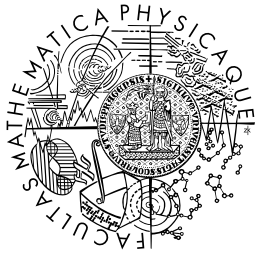


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Filip Jurčák

Material picker: Material recognition in images using machine learning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Petr Vévoda

Study programme: Computer Science - Artificial Intelligence

Study branch: Machine learning

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to sincerely thank my supervisor for everything he had done for me throughout the whole thesis, and there was a lot.
I would also like to thank my girlfriend Eďa for always pushing me to strive to be my best self and always being by my side.

Title: Material picker: Material recognition in images using machine learning

Author: Filip Jurčák

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Petr Vévoda, Department of Software and Computer Science Education

Abstract: The process of setting material properties for realistic appearance after rendering is usually tiresome and often requires carefully crafted skill for fine-tuning the parameters, as different combinations of these parameters can produce different-looking materials. To simplify this process, we introduce a solution to the texture transfer problem by creating a pipeline containing several deep neural networks. These networks subsequently represent solutions to inverse rendering and material segmentation by predicting intrinsic scene characteristics, like diffuse and specular albedo, surface normals, glossiness, view vector, texture coordinates, and segmentation, all from a single image. Artists can subsequently plug these inferenced properties inside their 3D scene representations and thus reduce the time needed to iterate over several design ideas. To train these networks, we generated high-quality dataset of substantial size using physically-based techniques to ensure good generalization on real-world images.

Keywords: inverse rendering, texture extraction, material recognition, deep learning, material segmentation

Contents

Introduction

Setting material appearance is one of the most crucial steps in modeling 3D scenes and probably the most important one for creating a realistic model look. This task is often long, tedious, and requires non-trivial skill, as a lot of parameters need to be set up for a material to look realistic after rendering. The number of parameters varies between used material models, but advanced models often need tens of correctly set up parameters.

As a result of this thesis, we want to ease the whole process by providing artists with a material picker tool. This tool is a series of deep neural networks that estimate a number of intrinsic properties of an image, which would help us recover material from a user-specified object in an image.

We do so by inventing a pipeline that attempts to solve several fundamental problems in computer graphics and computer vision – inverse rendering, material segmentation and texture transfer – all from a single image. Our method performs per-pixel estimation of a number of intrinsic scene characteristics, such as diffuse and specular albedo, surface normals, glossiness, view vector and texture coordinates. After the estimation, we use the predicted properties to segment each material in the input image. We then use all of these inferred properties to extract texture of a user selected material in the input image.

To train all models in our pipeline, we present a way to create a modern dataset by using advanced features of a physically-based V-Ray renderer to bridge the gap between synthetic data and real images. This is crucial, as we most often want to generalize well on real images, which is hard to achieve with synthetic images only.

This work is continuation of our previous work in this area [Jurčák, 2020] and some parts of the original text were re-used (mostly theoretical background).

1. Problem statement

Defining properties of materials in the scene is essential for matching appearance of real world materials, but is time consuming. Even for very simple material models like Phong reflection model, one needs to correctly set 7 attributes for a single material, and there exist much more complex material models requiring tens of values to set up to achieve more realistic look. On top of this, most of the materials in the wild contain features that make them unique - bathroom tiles can be of different patterns, wood floor can have different kinds of grain and so on. These perceived surface details are stored in material texture. To mimic true to life materials, creating authentic texture image by hand can be especially tiresome task. This drill have to be repeated for every material used and although there exist libraries with vast number of ready to use texture images, these libraries don't have to include exactly what user had in mind.

1.1 Our goal

The procedure of setting the properties can be dramatically simplified, as artists and graphic designers typically create new looks from already existing artworks. If they were able to transfer the desired material characteristics from an image of these designs, it could be an enormous time saver. Our goal is therefore to offer them a tool that would predict material attributes they work with the most from just a single image.
To accomplish this goal, we need per-pixel estimation of several material attributes, which can be achieved by doing inverse rendering of a scene. Based on this estimation we will then perform material segmentation and let user select for which material in the scene to determine underlying texture. We will cover our approach more closely in chapter 3.
As we will show in chapter 2, there has been a lot of research lately regarding inverse rendering. Using deep learning techniques to tackle problems from different areas of interest proved to be very successful, so it was naturally applied to the computer vision field, and in our case, to inverse rendering as well. We believe that combination of these approaches will enable us to solve our task successfully.
As this work requires knowledge of terminology, methods and concepts from both computer graphics and machine learning, we elucidate both of these areas in the next sections.

1.2 Terminology and methods used in rendering

1.2.1 Rendering

Rendering is a major subfield of the computer graphics area. It refers to a sequence of steps that produces a 2D image from 3D representation of a scene stored in a computer. During this sequence – which is also called the rendering pipeline – the algorithm for handling rendering of a scene needs to take model representations, apply transformations, map textures to objects, (optionally)

throw away parts of the scene which will not be rendered, illuminate the scene from all the presented lights and finally draw an image from the view of the camera. There are several types of renderings based on different rendering algorithms, mostly divided into two categories: non-photorealistic rendering and photorealistic rendering, sometimes also called physically based rendering (PBR). The latter implements the concepts of transport and scattering of light in the real world, which is far more computationally expensive than the former approach but produces more plausible results.
As the primary goal of this thesis is to create a tool that could be used on real world images, we employ only PBR techniques during generation of our data to eliminate gap between real and synthetic images.
To look real when rendered, PBR needs (among other parameters) to have correctly set up material model, usually referred to as BRDF.

1.2.2 Bidirectional Reflectance Distribution Function

Bidirectional Reflectance Distribution Function (or BRDF for short) is a probabilistic function $f_r(\omega_i, \omega_o)$ ($f_r(\omega_i \rightarrow \omega_o)$) describing how light is reflected based on surface attributes. More specifically, given incoming light direction ω_i and outgoing direction ω_o, it gives the probability that a photon arriving from direction ω_i will be reflected to direction ω_o.
There are several categories of BRDF models, of which the most impactful ones are the physically-based BRDFs. To consider a BRDF model to be physically based, it must meet the following properties:

- positivity: $f_r(\omega_i, \omega_o) \geq 0$
- obeying Helmholtz reciprocity: $f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i)$
- conserving energy: $\int_\Omega f_r(\omega_i, \omega_o) \cos\theta_i \, d\omega_i \leq 1 \;\; \forall \omega_o$

where $\cos\theta_i$ represent decrease of radiance with increasing θ_i (angle between ω_i and surface normal).
To achieve realistic material look, it is important to use such BRDF that satisfies these properties. Example of such BRDF can be physically based Phong BRDF, which is equal to

$$f_r^{Phong} = \frac{\rho_d}{\pi} + \frac{\rho_s(n+2)\cos^n\theta_r}{2\pi} \tag{1.1}$$

where ρ_d stands for diffuse albedo, ρ_s for specular albedo, n for glossiness and θ_r for angle between view vector and reflected light vector.

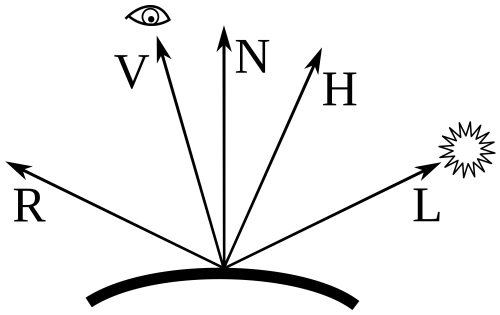


Figure 1.1: Illustration of vectors used in physically based Phong BRDF. N is normal vector, V is view vector, L is light vector, H is half vector between light and view vector and R is reflected light vector. Taken from Wikipedia [2023]

1.2.3 Reflection equation

Knowing the BRDF of a surface allows us to compute how much light coming from direction ω_i is reflected from the surface to direction ω_o. For that one has to multiply radiance L_i from direction ω_i, BRDF $f_r(w_i \rightarrow w_o)$ and $\cos\theta_i$. Summarized in mathematical notation:

$$L_r(\omega_i \rightarrow \omega_o) = L_i(\omega_i) \cdot f_r(\omega_i \rightarrow \omega_o) \cdot \cos\theta_i \tag{1.2}$$

In order to compute the total radiance reflected to direction ω_o we need to sum up the contributions from all light sources, direct or indirect. This can be done by integrating these contributions over upper hemisphere $H(x)$, which gives us the following equation

$$L_r(\omega_o) = \int_{H(x)} L_i(\omega_i) \cdot f_r(\omega_i \rightarrow \omega_o) \cdot \cos\theta_i \, d\omega_i \tag{1.3}$$

also called reflection equation. This integral generally does not have an analytical solution and has to be computed numerically.

1.2.4 Monte Carlo integration

A typically used numerical method for solving integrals in rendering is Monte Carlo integration. This technique uses random numbers to sample points at which the integrand is evaluated. Let's denote an integral that we want to approximate, as

$$I = \int g(x)dx \tag{1.4}$$

Monte Carlo estimator of I is defined as

$$\langle I \rangle = \frac{1}{N} \sum_{k=1}^{N} \frac{g(X_k)}{p(X_k)}, \tag{1.5}$$

where N is the number of samples taken, X_k, $k = 1, ..., N$ are the samples and $p(x)$ is a probability density function from which the samples were drawn.

By substituting equation 1.3 into equation 1.5, the result is

$$\langle L_r(\omega_o)\rangle = \frac{2\pi}{N}\sum_{k=1}^{N} L_i(\omega_{i,k}) f_r(\omega_{i,k} \to \omega_o)\cos\theta_{i,k} \tag{1.6}$$

where 2π stands for the probability density function ($p(X_k) = \frac{1}{2\pi}$) of uniform sampling directions on the hemisphere and $\omega_{i,k}$, $k = 1, ..., N$ are the sampled directions.
An image can be rendered by evaluating equation 1.6 for positions in a scene viewed from each of the image pixels. Given such an image, our goal is to estimate what material (i.e. $f_r(\omega_{i,k} \to \omega_o)$) was used when the image was rendered.

1.2.5 Inverse rendering

One of the possible methods for estimating what materials are present in an image is inverse rendering. Inverse rendering is one of the principal and long-standing problems in computer vision and computer graphics. Its main goal is to, provided an image or several images of a scene, estimate intrinsic properties of a scene, like depth, albedo, normals, reflectance, lighting and others. This problem is hard for several reasons, mainly, as stated in Li et al. [2019]: „This is an ill-posed task: these scene factors interact in complex ways to form images and multiple combinations of these factors may produce the same image.“ As we can see, there is an infinite number of solutions for parameters for a single image, which makes the problem hard or almost impossible to solve. However, some solutions are statistically more admissible than others. Citing Barron and Malik [2015], which says: „Our goal is therefore to recover the most likely explanation that explains an input image.“ To make this work, we need to come up with such statistics that would correctly approximate the real world. This is not straightforward, but recent advances in both optimization and learning based approaches show that it's possible to estimate a handful of properties correctly [Barron and Malik, 2015] and even better results when physically based datasets were used for training neural networks [Sengupta et al., 2019],[Li et al., 2019]. With these properties in hand, we want to estimate what is the material on the user-specified object in the image.

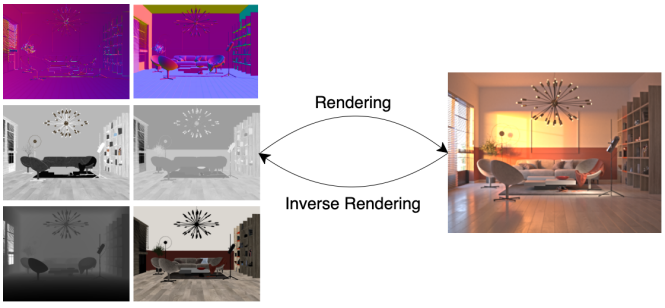


Figure 1.2: Rendering vs inverse rendering; on the left individual scene elements and on the right final output of the renderer, using elements on the left as input

1.2.6 Texture mapping

One step of the rendering pipeline is texture mapping. Texture mapping is a process of obtaining texture information for each point on the surface that we want to shade. To determine the color of the underlying material, the renderer performs a texture lookup by sampling the corresponding texels (texture pixels) for each point on the surface [Shirley and Marschner, 2009], so it's a function from surface space into the texture space. Denoted mathematically:

$$\phi : S \to T \tag{1.7}$$

$$: (x, y, z) \to (u, v) \tag{1.8}$$

This mapping is called the texture coordinate function and needs to be defined individually for every object present in the scene.

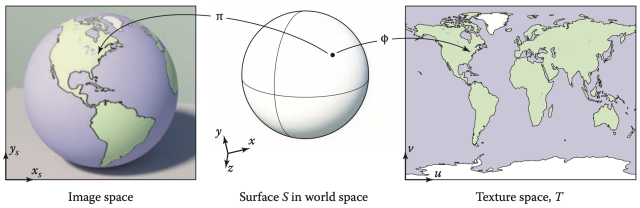


Figure 1.3: Example of texture mapping, taken from Shirley and Marschner [2009]

Yet not all mapping functions behave equally well when used for mapping the texture onto the object's surface. As most of the surfaces are not flat, used mappings have to account for potential distortions after the texture is applied. Some of the desired properties of a good mapping include:

- bijectivity: each point on the surface should map to different point in the texture space (if the texture is not repeated)
- size distortion: mapping should preserve distances between points on the surface and points in the texture
- shape distortion: mapping should preserve shapes, so shapes drawn in surface space should map to similar shapes in texture space
- continuity: edges of the texture should follow up on each other to ensure a small number of discontinuities

However, these properties usually go against each other, and choosing the right mapping means balancing the aforementioned qualities to achieve the most realistic look.

1.2.7 Texture transfer

As explained in previous section, artists usually have to specify which texture is mapped to each surface as part of the scene preparation before the image can be rendered. Artists often look for inspiration of these textures in already existing work and then try to replicate what they have seen elsewhere. Reproducing

the texture from scratch however is very inefficient so artists usually look at previously created textures in texture libraries and choose the one closest to the desired appearance.
In these use cases, it would be beneficial to have access to the underlying texture, but more often than not this information is not available. This problem was termed texture transfer. The underlying issue in this problem is that „an object's texture as seen in a photograph is distorted by many factors, including pose, geometry, and illumination." [Wang et al., 2016]. These deformations need to be reversed to recover the original texture, but we usually neither have access to full geometry of the object on which the texture was applied (as parts of the object are occluded) nor all light sources (to remove all highlights), this reverse mapping in general can't be performed exactly. Some approximation is possible nonetheless: we can estimate how the original mapping looked like from the non-occluded part of the object. Most of the object's surface is usually not covered entirely by one texture image, but the texture pattern is rather repeated to wrap around the object's surface. By observing the geometry onto which the texture was mapped on we can attempt to reverse the texture mapping process, in case the texture is not distorted too much.

1.2.8 Image inpainting

Image inpaiting is a technique which objective is to replace selected parts of the input image with plausible substitutions in such a way that (citing Jain et al. [2023])„an observer cannot distinguish between the inpainted regions and real regions of the output image." This problem was traditionally solved by diffusion process, but in recent years using generative adversarial networks (more closely explained in section 1.3.6) led to great progress in this area.
Examples where image inpainting is used include old photo restoration or photo-editing, by allowing users to remove unwanted objects from the taken image or replace them with objects of users' choosing.

