiplication, but encountered issues with the variable nature of the inference.

The easiest approach is to perform the inference for the whole screen regardless of how many pixels are covered by the target geometry, and then culling the irrelevant texels – this would waste precious time performing computations that are not required. Ideally, we would only want to perform the inference for the texels that are relevant – but this would require to fit the desired texels into a square matrix, performing the inference, and then putting the outputs in the corresponding texel again. In previous versions of cuBLAS one could make API calls from a CUDA kernel, which would significantly impact the inference complexity by then only requiring a single kernel processing each input texel.

Reports of the performance of a cuBLAS matrix multiplication compared to a naïve implementation such as ours varies – presumably on which GPU is used and what matrices are multiplied – but the consensus is that it is a moderate to large increase in throughput.

C.3 CUTLASS

An alternative to cuBLAS that still has the option to make accelerated matrix multiplication calls in a CUDA kernel is using CUTLASS, which is a collection of CUDA C++ matrix computations that employs similar strategies to the cuBLAS API to increase performance developed by NVIDIA. *Dynamic parallelism* is the concept of scheduling and launching GPU tasks from an already running GPU task – some CUTLASS functions are possible to use inside a CUDA kernel directly, although it requires very careful memory management and tuning of the kernel parameters to not encounter issues and race conditions. The potential benefit to using CUTLASS is mainly in the relative ease to fit into our use case – it does not require a lengthy engine setup as with TensorRT, nor does it require preprocessing to prevent redundant calculations as with cuBLAS. The potential increase in performance is unknown, but considering its similarity in implementation with cuBLAS we could assume BLAS-like increases for a single matrix.

Appendix D
Investigated neural network models

D.1 Naïve

A first test was conducted with the following simple approach: an MLP with four layers, 256 neurons each, the input being the UV-coordinate of the corresponding texel. Despite testing a variety of number of layers, neurons and activation functions, this network struggled to reproduce any of the images with high detail, often just outputting a blurry mess vaguely in the same colours as the desired output. We suspected that this was due to the lack of information in the input data, and thus started exploring ways to encode the input *(u, v)* to some other representation that fit more details.

The first representation we tried was combining the grid-like UV-mapping with the corresponding polar coordinates $(x, y) = (rsin\phi, rcos\phi)$ originating from the image center. This, however, did not result in any substantial gains in reconstruction quality – large radial patterns occurred, likely due to the polar coordinates being more pronounced in the edges (as in the scaling factor r being larger). Following this, we started exploring other network architectures capable of learning underlying patterns.

D.2 Autoencoders

Our first approach to expand the information in the input data was not to manually create any type of encoding, but rather to allow a network to create and train its own encoding. The autoencoder architecture is usually tasked with reproducing an input according to its underlying features, but we theorised it could be possible to learn underlying features of each *(u,v)* relating to its corresponding output texel, and thus be trained to learn the patterns of the image more efficiently.

Our first autoencoder implementation was a MLP architecture. After experimenting with different layer dimensions, number of neurons in each layer, and code sizes, the output was still similar to the output of the naïve linear network – vaguely triangular shapes approximating the image in broad strokes, but no fine detail even with much larger networks.

After this we tried using a convolutional approach to the autoencoder, with much better results. This was based on the MS-SSIM implementation by Zhao et al, which was further extended by a border of padding and a cropper for the input images. A significantly more detailed image could be produced, with some minor information loss in the edges. Despite this, we deemed this approach to be unsuitable for our use case due to the convolutional model being both more computationally intensive leading to much worse inference time and more complex to implement.

D.3 Frequency Encoding

After retracing our steps from the autoencoder approach, we tried applying the Gaussian random Fourier feature mapping presented by Tancik et al to the simple MLP mentioned above[10]. By mapping the texel UV-coordinates to a frequency space, we can get more high-frequency data from an input coordinate. The UV-inputs is a tensor containing the coordinates $(u, v)_{ij}$ of each image texel (i, j). This method quickly produced very accurate results with much quicker training times as well as being simpler to implement for the real-time inference.

D.4 Input Processing

To improve the quality of the image, different input processing were tested.

D.4.1 Naïve

The first input processing that was tried was no processing. This was a natural start and gave us a control baseline. The image was divided into a grid on the pixel-level as their UV-coordinates and evaluated to each respective color of the pixel.

D.4.2 Fourier Mapping

For the Fourier mapping explained in Section 2.3 five dimensions of matrices were tested as well as five different scales of the standard deviation for the Gaussian matrix $\boldsymbol{B}$. The Fourier mapping proportionally increases the input with its m dimension. For example, the UV-coordinates mapped with 8×2 result in an input of 16. Finally, this can suddenly bring about a much larger input than expected if not balanced properly.

D.4.3 Padding and Cropping

To minimise the data loss in the edges of the convolution MSSIM autoencoder a padding with a replicating pattern was added, meaning the values of the pixel in the outermost edge was prolonged a couple of pixels out and the dimensions of the image is increased. After running the training, the result image was cropped back to the original size to be evaluated to the validation image.

For the linear model a padding of the input data was added to further differentiate the different image maps from each other. The input for each image map would be the same.

D.4.4 Choice of Loss Function

To investigate the different loss functions characteristics and their suitability for the network, four loss functions where chosen from different types av loss functions. The two which are based on the absolute error are MSE and PSNR. The other two, MSSIM and ꟻLIP have perception and spatial aspects. Mixing the MSE and MSSIM via different ratios – to gain both the loss functions benefits – was briefly explored.

The MSE function from PyTorch's torch *nn* package was used as MSE. The MSSIM function that was used where from the *pytorch_msssim* package[1]. Lastly ꟻLIP was from the NVIDIA research GitHub[2].

Different parameters of the network was tested to find the best result. These are shown in Table D.1.

Table D.1: Different parameters tested for the linear model.

Parameter	Variations
Number of neurons, N	64, 128, 256, 384, 512, 768, 1024
Epochs, n	2000, 6000 12000, 36000
Gaussian scale, σ	5, 10, 15, 20, 30
Gaussian matrix size	8×2, 12×2, 16×2, 20×2, 32×2

Deviations in the pixel value shift was also tested. Range of 0 to 1 with increments of 0.2.

The different feature matrices will be notated by their dimensions. These consists of 8×2, 12×2, 16×2, 20×2, 32×2 and the specific values in matrices can be seen in the final results in Chapter 4.

Different amounts of epochs were tested in the range from 6000 to 24000. This was to explore how the image quality changed and to discover any potential over-training behavior, as the pure training time was outside of the scope.

[1]https://github.com/VainF/pytorch-msssim
[2]https://github.com/NVlabs/flip

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-10-04

EXAMENSARBETE Neural compression and decompression of textures in real-time rendering
STUDENTER Tove Börjeson, Filip Vannfält
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Per Andersson (LTH)

Komprimering av texturer med neurala nätverk

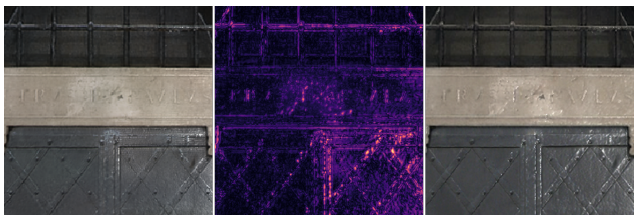
POPULÄRVETENSKAPLIG SAMMANFATTNING **Tove Börjeson, Filip Vannfält**

När den visuella kvaliteten i realtidsprogram ökar görs det på bekostnaden av lagringsutrymme, grafikminne samt bildhastighet. Detta examensarbetet utforskar hur olika neurala nätverk kan användas för att minska lagringsutrymmet samtidigt som man bibehåller kvaliteten av bilderna. Genom att låta nätverket generera dessa i realtid så behöver dessa inte sparas separat.

Inom datorgrafik beskriver man hur ljuset påverkas av en yta med hjälp av en samling texturer. Dessa kan exempelvis beskriva en ytas metallglans, färg, eller reflektans. Att öka den visuella kvaliteten på en rendering innebär ofta i grunden att öka upplösningen på dessa texturer. Detta gör dock att lagringsutrymmet ökar kvadratiskt och blir ett problem med avseende på både lagringsutrymme och realtidsminnet i GPU:n. I detta examensarbete utforskar vi hur man kan använda neurala nätverk för att minska lagringsutrymmet av texturer.

Ett neuralt nätverk kan ses som en generell uppskattare, som tränas för att efterlikna mönster och andra kopplingar mellan indata och utdata. Vi använder detta koncept för att skapa en modell där en pixelposition ges som input, och output är färgvärdena på de motsvarande materialtexturerna. Med denna metod så kan man således rekonstruera texturerna i realtid. Om det neurala nätverket är av mindre lagringstorlek än originalbilderna kan man därför använda det som komprimering.

Vi utforskar två neurala nätverkskonfigurationer och jämför dem med avseende på bildkvaliteten, lagringsutrymme samt hur snabbt rekonstruktionen görs i realtid. Utöver detta har vi även jämfört två olika sätt att applicera nätverken i ett befintligt renderingsprogram.



Figur 1: Resultatet från nätverket (t. v.) följt av skillnaden och originalet.

I Figur 1 kan en jämförelse mellan det neurala nätverkets resultat och den faktiska texturen ses. Den inställning av nätverket som användes i figuren är det nätverk som hade den bästa möjliga bildkvalitet, men som fortfarande innebar en minskning i lagringsutrymme. Resultaten visar på att tekniken kan användas som komprimering, men att kostnaden i både antal millisekunder per bild samt bildkvaliteten gör det svårt för denna metod att ersätta traditionella texturer i nuläget.