(a) Original image (b) Inpainted image

Figure 1.4: Example of object removal using inpaiting, inpainted part marked by blue mask

We will use network trained to tackle image inpaiting as part of our own solution to texture transfer problem, described later in chapter 3.

1.3 Terminology and methods used in machine learning

1.3.1 Machine learning

Machine learning is a set of methods that allow computers to learn complex concepts from simpler ones or from experience. We provide this experience in a form of a dataset that consists of information (usually called features) about the task for which we want to train the model. Generally, we let the algorithm decide which features are important and how will individual feature contribute to the final prediction. When the dataset includes the desired prediction amongst its features, we refer to this type of machine learning as supervised machine learning. There exists other types, such as unsupervised or semi-supervised machine learning, but in our work we only use supervised learning methods.
Machine learning is an outstandingly fast advancing area of research and it helps to push research forward in other areas as well. Computer graphics is not an exception: one of the examples is calculating direct illumination by utilizing machine learning techniques [Vévoda, Petr and Kondapaneni, Ivo and Křivánek, Jaroslav, 2018].

1.3.2 Neural networks

Human brains consist of nerve cells, which are called neurons. These neurons form large networks where they can propagate information from one neuron to the other. The purpose of neural networks (NNs) as a machine learning method is to mimic these networks to be able to learn and make decisions. The main difference between neural networks and traditional programming is that while in traditional programming we explicitly instruct a computer what to do in each step of the program, we don't instruct neural networks how to behave or how to solve the task. We simply allow it to examine the provided data and let it propose a solution. This solution can be viewed as mapping $\mathcal{F}'$, where $\mathcal{F}$ is the underlying mapping that we want to approximate. $\mathcal{F}'$ should be optimal in some sense - we need such $\mathcal{F}'$ that minimizes

$$\frac{\sum_{x \in X} error(\mathcal{F}(x) - \mathcal{F}'(x))}{|X|}$$

where X is a set of inputs to the neural network.
Neural network consists of several layers, which are called input, hidden and output layer, with input and output layers required in every neural network, but any non-negative number of hidden layers is allowed. Every layer consists of several neurons. In the most common type of neural network, all neurons from previous layer are connected with all neurons in the next layer. These connections are called **weights** and network learns them throughout the training. We can see weights between one layer to other in figure 1.5 and an example of a simple neural network in figure 1.6.
Neural network performs two operations – **forward propagation and back-propagation**. The former is used to get the prediction, the latter to adjust weights in the system to account for the computed error. During forward

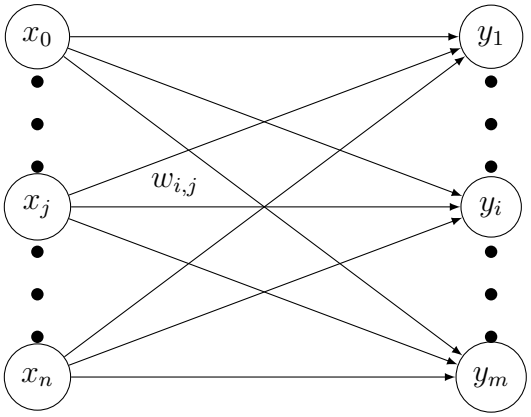


Figure 1.5: Connections between two layers of neural network, circles are neurons and lines represent weights, weight $w_{i,j}$ represents connection between neuron j in first layer and neuron i in second layer

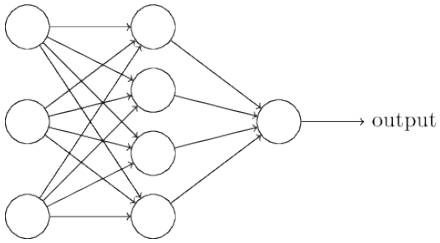


Figure 1.6: Example of a simple neural network with input, hidden and output layer. Circles represent neurons, lines between neurons show connections from neuron in one layer to neuron in the next layer. Taken from Nielsen [2015].

propagation, neural network computes values for all neurons in the next layer based on the previous layer. These values are then fed through some non-linearity function f to keep all the values in certain range (for example between $0 \leq x \leq 1$ or $-1 \leq x \leq 1$). Computed values are called **activations** of neurons. This process repeats until the network computes values in the output layer. Equation 1.9 summarizes the process of computing activation of one neuron, where n and m are the number of neurons in first layer and second layer respectively. Common thing to help neural network learn better is to add a **bias term** to the layer and initially set it as $x_0 = 1$.

$$y_i = f(\sum_{j=0}^{n} w_{i,j} * x_j) \quad \forall i \in \{1, \ldots, m\} \tag{1.9}$$

When we have computed the prediction, we need to adjust weights in a network to account for the difference between predicted value and the actual value. The error is then propagated back through the network in order to compute gradient, which is in turn used by some optimization method (for example gradient descent) to find local minimum of an error function, which is a metric for evaluating network's performance. The process of forward and back-propagation is repeated many times for every entry in the dataset until the process converges into some local minimum. This process is called training of the model. As the error function can have many local optima to which the training can converge it is recommended

to train the model several times with different model initialization to prevent getting stuck at some local optimum which is far from global optimum and to ensure robustness of the model.
Similarly to what model we use and how the model is trained, it is equally important to ensure the quality of the data on which we want to train the model. If the data is noisy, so will be the output of the network and that will prevent model from good generalization on unseen data. Machine learning models usually require plenty of experience to grasp the right abstractions: if there are only limited number of samples, straightforward approach is to just remember all of them and not learn anything useful, so having enough data for training is as substantial as having outlier-free data.

1.3.3 Deep learning and deep neural networks

Deep learning is a special kind of machine learning which is capable to learn more complex functions than simpler methods of machine learning. Every neural network that has more than one hidden layer can be considered a deep neural network. These multiple layers help the network to develop several levels of abstraction, which can give deep networks an upper hand in recognizing complex patterns over other methods or models [Goodfellow et al., 2016]. This is why so many solutions to pattern recognition problems employ this technique, but because of the relatively high computation power required for it's training, it wasn't used until very recently. Most models for inverse rendering use deep convolutional neural networks, which we will define in the next section.

1.3.4 Convolutional neural networks

Convolutional neural networks (CNNs, or sometimes just convolutional networks), are neural networks that are specially designed to process grid-like structured data, like images or videos.
Neural networks use matrix multiplication and activation function to compute the activation of neurons in the next layer. CNNs on the other hand, use a different approach - at least in one of their layers, they use a special kind of linear operation called convolution, which is defined as

$$s(t) = \int x(a) * w(t-a)\mathrm{d}a$$

where x is often referred to as the input and function w as the kernel. The output of the convolution is referred to as the feature map(s). Convolutional layers convolve the input with the help of the kernel function – which is just a function that transforms original input space into space, where it can be easier to train the model due to change from non-linear to a linear problem – and pass its result to the next layer. This is similar to the response of a neuron in our brain to a specific impulse. Because of this property, CNN is a great model for extracting edge information from images.
Convolution layer in CNNs consists of convolution stage, detector stage and pooling stage. During convolution stage, several convolutions are run in parallel to produce many layers of linear activations. During detector stage, all of these layers are run through some non-linear function to produce activations in certain range.

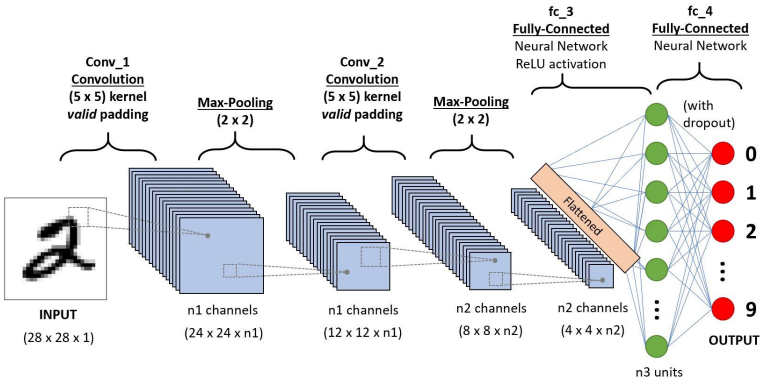


Figure 1.7: Typical CNN architecture for digit recognition, taken from Sumit Saha [2018]. The original image is run through several convolutional layers before finally being flattened with digit predictions as output

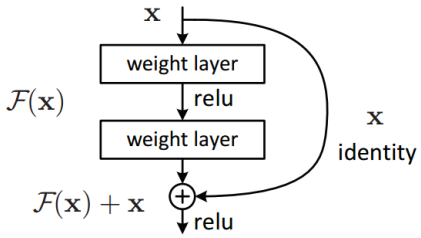


Figure 1.8: Example of residual block

And finally, during pooling stage, we use pooling function to produce statistical analysis of a specific neighbourhood in every layer. Typical CNN architecture for digit recognition can be seen in figure 1.7.
Another important property of CNN is its effectiveness when compared to traditional neural networks - performing convolution in layers of CNN is faster and requires orders of magnitude less storage then using NN for the same kind of problem [Goodfellow et al., 2016]. As a result, CNNs perform tremendously on image recognition tasks and are now one of the state-of-the-art solutions for this challenging problem.

1.3.5 Residual neural network

As we stated in section 1.3.3, deep networks have the ability to learn several layers of abstraction, which means that depth is important. This is especially valuable when working with images or videos, as these layers can help decompose input image into low to high level features of the image. However, just adding more and more layers brings problems like non-convergence of the whole network or accuracy degradation. The former was mostly resolved by normalized initialization, the latter by introduction of residual learning, with residual neural network as its architecture [He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian, 2016]. Residual neural network is network consisting of residual blocks, as shown in figure 1.8. Idea behind this block architecture is that rather than finding mapping $\mathcal{H}(x)$ that would be optimal for this block without any

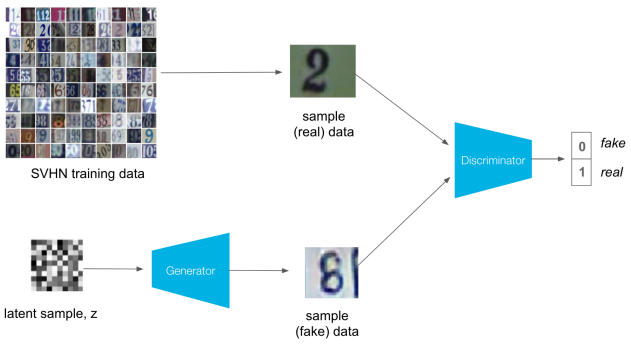


Figure 1.9: Visualized GAN architecture, taken from Medium [2023].

prior, we reformulate the mapping that the block should learn to $\mathcal{F}(x) = \mathcal{H}(x) - x$, so the output mapping then becomes $\mathcal{F}(x) + x = \mathcal{H}(x)$. This substantially helps with training, particularly in cases where output of the block should be very similar to its input (meaning identity mapping is the optimal mapping). We use residual blocks in our networks a lot because they enable us to train deeper models, as they are easier to optimize than conventional CNN networks [He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian, 2016].

1.3.6 Generative Adversarial Networks (GANs)

Generative Adversarial Networks are a class of machine learning algorithms used for generative modeling. The architecture for this algorithm consists of two models – Generator and Discriminator – that compete with each other during training by acting like adversaries to one another. Generator's input is random noise sample and it outputs synthetic data (audio, text or images, depending on the use case). This output is then batched with sample of real data from the dataset and fed into the discriminator, whose task is to recognize if input image is real or fake. During training, models take turns where in each turn only one updates its weights, and thus getting better over time at their respective tasks. The overall goal is to train such generator which outputs can't be recognized from real data by the discriminant [Goodfellow et al., 2014]. At runtime, discriminator is not used and the generator infers the output by itself.
The ability of GANs to output realistic-looking and high-quality data make them an ideal choice for various applications, such as image synthesis or image editing. However, GANs suffer from several inherent difficulties: for example, discriminator can easily recognize fake images produced by the generator at the beginning of the training; on top of that, generator can sometimes produce satisfactory output only for a subset of the desired distribution present in the training data – this problem is called mode collapse. All of these problems can be mitigated by using proper loss functions, but even small adjustments to the parameters of these loss functions can lead to extremely unstable training. Due to its limitations, GANs have been surpassed in recent years by another class of generative machine learning algorithms, called diffusion models.

1.3.7 Diffusion models

Similarly to GANs, diffusion models also produce images from random noise samples, but their training dynamic is different: during training they learn to remove random Gaussian noise that is added to input image in several steps. This dynamic is modeled by parameterized Markov chain. During inference, models do the reverse operation: in multiple steps they add noise to the input sample until the desired output quality is reached [Ho et al., 2020]. This leads to much stable training compared to GANs and thus offer more room for experimentation.

2. Prior work

In this chapter, we would like to refer to research done related to the problems we set out to solve: inverse rendering, material segmentation and texture transfer. Although these topics have been studied for several decades now, significant progress was achieved in only a past decade or so due to the inception of deep learning methods.

2.1 Inverse rendering

As inverse rendering of a scene is difficult task, previous research in the field focused either on subproblems of this problem (like inverse rendering of an object instead of a whole scene), estimation of a small number of properties of a scene [Li et al., 2019], [Sengupta et al., 2019] or a small number of materials [Bell et al., 2015].
To estimate intrinsic characteristics of an image authors in Li et al. [2019] and Sengupta et al. [2019] used deep neural networks. To obtain the data for training, they augmented SUNCG dataset [Song et al., 2016] by mapping photorealistic materials to geometries in this dataset or completely re-render images by using physically based renderer, as the original dataset was rendered only with OpenGL using Phong BRDF model and does not look realistic.
In Sengupta et al. [2019], authors proposed a pipeline for estimating diffuse albedo, environment map and normals from a single image by using Inverse Rendering Network (IRN) and combination of two modules - direct renderer for computing direct illumination and Residual Appearance Renderer (RAR) for computing shading and reflections - to re-synthesize the input image from estimated components and to learn from real images where ground-truth data is not available. To train IRN to correctly predict environment map, they had to generate ground-truth data, as the environment map with which the scene was rendered was used as exterior lighting, but it does not reflect illumination inside the scene. To address this issue, they also trained neural net (EnvMap net) to predict best average environment map for the whole scene (including illumination inside the scene), which they then set as their ground-truth for this parameter of IRN. The environment map predicted by IRN was then used in the direct renderer to approximate incoming illumination using numerical quadrature.
Different approach was presented by Li et al. [2019]. In this paper, authors were able to predict diffuse albedo, normals, specular roughness, depth, and spatially-varying lighting, which is a technique for estimating per-pixel illumination. Obtaining such data unwisely is computationally expensive and memory consuming, thus they resolved to use spherical Gaussian lobes, that preserve all lighting frequencies but require far less parameters to store. This very detailed pre-computed irradiance enabled them to include differentiable renderer into their pipeline and simulate image creation process without any rendering related code written by the authors. Due to this precise estimation of parameters, state-of-the-art object insertion and material editing were made possible.

2.2 Material Classification and Segmentation

Material segmentation is especially challenging, as real-world materials have a rich texture, and the final look of the material is a combination of many scene properties like lighting, depth, normals, and others.
There exists a large number of classifiers for classifying images into classes (like dogs, cats, etc.): e.g. AlexNet [Krizhevsky et al., 2012], VGG [Simonyan and Zisserman, 2014] and GoogLeNet [Szegedy et al., 2015]. These classifiers take an input image and their output is per-class probability of the object in the image belonging to that class. Most used approach to image segmentation we found was the use of transfer learning on models pre-trained as classifiers [Long et al., 2015], [Bell et al., 2015]. Transfer learning is method for re-using parts of already trained model (and possibly change the output layers), and retrain only those layers that were not taken from the pre-trained model.
In Long et al. [2015], authors removed final classification layer and used several upsampling layers to output 21 feature maps of the same size as input image. These 21 maps represented per pixel probability of the pixel belonging to 21 classes they had in the dataset. To get the final image they had to apply post-processing by taking per-pixel maximum over these feature maps, with index of the map that contained maximum assigned as the final value. It is worth noting that the new model was trained on the exact same dataset as the pre-trained model.
On the other hand, Bell et al. [2015] introduced completely new and larger dataset with 23 material categories on which they fine-tuned a pre-trained model. Authors thus proved that transfer learning also works on different dataset than it was originally trained on, at least for image segmentation.
Unsurprisingly, the deeper the trained model was, the better it performed, with either GoogLeNet (22 layers) or VGG (16 layers) as winners in both publications. After introduction of residual networks, the state-of-the-art network for full image segmentation became DeepLab [Chen et al., 2016], taking advantage of its unprecedented depth - model that was retrained had more than 100 layers.
Different technique for segmentation into class was used by authors in He et al. [2018]. Their network predicts several regions of interests from the input image and each region is then further processed to decide whether the region contains one of the objects of interest and bounding box coordinates of the object within the selected region. Final part of the network is then trained to predict the desired segmentation.

2.3 Texture transfer

Previous research in the area of texture transfer consisted mostly of solutions to reduced problems instead of finding an answer to the original problem, mainly because of the restricted access to ground-truth texture data for all the materials. Li et al. [2019] used material estimation and predicted illumination from image to enable material replacement or re-lighting of the scene under novel lighting conditions using labeled data. This solution however did not include estimation of the original texture, but only replacing it with another, so it did not fit our desired use case.

In Munkberg et al. [2023] authors came up with solution to texture transfer problem by framing it as an inverse rendering task. Their neural networks were able to predict object topology, used materials and lighting, but their method was developed under strong assumptions: they assumed objects were primary focus of the image (covering substantial part of the photograph), were not rotated in any considerable way and that image was taken by a camera with flashlight. These restrictions limit usage of this solution to very small subset of scenarios that we would like to support.

Another example, this time by applying unsupervised learning to this problem was achieved by Wang et al. [2016]. In this work they attempted to transfer textures from objects of specific shape to objects of similar shape using patch extraction without having access to the underlying texture data. Data for this kind of task is very hard to come by and obtaining it is immensely time-consuming, their extraction capabilities were limited to some object shapes only and could not be used for generic structures.

As generative models have taken huge step in the output image quality, in Carson Katri [2023] authors utilized diffusion model to generate texture via text prompt to guide the model to the desired texture output. Instead of starting from random noise this tool also accepts texture image as input for only small refinements. Although this approach is valid and definitely useful, we believe this is not the solution to the original problem of texture transfer.

3. Our approach

3.1 Inverse rendering

As our start point we decided to replicate paper by Sengupta et al. [2019], as it was easier to reproduce than other inverse rendering papers. In their approach only diffuse albedo ρ_d, environment map and normals were estimated from IRN so the only choice for BRDF was ideal diffuse BRDF defined as

$$f_r = \frac{\rho_d}{\pi} \tag{3.1}$$

As most of the materials in real world are not only made of diffuse component, we decided to use more sophisticated BRDF model, concretely physically based Phong BRDF. To achieve this, on top of 3 properties estimated by Sengupta et al. we altered the IRN's architecture to also predict specular albedo, glossiness and view vector by stacking more residual blocks for each added parameter.
When inspecting code for direct renderer written by Sengupta et al., we found out that the equation for computing direct illumination was as follows:

$$f_{direct} = \frac{4\pi * \frac{\pi}{2}}{648} \sum_{i=1}^{648} \frac{\rho_d}{\pi} * L(\omega_i) * (\omega_i \cdot N) \tag{3.2}$$

where 648 corresponds to 18×36 light directions (one for each pixel of the environment map predicted by IRN), ρ_d stands for diffuse albedo (and thus the term $\frac{\rho_d}{\pi}$ for diffuse BRDF), ω_i for direction of incoming light vector, $L(\omega_i)$ for incoming illumination from direction ω_i (i.e. value of the corresponding pixel of the environment map) and N for normal of the surface.
The term

$$4\pi * \frac{\pi}{2} \ / \ 648 \tag{3.3}$$

corresponds to the size of one pixel of environment map when mapped to sphere. Equation 3.2 is incorrect, as pixels mapped closer to the poles of the sphere will occupy less sphere surface than those mapped closer to equator of the sphere. To account for this distortion, the contribution of incoming light from direction ω_i should be multiplied by the constant size of a pixel ($= 4\pi * \frac{\pi}{2}/648$) times cosine of deviation of the direction from the equator, i.e. $\cos\theta_l$ for $\omega_i = (\theta_l, \phi_l)$ in spherical coordinates. To summarize, fixed direct render function derived from equation 3.2 is then

$$f_{direct} = \frac{4\pi * \frac{\pi}{2}}{648} \sum_{i=1}^{648} \frac{\rho_d}{\pi} * L(\omega_i) * \cos\theta_l * (\omega_i \cdot N) \tag{3.4}$$

To recall, BRDF for physically based Phong is

$$f_r^{Phong} = \frac{\rho_d}{\pi} + \frac{\rho_s(n+2)\cos^n\theta_r}{2\pi} \tag{3.5}$$

where ρ_d and ρ_s are diffuse and specular components respectively, n stands for glossiness and $\cos^n\theta_r = (V \cdot R)^n$, V representing view vector and R corresponds to reflected vector, which is a light vector L flipped according to normal N and its calculation is specified in equation 3.6.

$$R = 2(L \cdot N)N - L \tag{3.6}$$

This direct renderer implementation however has one problem: it only uses environment lighting of fixed size, specifically 18×36, which leads to a problem when the input image has much higher resolution than 240×320 pixels. We thus adjusted the EnvMap network's architecture to always predict lighting estimate of $\frac{1}{8}$th of the size of the input image. Direct renderer will have to account for this variability, but this is not a problem since we will just sample more light directions and this will directly reflect in the equation 3.3, where term 648 will be replaced by the product of the sizes of the input environment map. In conclusion, our direct render function with physically based Phong BRDF is defined as

$$f_{Phong} = \frac{4\pi * \frac{\pi}{2}}{s} \sum_{i=1}^{s} (\frac{\rho_d}{\pi} + \frac{\rho_s(n+2)\cos^n \theta_r}{2\pi}) \ * L_i(\omega_i) * \cos\theta_l * (\omega_i \cdot N) \qquad (3.7)$$

where s stands for the product of the environment map spatial resolution. When implemented by matrix multiplication, our direct renderer function runs almost as fast as the original implementation of the direct renderer, even though our function uses twice as many parameters. Comparison of results from original direct render function and our own implementation can be found in Appendix A. In addition to all the elements needed for Phong's physically based BRDF, we also updated the network to predict texture coordinates for each material in the scene. For single material, these coordinates represent values of the mapping used to wrap the material around the object's surface. We can use this enumeration of the function values to estimate how the original mapping function looked like and by somehow reversing the process arrive at the original texture image. Not all materials however have these coordinates, as sometimes only single color is applied to the whole surface, which does not require any mapping. To help the network learn the distinction between these two types of materials, we marked the surfaces that use only single color as black. Our ground-truth data for texture coordinates is shown in figure 3.1.

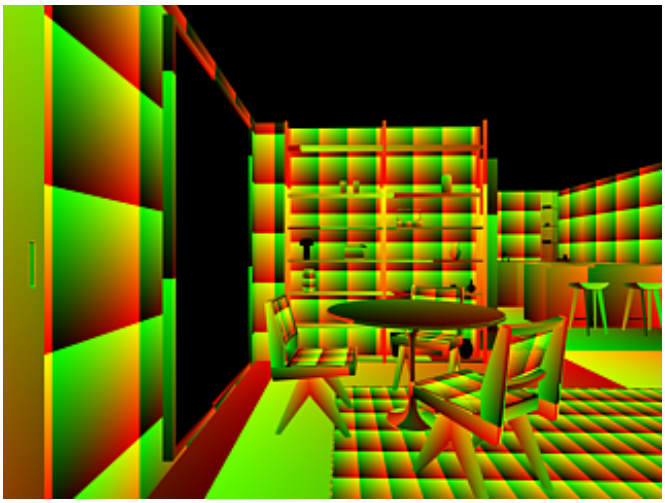


Figure 3.1: Texture coordinates element

We will show how we use this prediction in section 3.3 later in this chapter. While investigating how IRN network improves its prediction over time during our experiments, we noticed very strange behaviour with surface normals element, as show in figure 3.2.

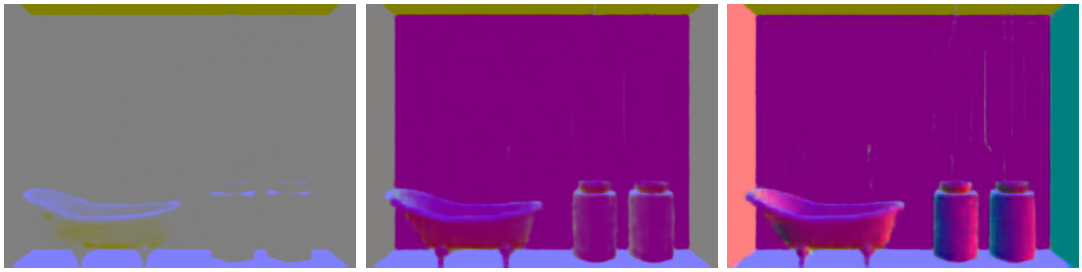


Figure 3.2: Surface normals output during training; from the early parts of the training (left) to final parts of the training (right)

As we can see, network first learned where ceiling and floor are in the input image, then started grasping concept of walls opposite to the camera and then finally learned fill-in the side walls. This happened on all training images we investigated. This was surprising and prevented the network to learn good normals representation early in the training. The underlying issue had to do with how data for this element were created: vectors were in global coordinates instead of camera coordinates. This meant that if the scene was rotated horizontally in space, normal vectors would look dramatically different on most of the object surfaces, with the exception of ceilings and floors, which were usually equally positioned in global coordinates. After we came to this realization we changed normal and view vectors data in our dataset, which lead to much stable training for these two elements.

Compared to our previous work [Jurčák, 2020], we decided to remove the RAR network from our pipeline, as it was only necessary because the original authors wanted to train networks on real photographs, for which they did not have ground-truth data. As we already had more labeled data to train from than what was included in their original dataset, we chose to get rid of this network completely. With RAR not being part of our pipeline anymore we looked at other parts that could get optimized, which turned out to be environment map prediction. Environment lighting was estimated by IRN and used only by RAR's training loss and thus become obsolete to predict, so we removed the part of IRN network that was responsible for the light approximation. As this light approximation was only used in validation of predicted render elements, we decided to only use it for fine-tuning of the IRN network, which, as consequence, meant that we also didn't have to use our direct renderer for all the training, but just for fine-tuning as well..

The simplification of IRN training loss had another hidden benefit – as we discovered, using our implementation of direct renderer presented huge bottleneck mainly due to its GPU memory consumption. When we removed part of the loss that was using the direct renderer, we were able to train with four times bigger batch sizes, which sped up our training by the same factor. This was substantial improvement, as previous training of IRN network took almost a week on much smaller dataset, which would be unfeasible for us when scaled to our current bigger dataset.

3.2 Material Segmentation

At first we tried to follow the transfer learning approach for material segmentation. As our primary goal is not to know precisely what is the class of the segmented material, our solution focused on material segmentation in the image without classification, which slightly simplifies the problem. When initialized, most architectures described in the previous chapter take number of classes as an argument, with this number describing how many feature maps should the model have in the output layer. To get a segmented image from neural network without post-processing (as we do not have the exact number of classes for materials in real world), we trained DeepLab model with 3 feature maps as output to mimic RGB image. To our surprise, the model was not able to learn underlying segmentation, even when trained from scratch, as shown in figure 3.3.

(a) Original image (b) GT segmentation (c) Predicted segmentation

Figure 3.3: Incorrect segmentation by DeepLab model

Similar issue would arise if we decided to replicate Mask R-CNN paper by He et al. [2018]. Segmentation there was performed only for certain number of classes, which is not satisfactory for our use case as we don't care about the particular material category but only need to segment the materials from each other.
To solve this problem, we chose different architecture, in particular the subnetwork that we are using in IRN for estimating normals or albedo. This architecture had no problem to learn underlying segmentation, as presented in figure 3.4. We named this network Material Segmentation Network, or MSN for short. Exact architecture of the model is described in section 5.2.3.

(a) Original image (b) GT segmentation (c) Predicted segmentation

Figure 3.4: Proof of work - MSN

As part of our efforts to simplify the segmentation task even further, we also tried amending our material segmentation data to only include edges between different materials, as shown in figure 3.5.

(a) Original material segmentation (b) Black and white material segmentation

Figure 3.5: Comparison of original and altered material ID element

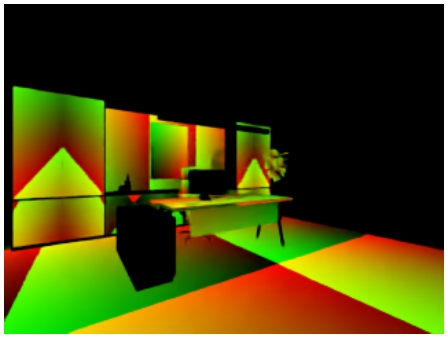
With this data set as our desired output however the network was not able to learn anything and mostly predicted full white image for any kind of input image: this was understandable, as almost all the pixels in the ground-truth image were white. We tried several architectures with various depths, but none of the models worked and although we could try to use different weights for the black pixels (e.g. black pixels having weight 1000× more than white pixels) we decided to abandon the effort, as we were already satisfied with the results we were seeing for our original problem setting.

3.3 Texture transfer

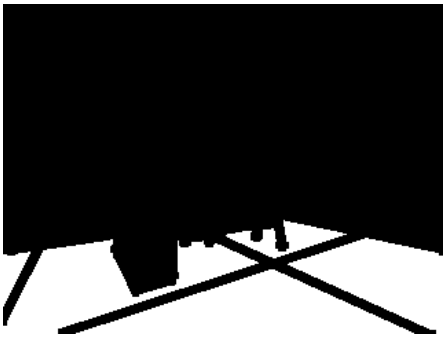
As we established earlier in chapter 1, no texture transfer solution will be perfect because of the texture deformation after the texture is applied on surface, which leads to incorrect approximation of the texture mapping function. In section 3.1 we discussed how we can estimate this mapping from the enumerated function values by optimizing for the most probable function yielding these values, and by obtaining its inverse we could arrive at the original texture.
After seeing that ResNet block model architecture worked also for segmentation, we were eager to use it for texture extraction as well. This problem setting however turned out to not be ideal usage for ResNet architecture, as we were not able to output sufficient texture quality. Results were blurry, which can sometimes happen when using L_2 loss, but we were not able to achieve significant progress with L_1 as well. Some materials can have small presence in the overall picture, so we attempted to simplify the problem by only trying to train network on materials that occupied more than 10% of the whole image but again to no success.
After reading about perceptual losses – losses that operate on higher-level features than simple pixel-to-pixel losses [Johnson et al., 2016] – we experimented with *SSIM* during our training, but this turned out to be yet another dead end.
As the last resort, we tried different approach: instead of trying to find the inverse to mapping function from function values directly, we first simply re-projected the texture using the obtained texture coordinates onto unit square, but this can output texture with holes or other imperfections. This however, is perfect environment to apply inpaiting network [Jain et al., 2023]. The authors

specifically tested their network on masked textures to asses performance of the model, with outstandingly pleasant results even for very high spatial resolution. We thus decided to use the network as-is, without any training on our part. Example of texture extraction from single material is shown in figure 3.6.

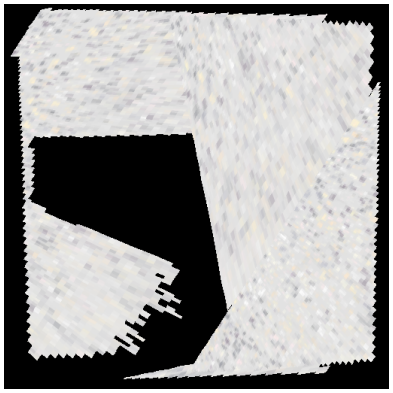
(a) Original image



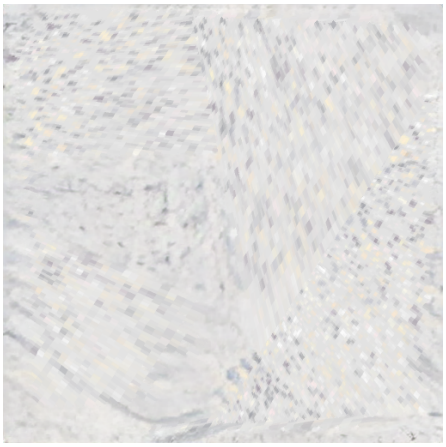
(b) Predicted texture coordinates



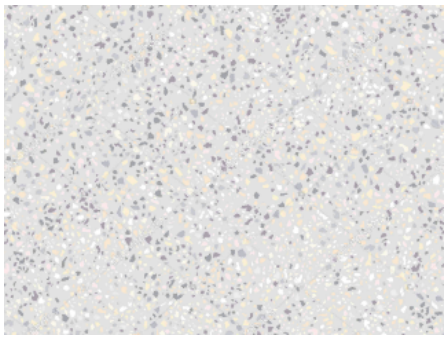
(c) Predicted material segmentation



(d) Re-projected texture using texture coordinates



(e) Inpainted texture



(f) Original texture used

Figure 3.6: Example of single texture extraction

3.4 Material Picker Pipeline

Our whole pipeline for material texture extraction is shown in figure 3.7.
Based on the results from previous work in this area (which was mentioned in chapter 2) we believe our approach is the first attempt to solve the texture

transfer problem in a holistic way - that means, without any restrictions on shape, illumination or the underlying texture used.

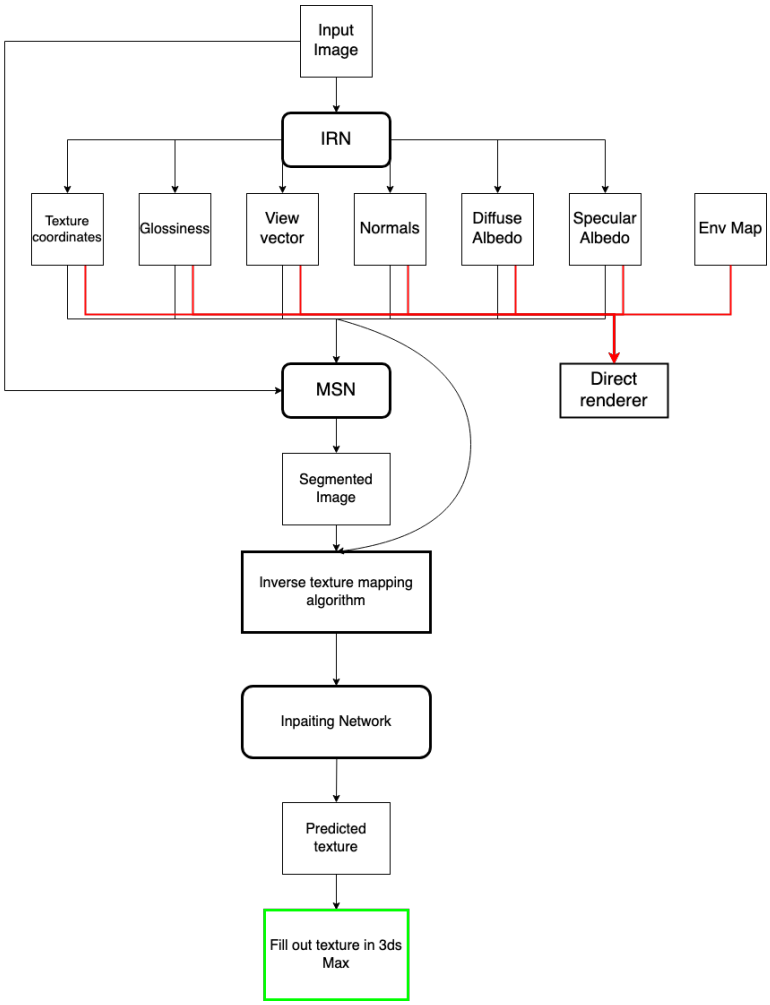


Figure 3.7: Our texture extraction pipeline; red line represents potential fine-tuning of IRN

4. Dataset

The main goal of machine learning is to gain the ability to generalize well on new, previously unseen data, in our case real-world images. This generalization is often only possible if the testing data comes from the same distribution as the training data. This distribution is in general difficult to obtain, especially in computer vision and computer graphics where we usually work with real-world imagery, for which it is problematic to obtain ground-truth data. While we can collect depth and normals of a scene via depth sensors (as was done in [Nathan Silberman and Fergus, 2012]), it is complicated to generate data for albedo, lighting or other intrinsic scene properties.
We can however overcome the issue of generating high-quality ground-truth data with physically based rendering. When we render a scene following physically based techniques (thus, physically simulate light transportation in the scene), we can generate real-world like images for which we can obtain many properties of the rendered image, hence bridging the gap between synthetic datasets and real images.

4.1 Main images and render elements

Because of the ongoing lawsuit with regards to SUNCG dataset [Futurism, 2019] (used in several works mentioned in chapter 2) we could not have used this dataset in our project, so we decided to render our own dataset using PBR techniques to match the required image quality. At first, we started generating the dataset from roughly 140 scenes downloaded from Evermotion website [Evermotion, 2020] by placing around 10 virtual cameras inside every scene using 3ds Max [Autodesk, 2020]. Each scene was then exported to *.vrscene* file and rendered via V-Ray renderer [Chaos Group, 2020a] from the viewports of these cameras to produce unique geometry for every camera view, with example in figure 4.1. V-Ray supports feature called render elements [Chaos Group, 2020c] that can be used to output additional render outputs alongside the rendered image by modifying the scene object file. Before rendering we therefore modify *.vrscene* files to generate ground-truth data used for inverse rendering. Examples of the main images in our dataset are shown in figure 4.2.

4.2 Light selects

To further enlarge the dataset, we made use of V-Ray's Light select feature [Chaos Group, 2020b] which renders additional images by using some subset of lights present in the scene. We initially started with using only $\frac{1}{10}$ of the lights present, but we found out that V-Ray optimizes computation for the main image only, which has most of the lights turned on. When only a subset of the lights is turned on for the light select element, rendered image can be quite noisy, or due to poor selection of the lights it can even be full black image.
We thus tried to do it the other way around - we used $\frac{9}{10}$ of all lights in the scene for each light select element. This approach at times produced images

Figure 4.1: Different camera views for the same scene

Figure 4.2: Examples of images in our dataset

Figure 4.3: Different lighting for the same camera view suitable for training

that were very similar to each other and because we want to have diverse set of lighting conditions present in the rendered data, we couldn't use these as well. By combining these two approaches we generated up to 20 additional images per scene under different illuminations, but usually only 5-7 were suitable for training for most of the scenes.
Examples of good light alternations are shown in figure 4.3.

4.3 Environment maps

As we decided to replicate approach showed in Sengupta et al. [2019] we also had to include environment maps into our dataset. To ensure that we had enough maps for training, we opted for combination of publicly available HDRI maps on HDRI Haven website [HDRI Haven, 2019] - with 105 maps - and our own dataset of environment maps by generating 360° panoramas of $\frac{1}{8}$ of size of the main image for every scene, yielding around 11 thousand environment maps.
In total, we have about 55× more environment maps available than what was used in Sengupta et al. [2019].

4.4 Material segmentation

To generate data for material segmentation, V-Ray provides Material ID render element containing indices of all directly visible materials in the scene, one per pixel. We modified output of the original render element by replacing each index by an RGB color computed by taking the most frequent diffuse albedo ρ_d^* and specular albedo ρ_s^* in all pixels with the same index and combining them using formula:

$$\frac{\rho_d^* + \rho_s^*}{2}$$

In our testing, we found this to work well in assigning different colors to different materials and not overlap too much. One problem, however, arises. For now, we do not have to know the values of diffuse and specular albedo that made the final pixel value in the segmented image. If we wanted to get those values (for example, to adjust values predicted by neural net), we would have to choose an invertible coding.

4.5 Texture data

As our main goal is to transfer textures from the input image, we also need to know which texture was used for each material. We extracted these as separate step of our data generation pipeline by using the Material ID element we utilized for material segmentation. For each material ID, we render a scene where we place square tile on which we map the material texture and position the camera directly above this tile.

Figure 4.4: Texture extraction example, here showcasing extracture of diffuse part of the texture

4.6 Cloud data

However, as the process of placing the cameras inside the scene manually is time-consuming - when you place a camera in the scene, you have to render camera preview to assure you accidentally did not hit a wall - and can't be parallelized, this approach would not allow us to create dataset of an appropriate size. We tried some automation of the camera placement, but we couldn't make the procedure robust enough to always lead to usable outputs. Therefore, we had to explore other options to extract data that we needed.
As this project was done in collaboration with of Chaos Group [Chaos Group, 2023b] we had access to data that customers uploaded to Chaos Cloud [Chaos Group, 2023a], which is cloud-based service that allows users to speed up their rendering times by leveraging powerful infrastructure to render their content remotely. Users upload data to this service daily, so we had abundance of data, but as these were user scenes, which a lot of them were not meant to be final renders we realized we would have to filter out a lot of scenes that were not finished or were not indoor scenes (as users of our tool would most probably be indoor architects and would use the tool for indoor scenes only). Examples of these kind of scenes are in figure 4.5.

Figure 4.5: Examples of incorrect user scenes that had to be filtered out

As we wanted to reduce time needed to produce high quality data, we extracted correct data in several steps, leading to our **data extracture pipeline**:

1. Download batch of scenes from Chaos Cloud
2. Render scene previews in the batch
3. Filter out non-indoor or unfinished scenes, move correct scenes for final render
4. Pre-process correct scenes for final render
5. Perform final render of the remaining scenes in the batch (generating main image, render elements, environment map and texture data)
6. Filter out scenes that still have issues, which included:
    - Check main images for noise
    - Check environment maps
    - Check light selects
    - Check render elements
    - Check texture elements

One batch of scenes usually contained around 500 scenes, from which only 100 were usually indoor scenes with enough quality to be used for our project. Scene previews were rendered for 30 seconds only and final render was capped at 20 minutes.
If some images were still noisy in step 6, we would re-render them for about in hour, but sometimes even this was not enough to achieve sufficiently low noise level and these scenes had to be removed. Scenes were also removed if they contained malformed data, e.g. missing material IDs (which messed up texture extraction) or all black render elements.

4.7 Filtering app

To help with filtering such massive amount of data, we created a small app using Flask [Pallets, 2023] for better visualization of results and to automate

manual work needed during filtering. The app was used in steps 3 and 6 of our data extracture pipeline to conveniently see and filter several hundreds of scene images. For checking validity of the render and texture elements, each scene is evaluated separately. Screenshots from working version of the app can be found in appendix B.

4.8 Higher resolutions

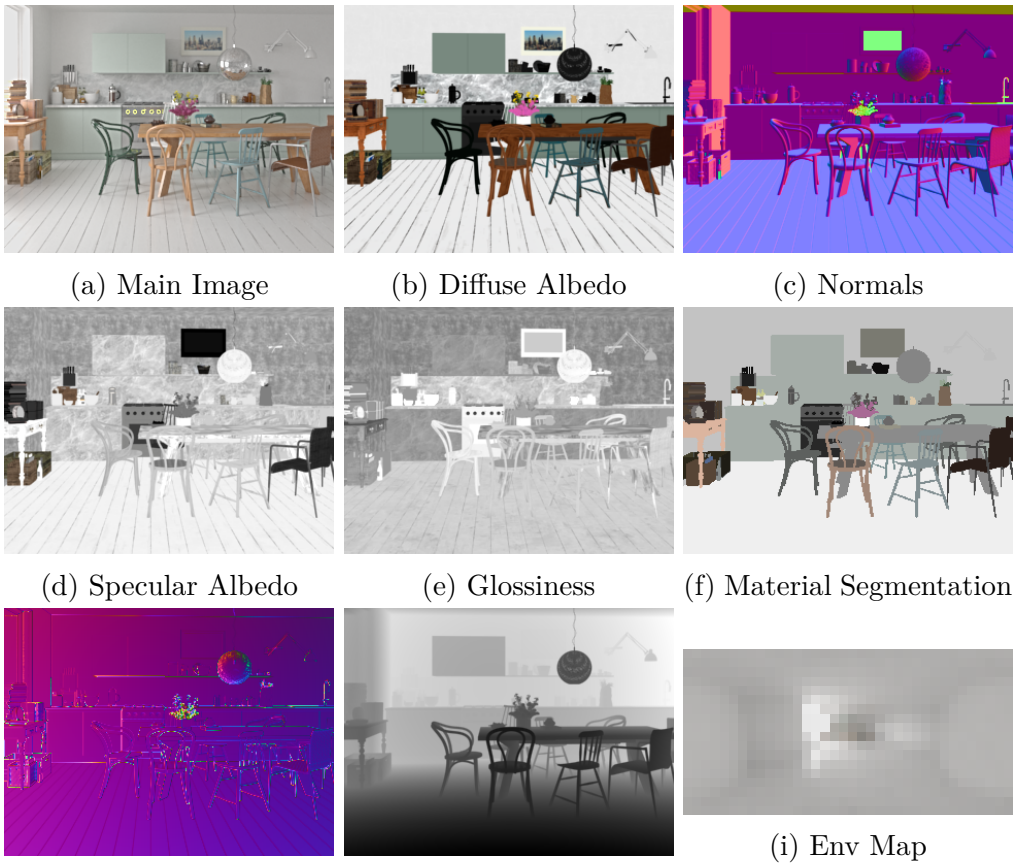
As we initially followed approach described in Sengupta et al. [2019], where authors used input images of spatial resolution 240×320 (height $\times$ width), all our data were rendered in this resolution as well. But as users might use our texture extraction tool on images with much higher resolutions, we rendered more than $\frac{1}{3}$ of our dataset in 480×640 resolution, and to test how our tool would perform on even bigger images we rendered one batch of scenes in 960×1280 resolution. The reason why we only rendered some of the scenes in higher resolutions was time restriction - time needed to render image which resolution doubled grows by a factor of four, which means that when it took 20 minutes to render image with resolution 240×320, rendering this image in 480×640 would took 80 minutes.

4.9 Dataset summary

All of the rendering was done on 2 30-core CPU machines. We processed tens of terabytes of raw scene files to extract tens of gigabytes of data in EXR format, which was ten converted to several gigabytes of PNG data used for training. In the end, our dataset consists of 11146 unique scene geometries, around 11 thousand environment maps and around 60 thousand images under different lighting conditions in total. Ground-truth data for each scene include diffuse albedo, specular albedo, normals, depth, glossiness, view vector, per-pixel material IDs and texture coordinates. Compared to our own previous work [Jurčák, 2020] we have grown the dataset to 12$\times$ the initial size while adding several render elements and textures data. In figure 4.6, you can see example of ground-truth data for one scene.

Complexity of our scenes and richness of materials is unmatched to previously used datasets, which makes this data superior for training.

As V-Ray supports more than 60 render elements and most of these elements require only fraction of time to generate them when compared to render time of the main image, our dataset is easily extendable with new properties for additional work in the future.



(a) Main Image (b) Diffuse Albedo (c) Normals

(d) Specular Albedo (e) Glossiness (f) Material Segmentation

(g) View vector (h) Depth (i) Env Map

Figure 4.6: Example of ground-truth data for a scene

5. Implementation, network architecture and training

5.1 Implementation

Because of its excellent machine learning support and community, we chose to write all of our code in Python and train all the models using the PyTorch framework [PyTorch, 2020] because of its straightforward setup for distributing training on multiple GPUs.
To help with quick experiments, we developed a framework for easy experimentation setup via command line arguments. This framework took care of initialization of the models, check-pointing and reporting during training and made adding new models very straightforward. Progress of training and validation errors was tracked by Tensorboard, which we had to integrate as PyTorch does not come with any visualization tool out of the box. Also, to assess any potential visual issues we also frequently saved outputs of the networks from subset of training and validation data, which helped us investigate issues like the one we mentioned in section 3.1.
As the final result we created 3ds Max plugin that encapsulates all our networks, runs inference on image given by the user and extract texture from material specified by the user.

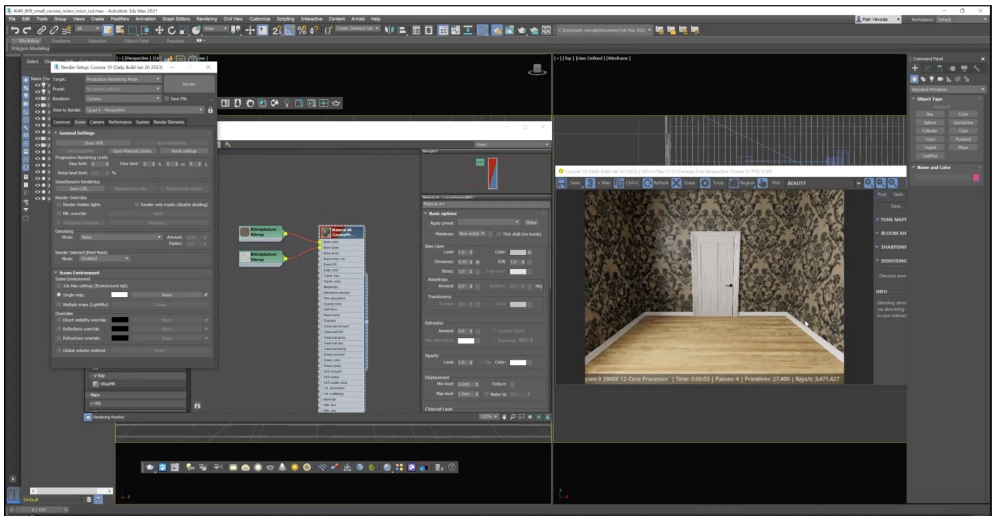


Figure 5.1: Screenshot of our 3ds Max plugin

5.2 Network architecture

In this chapter we briefly present architecture details for all models that were trained as part of our final solution. We will not present architecture of the inpainting network, as we did not altered it in any way. For details about the network and training details thus please refer to the original paper.

5.2.1 EnvMap

Our EnvMap model's architecture is defined as follows:

ReflectionPad(3) → Conv7×7(3, 64) → Conv3×3(64, 128) →
Conv3×3(128, 256) → 4 × ResNetBlock(256) → Conv1×1(256, 256) →
→ Conv3×3(256, 128) → Conv3×3Tanh(128, 3) → Upsample(l_h, l_w)

where ConvN×N(x, y) indicate 2D convolutional layer with kernel of size $N \times N$ and stride 2, x input channels and y output channels, succeeded by batch normalization and ReLU activation; ConvN×NTanh(x, y) stands for ConvN×N(x, y), but with Tanh as activation function; ReflectionPad(N) represents reflection padding with N padded items in each direction; Upsample(x, y) denotes layer that upsamples input into output with size $x \times y$ using bilinear interpolation; 4 × ResNetBlock(N) is a series of 4 consecutive block, with each ResNetBlock being

ReflectionPad(1) → Conv3×3(N, N) → BN(N) → ReLU →
ReflectionPad(1) → Conv3×3(N, N) → BN(N)

where BN(N) denotes batch normalization over input of size N × N; l_h corresponds to the height and l_w corresponds to the width of the output environment map, since these are tied to the size of the input image (concretely $\frac{1}{8}$ of the input image sides).

5.2.2 IRN

IRN consists of encoder **Enc**, defined as

ReflectionPad(3) → Conv7x7(3, 64) → Conv3x3(64, 128) → Conv3x3(128, 256)

which output is then fed through 15 × ResNetBlock for each estimated parameter. Each parameter is then upsampled back to the original input size by decoder **Dec**, defined as

TransConv3×3(256, 128) → TransdConv3×3(128, 64) → ReflectionPad(3) →
Conv7×7(64, 3) → Tanh

with glossinness as an exception, which the second-to-last layer is Conv7×7(64, 1). TransConvN×N(x, y) represent transposed convolution with kernel size N × N, x input and y output feature maps respectively.

5.2.3 MSN

Architecture for MSN is set as **Enc** → 9 × ResNetBlock → **Dec** with ResnetBlock defined in section 5.2.1 and **Enc** and **Dec** defined in section 5.2.2.

5.3 Training procedure

We have performed all of our training on two GPU servers, each of them equipped with two NVIDIA GeForce RTX 2080 Ti graphic cards. Thanks to this graphic card's big RAM, we were able to fit reasonably large batch sizes, which significantly reduced training time and stabilized training across all models.
We trained all our networks using L_1 loss. During our experiments we evaluated other loss functions as well, but all of them led to inferior results.
Building upon our previous work, we began re-training networks on the new dataset, starting with MSN which previously yielded very good results and was faster to train than other networks. This network took about a week and a half to train, including experiments with higher resolution data.
We then proceeded with EnvMap network by training it first on images produced by our direct renderer and then fine-tuned on synthetic images from our dataset by using ground-truth data for each scene and only optimizing the environment lighting. This network was not trained on images with higher resolutions due to direct renderer's substantial memory consumption, which we did not have time to optimize.
We left training of IRN network to the end, as it was the biggest network in our pipeline and we first wanted to assess if our dataset does not produce issues for other networks that had to be trained. This network took about 3 weeks to train from scratch, so we had little time to fine tune it on higher resolution data. The training is still ongoing and we hope that fine-tuning it will yield even better results, as was the case with MSN.
We chose Adam [Kingma and Ba, 2014] as our optimizer for minimizing cost function, as this optimization method outperformed all other methods like SGD by constantly giving lower training and validation error. MSN was optimized with constant learning rate $\alpha = 0.001$ and IRN with constant $\alpha = 0.0001$. Learning rate schedulers were also tested, but compared to constant learning rates did not produce better results.

6. Results

In this chapter, we present results of our trained models to see how well they generalize.

6.1 MSN

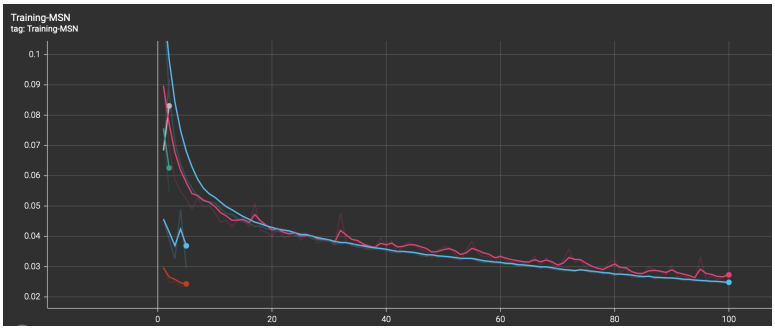
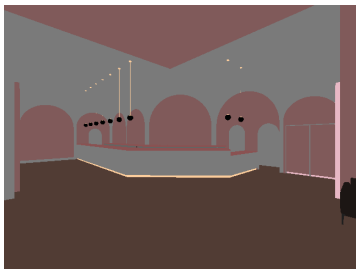
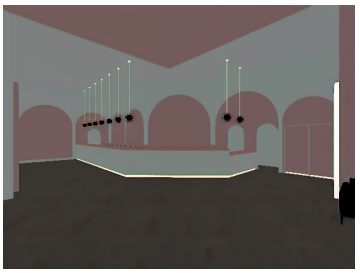


Figure 6.1: Training and validation error for MSN; long blue and purple line corresponds to training and validation errors on lowest resolution images, short blue and red line in the lower left part corresponds to training and validation errors on middle images resolution, green and grey corresponds to training and validation errors on the highest resolution images

(a) First example: Original image



(b) First example: GT segmentation



(c) First example: Predicted segmentation

(d) Second example: Original image

(e) Second example: GT segmentation

(f) Second example: Predicted segmentation

Figure 6.2: MSN results on train data

As we can see from the error progress chart in figure 6.1, we achieved good generalization of our model, with virtually no difference on lowest resolution

images (240 × 320). We then fine-tuned the model for only 10 epochs on the middle resolution (480 × 640), which still improved the model. However, as we had only one set of scenes in the highest resolution available, model started to significantly overfit the data and we thus stopped the training after just few epochs. Visual results on test data after the fine-tuning on middle resolution can be found in figure 6.2.

6.2 IRN

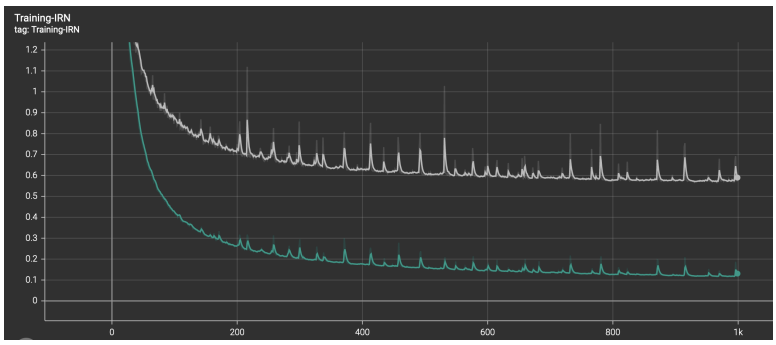


Figure 6.3: Training and validation error for our best IRN training run

As we can see from figure 6.3, we were not able to achieve similar level of generalization for IRN as we were for MSN, but results still look good visually, as can

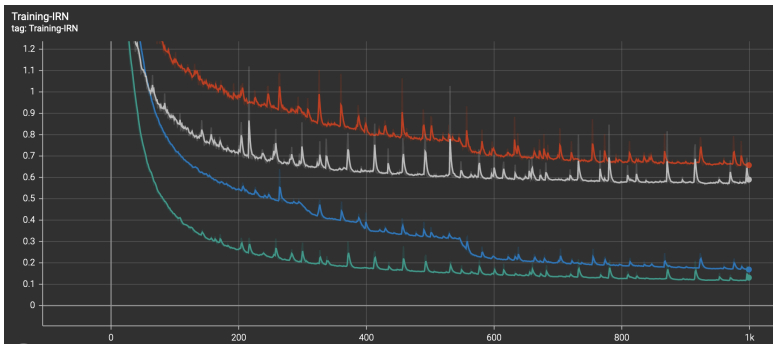


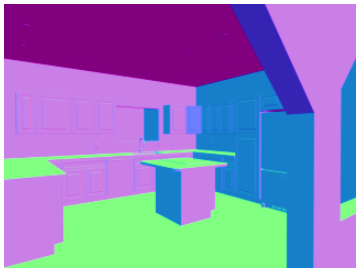
Figure 6.4: Fixing data helps with generalization; training (blue) and validation (red) error for IRN training run before several render elements were fixed, training (green) and validation (grey) error for IRN training run after the fix

In figure 6.4 how fixing problems in our dataset helped our model achieve lower error on both training and also validation sets.
Example of outputs of the IRN network on testing data are shown in figures 6.5 and 6.6. Judging by visual output only we see that the best prediction was achieved on elements that were fixed, meaning normals and view vector. As learning these elements was relatively simple task compared to other elements, where network had to learn how to decompose material appearance into several properties, it did not come as a surprise.

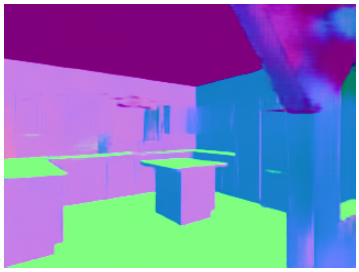
(a) Original image

(b) GT diffuse albedo

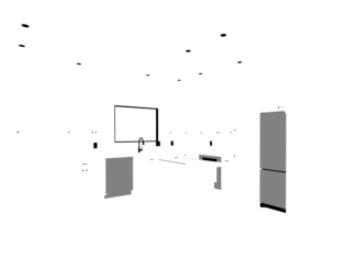
(c) Predicted diffuse albedo



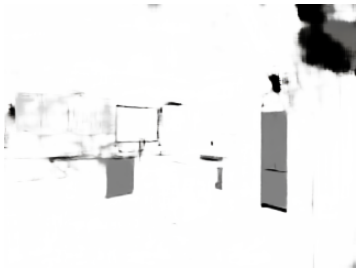
(d) GT normals



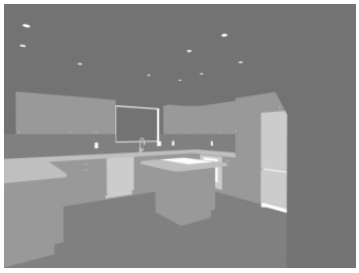
(e) Predicted normals



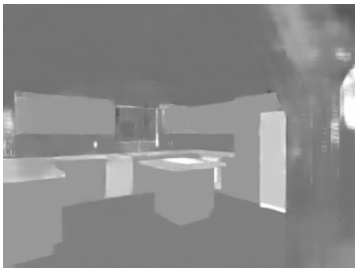
(f) GT specular albedo



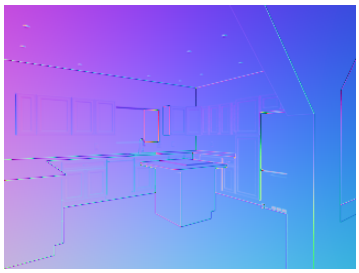
(g) Predicted specular albedo



(h) GT glossiness

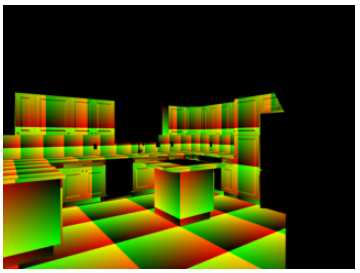


(i) Predicted glossiness

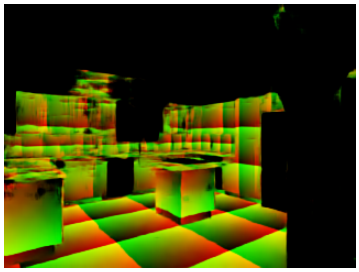


(j) GT view vector

(k) Predicted view vector



(l) GT texture coordinates



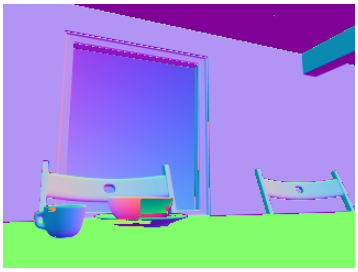
(m) Predicted texture coordinates

Figure 6.5: IRN results on test data #1

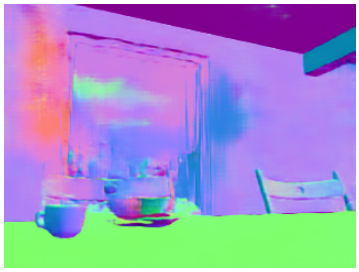
(a) Original image

(b) GT diffuse albedo

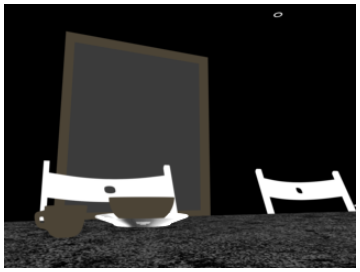
(c) Predicted diffuse albedo



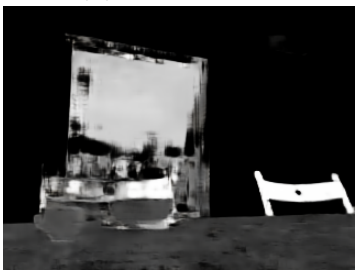
(d) GT normals



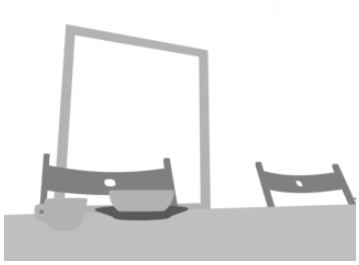
(e) Predicted normals



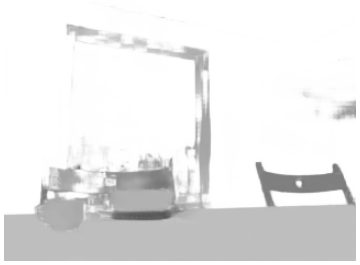
(f) GT specular albedo



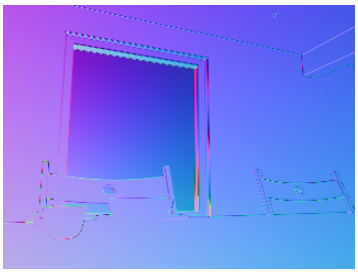
(g) Predicted specular albedo



(h) GT glossiness

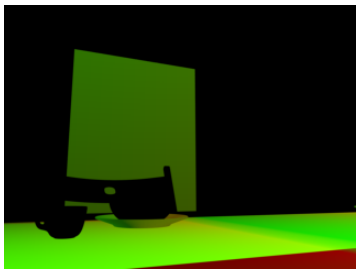


(i) Predicted glossiness



(j) GT view vector

(k) Predicted view vector



(l) GT texture coordinates

(m) Predicted texture coordinates

Figure 6.6: IRN results on test data #2

6.3 Material Picker pipeline results

In figures 6.7, 6.8, 6.9 are examples of the whole pipeline output, where in each figure: first image was original image; second row were ground-truth data; third row predicted data by our trained networks; fourth row represented ground-truth segmentation and what could be achieved by re-projecting the segmented material and then inpainting it, comparing it to reference texture; fifth row the same as fourth row but using predicted material segmentation.

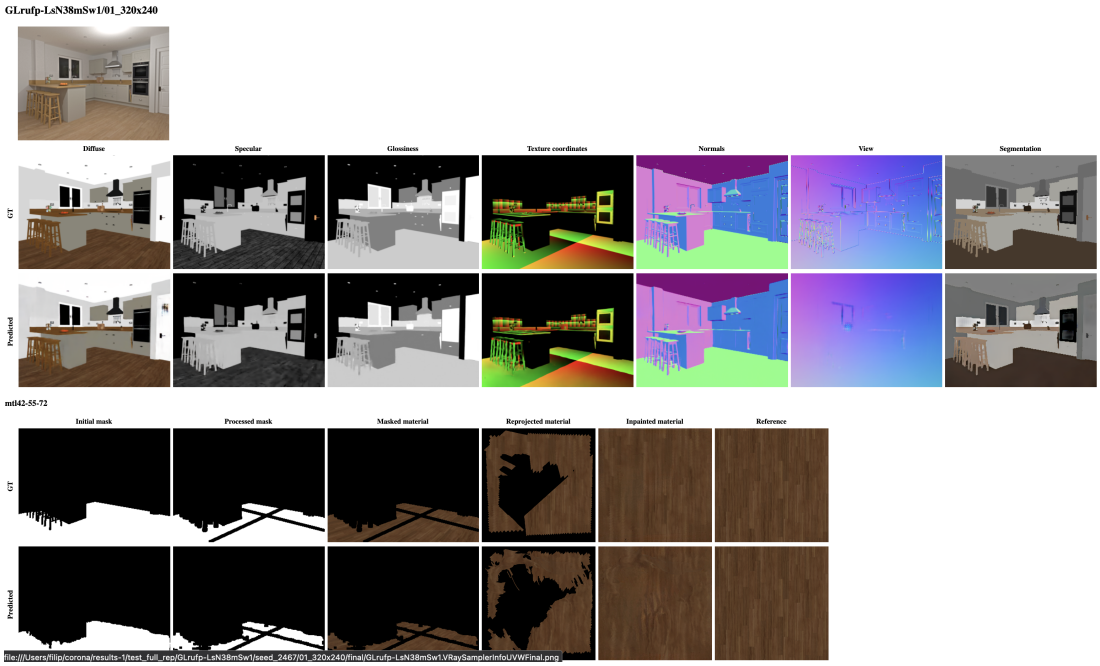


Figure 6.7: Material picker pipeline example output # 1

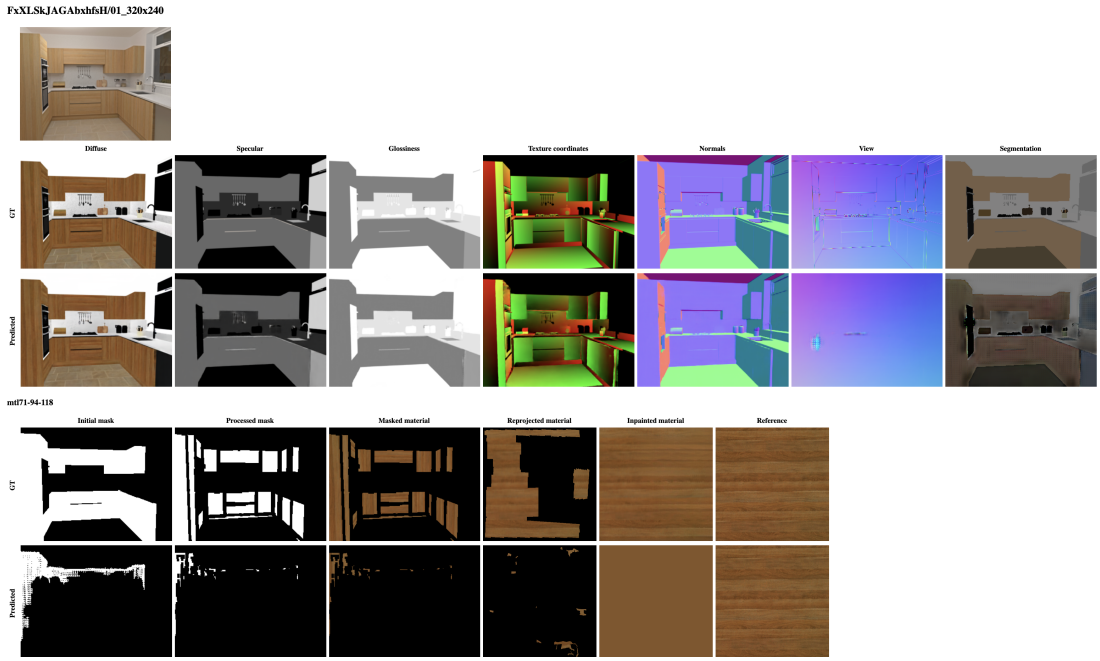


Figure 6.8: Material picker pipeline example output # 2

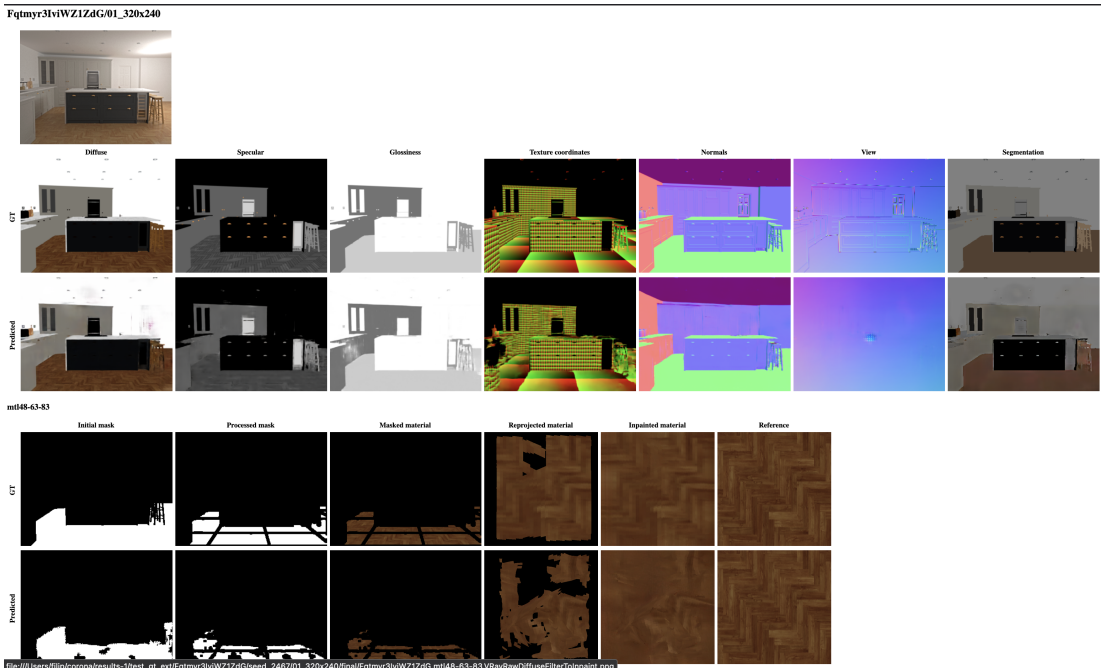


Figure 6.9: Material picker pipeline example output # 3

We can see that when render elements are very well predicted, segmentation network can segment materials more closely and thus capturing vital features needed during re-projection. As we can see in figure 6.8, the segmented part is very small and the inpaiting network can't save this kind of situation. On the other hand, there are also some very promising examples where the texture was estimated correctly: very close approximation was achieved in figure 6.7, less acceptable output in figure 6.9. More examples of the whole pipeline output are shown in appendix C.

6.4 Comparison to other work

As we explained in chapter 2, the two publications related to inverse rendering used improvements of the SUNCG dataset, which is a subject to an ongoing lawsuit [Futurism, 2019], so authors of both papers could not made their datasets or trained models publicly available. At the time of writing this thesis, the lawsuit was still not resolved, so we were not able to try and compare the trained models to our results.

Conclusion

In this thesis, we presented a method that attempts to solve texture transfer problem by leveraging per-pixel estimation of material properties in the image by training deep neural networks. We demonstrated that deep neural networks are powerful learning representations that can learn useful priors, even when it comes to such unconstrained problems like inverse rendering or segmentation. Our pipeline collectively estimates diffuse and specular albedo, surface normals, glossiness, view vector, and texture coordinates, alongside per-pixel material segmentation, from a single image. These properties are then used to reproject the segmented material (selected by the user) and inpaint parts of the texture that we couldn't predict correctly due to deformations or occlusions.

We packaged all the aforementioned effort into a 3ds Max plugin that serves as a wrapper for all our trained models. By integrating our models directly into the program, the estimated properties of a user-specified object in an image users can now apply the predicted texture on any of the objects in their scene representation and alter the texture if refinement is needed.

Before delivering this tool to the end-users we still have a lot of work to do to make our models more robust and reliable, either through generating more data or using better training procedures and architectures. We believe that fine-tuning our network on higher resolution data is of upmost importance, as the models improved dramatically after the fine-tuning. Generating more data in higher resolutions will thus be our first step to improve the output quality.

No solution will ever be perfect, but by having access to the right tools (in our case high-quality labeled dataset) we can make everyday lives of people that will use our tool little bit easier.

Bibliography

Autodesk. 3ds Max. https://www.autodesk.com/products/3ds-max/overview, 2020. Retrieved: 26-05-2020.

Jonathan T. Barron and Jitendra Malik. Shape, illumination, and reflectance from shading. *TPAMI*, 2015.

Sean Bell, Paul Upchurch, Noah Snavely, and Kavita Bala. Material recognition in the wild with the materials in context database. *Computer Vision and Pattern Recognition (CVPR)*, 2015.

Carson Katri. Dream textures. https://github.com/carson-katri/dream-textures, 2023. Retrieved: 16-07-2023.

Chaos Group. V-Ray. https://www.chaosgroup.com/vray/3ds-max, 2020a. Retrieved: 21-04-2020.

Chaos Group. V-Ray light selects. https://docs.chaosgroup.com/display/VRAY4MAX/VRayLightSelect, 2020b. Retrieved: 25-05-2020.

Chaos Group. V-Ray render elements. https://docs.chaosgroup.com/display/VRAY4MAX/Render+Elements, 2020c. Retrieved: 21-04-2020.

Chaos Group. Chaos cloud. https://www.chaos.com/cloud-rendering, 2023a. Retrieved: 15-07-2023.

Chaos Group. Chaos group. https://www.chaos.com/, 2023b. Retrieved: 15-07-2023.

Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP, 06 2016. doi: 10.1109/TPAMI.2017.2699184.

Evermotion. Evermotion. https://evermotion.org/, 2020. Retrieved: 25-05-2020.

Futurism. A startup is suing Facebook, Princeton for stealing its AI data. https://futurism.com/tech-suing-facebook-princeton-data, 2019. Retrieved: 26-05-2020.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

HDRI Haven. HDRI Haven. https://hdrihaven.com/hdris/?c=indoor, 2019. Retrieved: 26-05-2020.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018.

He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

Jitesh Jain, Yuqian Zhou, Ning Yu, and Humphrey Shi. Keys to better image inpainting: Structure and texture go hand in hand. In *WACV*, 2023.

Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution, 2016.

Filip Jurčák. Material picker: Material recognition in images using deep learning. Master's thesis, Comenius University in Bratislava Faculty of Mathematics, Physics and Informatics, Bratislava, Slovakia, 2020.

Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 12 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

Zhengqin Li, Mohammad Shafiei, Ravi Ramamoorthi, Kalyan Sunkavalli, and Manmohan Chandraker. Inverse rendering for complex indoor scenes: Shape, spatially-varying lighting and SVBRDF from a single image. *CoRR*, abs/1905.02722, 2019. URL `http://arxiv.org/abs/1905.02722`.

J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

Medium. GAN architecture. `https://miro.medium.com/v2/1*-ucVYsbDnwa2NM-f5qm_Yg.png`, 2023. Retrieved: 16-07-2023.

Jacob Munkberg, Jon Hasselgren, Tianchang Shen, Jun Gao, Wenzheng Chen, Alex Evans, Thomas Müller, and Sanja Fidler. Extracting triangular 3d models, materials, and lighting from images, 2023.

Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor Segmentation and Support Inference from RGBD Images. In *ECCV*, 2012.

Michael A Nielsen. *Neural networks and deep learning*, volume 2018. San Francisco, CA, USA:: Determination press, 2015.

Pallets. Flask framework. `https://flask.palletsprojects.com/en`, 2023. Retrieved: 15-07-2023.

PyTorch. PyTorch. https://pytorch.org/, 2020. Retrieved: 28-05-2020.

Soumyadip Sengupta, Jinwei Gu, Kihwan Kim, Guilin Liu, David W. Jacobs, and Jan Kautz. Neural inverse rendering of an indoor scene from a single image. *CoRR*, abs/1901.02453, 2019. URL http://arxiv.org/abs/1901.02453.

Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., USA, 3rd edition, 2009. ISBN 1568814690.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. Semantic scene completion from a single depth image. *arXiv preprint arXiv:1611.08974*, 2016.

Sumit Saha. A comprehensive guide to convolutional neural networks — the ELI5 way. https://towardsdatascience.com, 2018. Accessed: 02-03-2020.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015. URL http://arxiv.org/abs/1409.4842.

Vévoda, Petr and Kondapaneni, Ivo and Křivánek, Jaroslav. Bayesian online regression for adaptive direct illumination sampling. *ACM Trans. Graph.*, 37 (4):125:1–125:12, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201340. URL http://doi.acm.org/10.1145/3197517.3201340.

Tuanfeng Y. Wang, Hao Su, Qixing Huang, Jingwei Huang, Leonidas Guibas, and Niloy J. Mitra. Unsupervised Texture Transfer from Images to Model Collections. *ACM Trans. Graph.*, 35(6), 2016. ISSN 0730-0301. doi: 10.1145/2980179.2982404. URL https://doi.org/10.1145/2980179.2982404.

Wikipedia. Phong reflection model. https://en.wikipedia.org/wiki/Phong_reflection_model, 2023. Retrieved: 16-07-2023.

List of Figures

List of Abbreviations

IRN	Inverse Rendering Network
MSN	Material Segmentation Network
TTN	Texture Transfer Network
RAR	Residual Appearance Network
PBR	Physically-based rendering
BRDF	Bidirectional Reflectance Distribution Function
GAN	Generative Adversarial Network

A. Comparison of results of direct renderer implementations

Here we present comparison of results between fixed direct renderer implementation as defined in equation 3.4 and our own implementation using physically correct Phong BRDF. As we can see in figure A.1, due to inclusion of specular albedo and glossiness into the implementation, we can render much better images that are more similar to the original image. Images were rendered with an environment map inferred by our trained EnvMap model and ground-truth data for each scene.

Figure A.1: Comparison of direct render results, with original image (left), image rendered by original direct render implementation (middle) and image rendered by our own implementation of direct render (right)

B. Filtering app screenshots

In this attachment you can see screenshots from the working version of filtering app we used to extract data for our project.

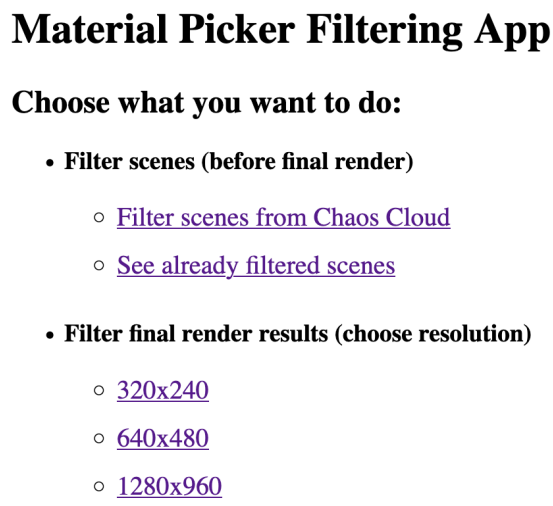


Figure B.1: Start page of our filtering app

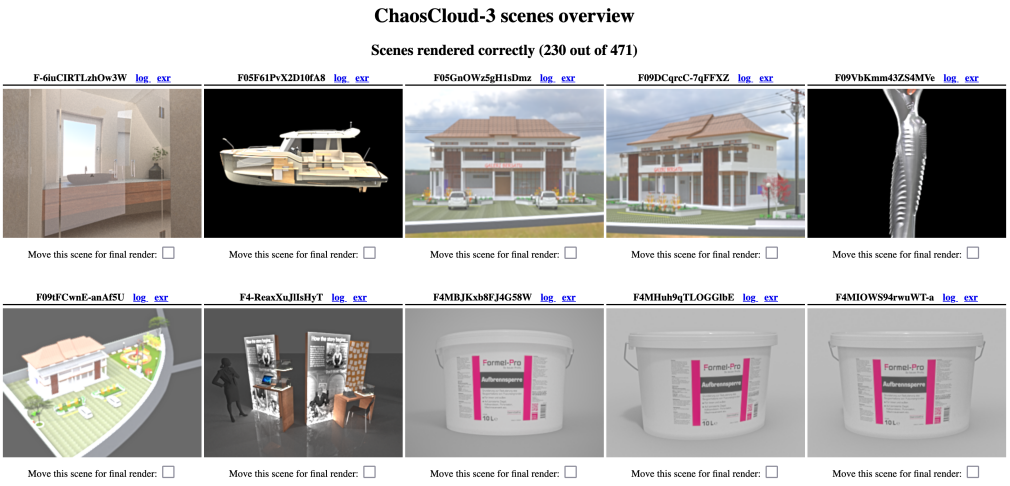


Figure B.2: Scene previews after test render

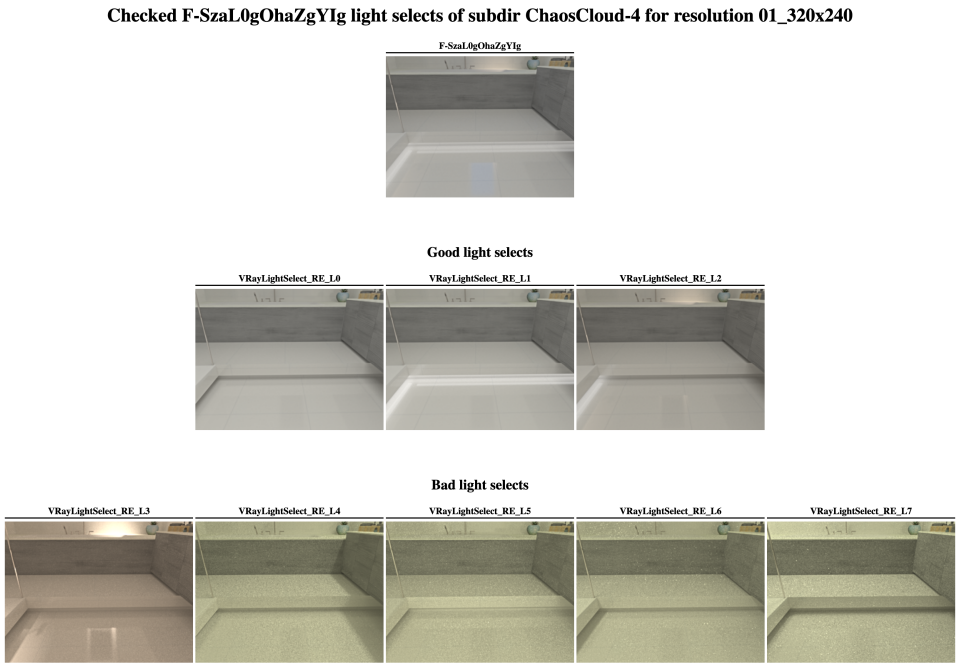


Figure B.3: Light selects subview of a scene

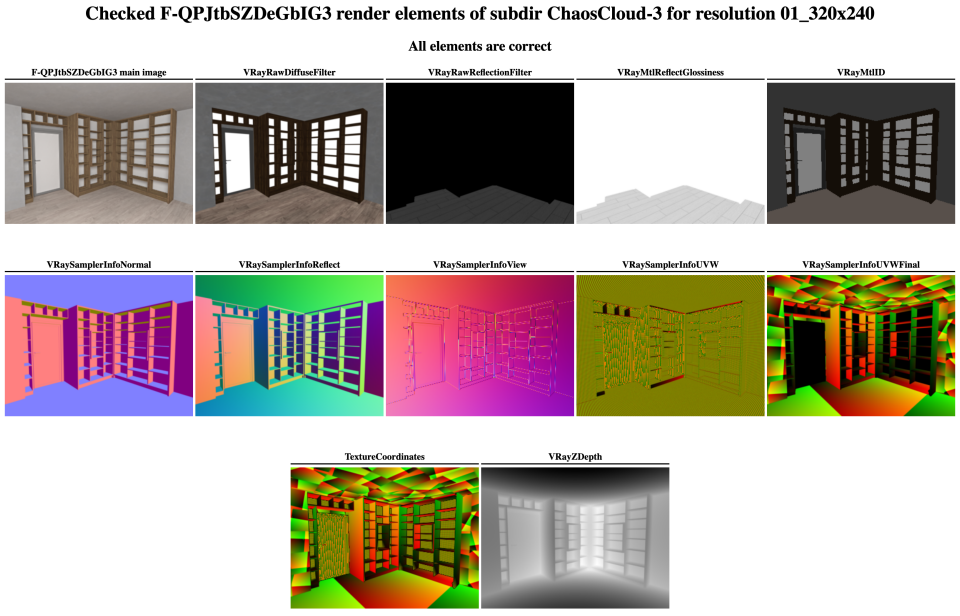


Figure B.4: Render elements subview of a scene

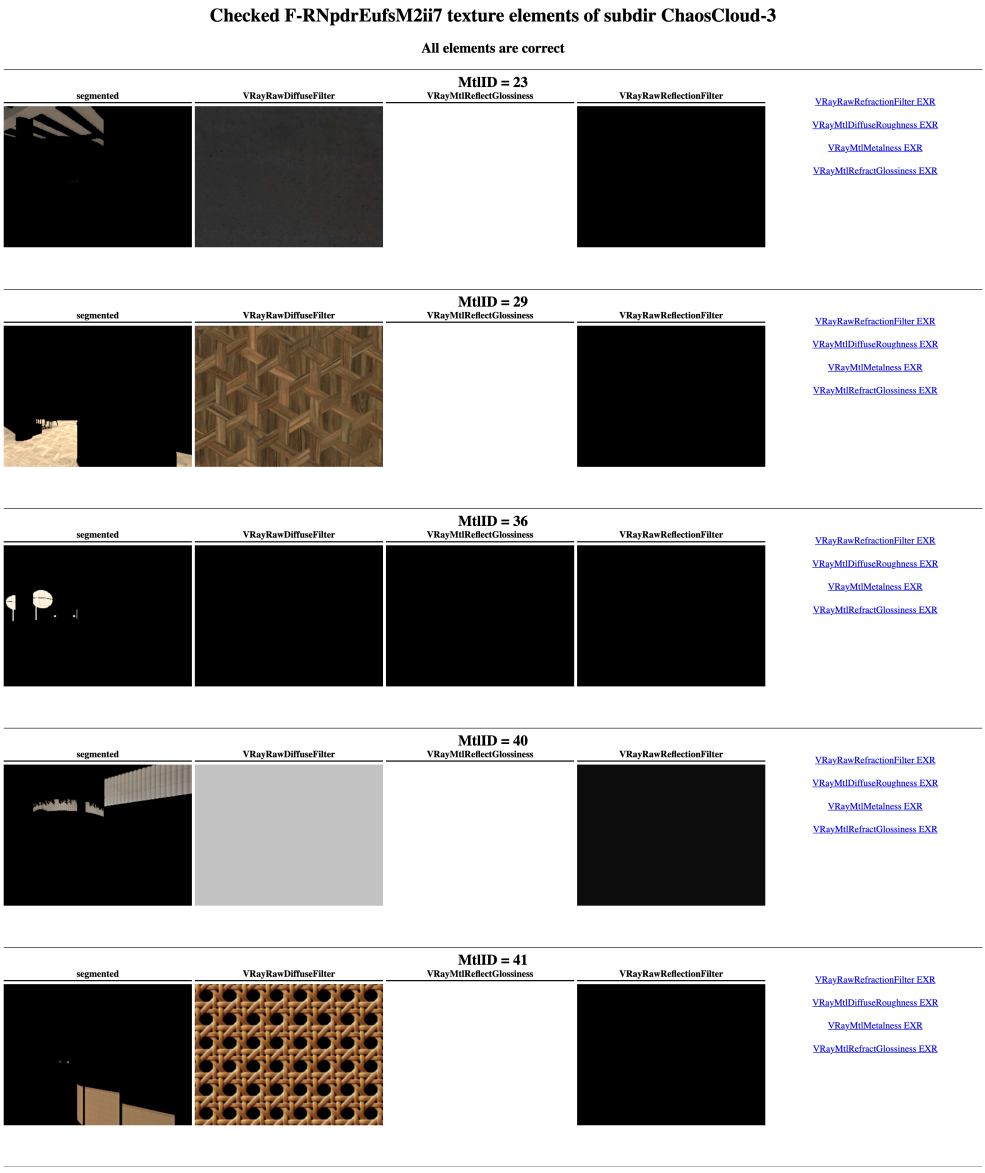
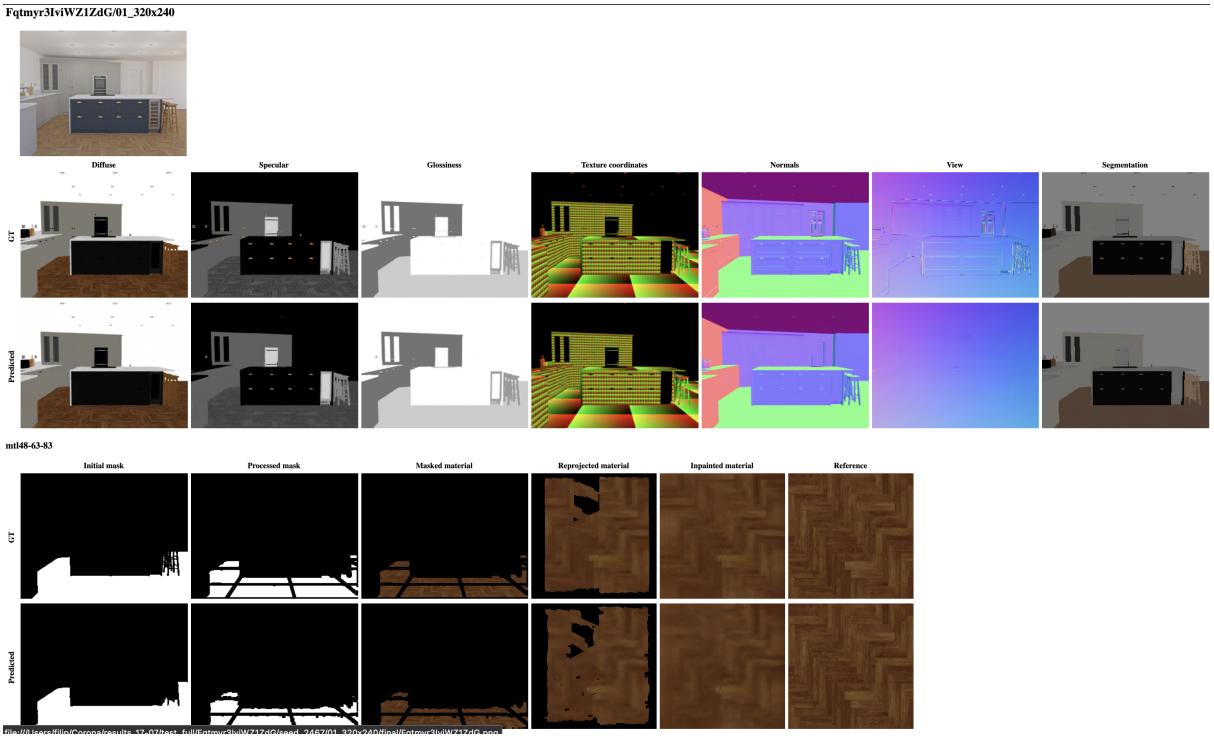
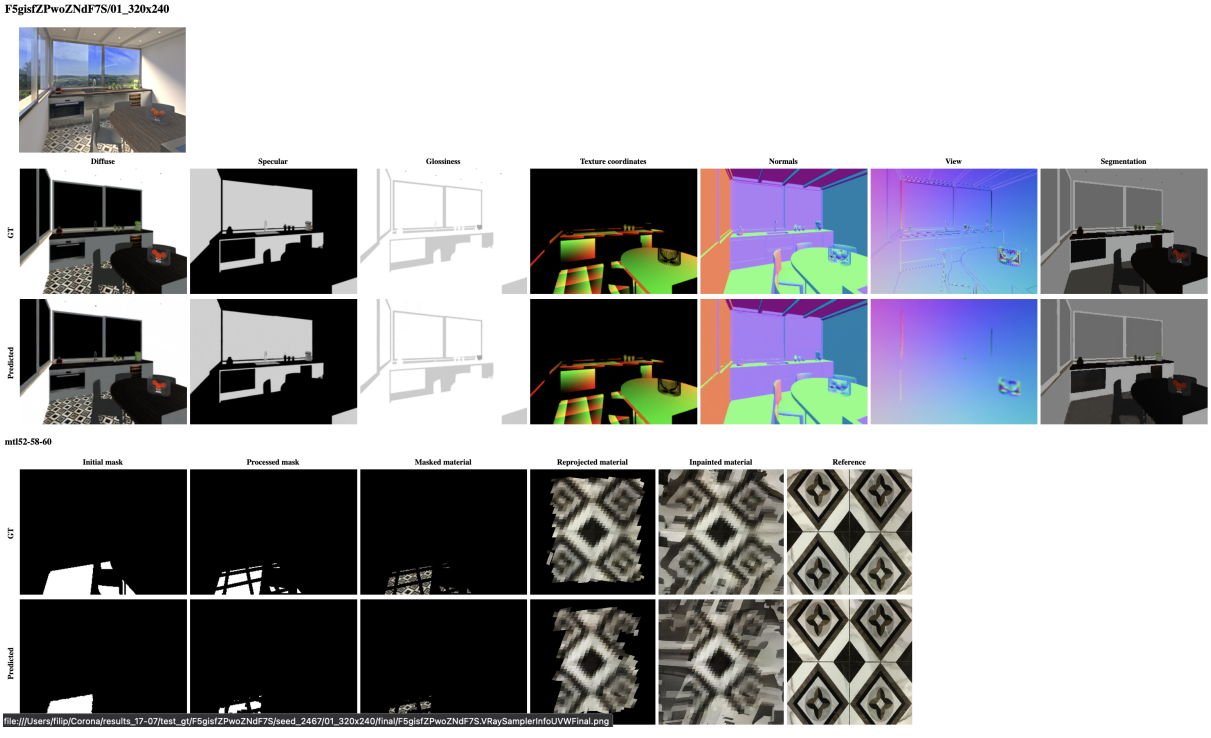
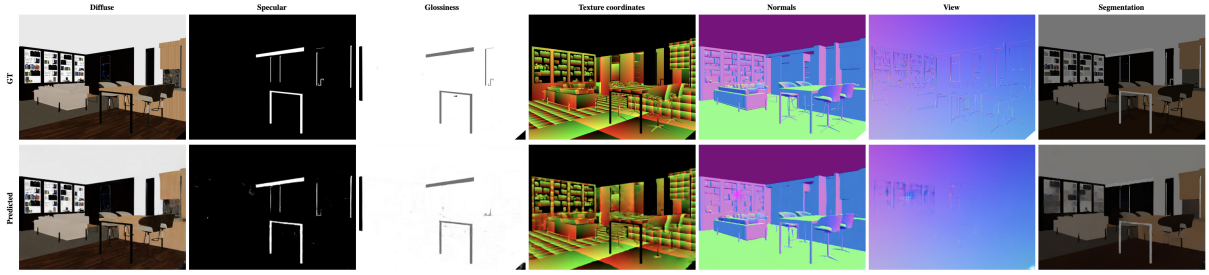


Figure B.5: Texture elements subview of a scene

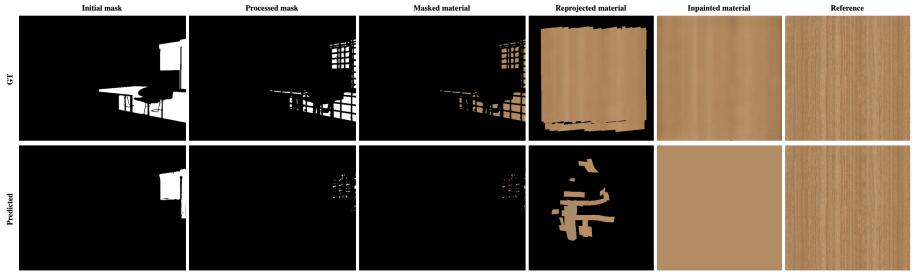
C. Material picker pipeline outputs

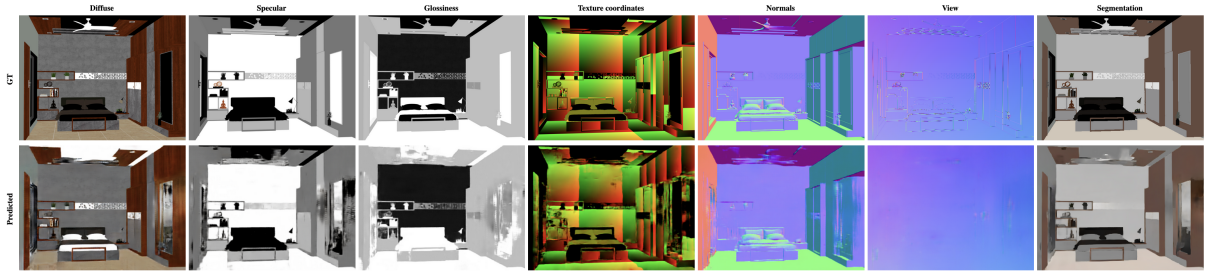




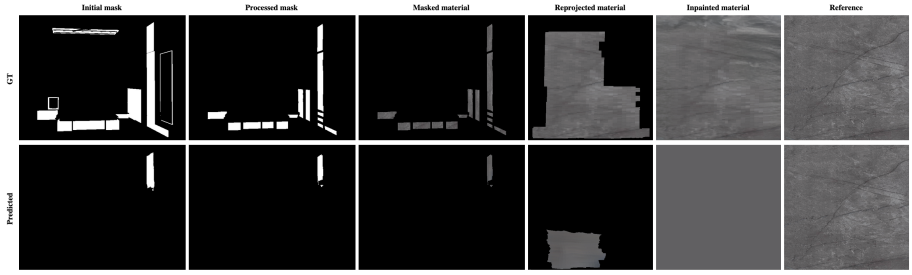


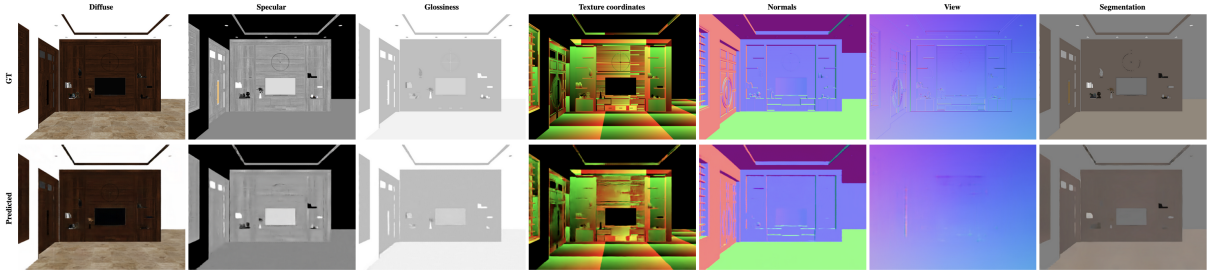
mtl45-67-91



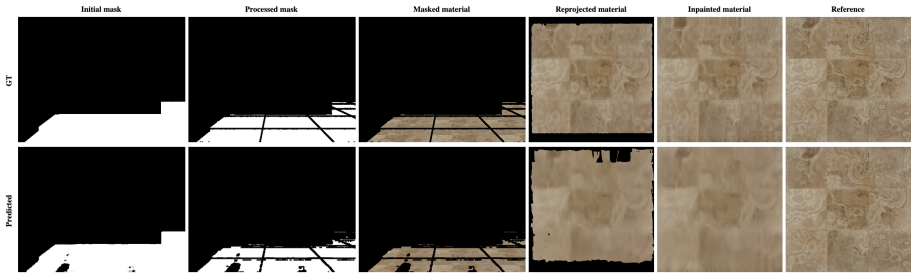


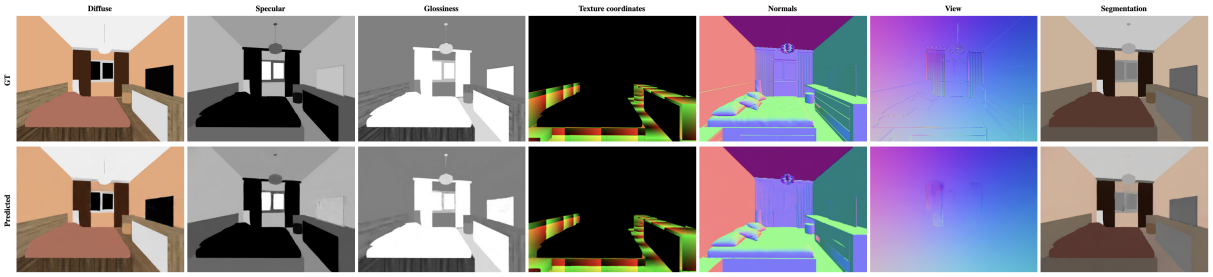
mtl143-142-143



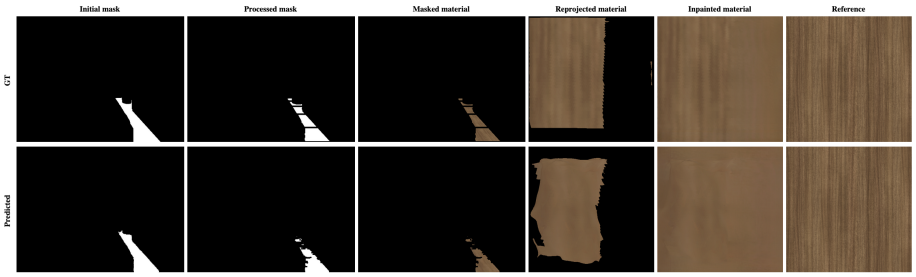


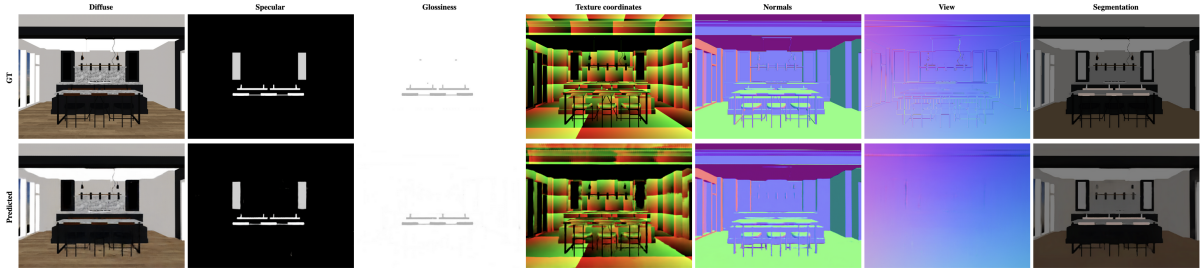
mtl122-136-147



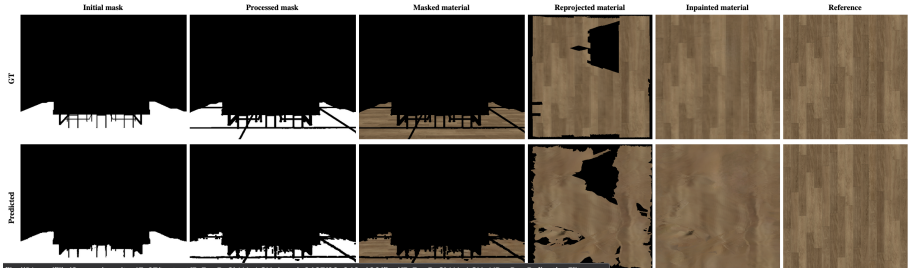


mtl102-117-133




Diffuse
Specular
Glossiness
Texture coordinates
Normals
View
Segmentation
GT
Predicted

mtl37-52-66


Initial mask
Processed mask
Masked material
Reprojected material
Inpainted material
Reference
GT
Predicted