Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Ole Martin Ingebo

Generation and Evaluation of Realistic Training Image Data for Machine Learning-Based Crack Detection

Master's thesis in Informatics
Supervisor: Rudolf Mester
June 2023

NTNU
Norwegian University of Science and Technology

DNV

Ole Martin Ingebo

Generation and Evaluation of Realistic Training Image Data for Machine Learning-Based Crack Detection

Master's thesis in Informatics
Supervisor: Rudolf Mester
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis explores generating and utilising synthetic data to improve the performance of supervised machine learning-based crack detectors in the context of surface inspection on ships. The research, conducted in collaboration with DNV, a leading maritime services and technology provider, primarily focuses on cracks that appear in ship tanks, a critical element of maritime safety. A synthetic data generation pipeline was developed, which involved creating geometrically accurate scenes, applying photorealistic textures and materials, and implementing a method to produce and segment photorealistic cracks.

The performance of a detector was evaluated based on the synthetic data generated. The results indicate that synthetic data enhance the detector's performance, particularly in identifying fine and consistent cracks. However, the synthetic data used in this study primarily represented fine and consistent cracks, indicating the necessity to diversify the synthetic data generation for broader and more effective detection capabilities. This observation underscores the importance of data variety in the training process of detectors. Further research is therefore suggested to expand the synthetic data domain to include a wider variety of crack types and environmental conditions.

Sammendrag

Denne avhandlingen utforsker generering og bruk av syntetiske data for å forbedre ytelsen til maskinlærings baserte sprekk detektorer i konteksten av overflate inspeksjon på skip. Forskningen, utført i samarbeid med DNV, en leverandør av maritime tjenester og teknologi, er hovedsakelig fokusert på sprekker som dukker opp i skips tanker, et kritisk element i maritim sikkerhet. Et system for generering av syntetiske data ble utviklet, som innebar å skape geometrisk nøyaktige scener, anvende fotorealistiske teksturer og materialer, og implementere en metode for å produsere og segmentere fotorealistiske sprekker.

Ytelsen til en detektor ble vurdert basert på de syntetiske dataene som ble generert. Funnene indikerer at syntetiske data forbedrer detektorens ytelse, spesielt ved identifisering av tynne og uniforme sprekker. Imidlertid representerte de syntetiske dataene som ble brukt i denne studien hovedsakelig tynne og uniforme sprekker, noe som indikerer nødvendigheten av å diversifisere syntetiske data for bredere og mer effektive deteksjons evner. Denne observasjonen understreker viktigheten av datavariasjon i treningsprosessen til detektorer, noe som antyder at en detektors ytelse er tett knyttet til mangfoldet av data den har blitt trent på. Dette understreker behovet for videre forskning for å generere syntetiske data med en større variasjon av sprekktyper.

Contents

1. Introduction

This thesis aims to investigate the generation and use of synthetic data for training a supervised machine learning-based crack detector, focusing on detecting cracks that appear in ship tanks. The research has been conducted in cooperation with DNV, a leading global maritime services and technology provider. The results of this research could have practical implications for the shipping industry by providing a more accurate and efficient method for detecting cracks in ship tanks, enhancing safety and reducing maintenance costs.

DNV is an accredited certification body and classification society. One of the largest industries in which they operate, is the maritime sector. DNV performs numerous services for customers in the maritime domain, among them surveying ships. DNV began a research program in mid-2018 named ADRASSO (Autonomous Drone-Based Survey of Ships in Operation). The ADRASSO project was active from 2018 through 2021 [1]. Furthermore, it was extended by a new research project named REDHUS (REmote Drone-based ship HUll Survey). The REDHUS project began in January 2021 and will continue until 2024 and is a cooperation between DNV, NTNU, the robotics company ScoutDI, bulk ship operator Klaveness, and shuttle tanker owner Altera [2].

REDHUS aims to develop and demonstrate a concept using autonomous drones for remote surveys of offshore vessels and ships. The vision is that the drone autonomously navigates and collects information about its surroundings. In REDHUS, the drone will collect visual data from a video camera and depth data from a lidar. The video data will be processed with computer vision algorithms for detecting defects such as cracks, corrosion and deformations. The REDHUS drone used for cargo tank inspection is based on the ScoutDI 137 Drone System [3], which is depicted in figure 1.1.

Figure 1.1.: Scout 137 Drone System. Image from [3].

Portions of this chapter have been copied from Preparatory Project: REDHUS Crack Detection, Ole Martin Ingebo, December 2022.

1.1. Background of Thesis

Ship Surveying
The hull, including various tanks on the ship, is inspected during a survey. Today, this is done manually by human surveyors, which may use rope access or constructed scaffolding to reach elevated, hard-to-reach areas. Common denominators are that the operations are time-consuming, and pose a risk to the personnel.

The need to be thorough in the surveys is to sustain ship safety, which the IMO (International Maritime Organisation) ensures. Classification societies such as DNV carry out inspections in line with IMO regulations. DNV has utilized drones to perform surveys in enclosed spaces, such as tanks, since 2016. The efficiency is increased, not just for the surveyors from DNV but also for the customer. The customer can potentially start their next voyage earlier due to a reduced survey time and a quicker reporting of the survey, as the drone shall ideally find issues and errors in real-time, leading to shorter decision times for all parties involved.

The total cost for a single survey can surpass 1M USD [4] when all expenses have been accounted for. These expenses come in addition to the owner's lost opportunity for revenue in the period the ship is out of service. This cost may be reduced with the help of drones, nullifying costly operations such as the installation of scaffolding, which may cost 200k USD. Emptying tanks of, e.g. methane gas and injecting oxygen or other inert gases may cost 100k USD for LNG ships. Having a vessel in the dock is also expensive. Renting an oil tanker's location in the dock may cost 100k USD for 1-2 days with reasonable rates [4].

The current drone-based inspection available to customers is manually operated by DNV personnel. The operation encompasses both control of the drone and review of the footage. The REDHUS project aims to accomplish this autonomously, and a central component is to be able to inspect the footage for cracks without a reduced need for a human in the loop as an assistance tool for surveyors. It is creating a necessity for a solution to perform surface inspection, specifically crack detection.

Crack Detection Techniques
Traditional image processing techniques for detecting cracks may be methods to detect edges, i.e. using the fast Fourier transform [5]. After an edge-detection algorithm is applied, an image is determined to have a crack. The determination is based on a threshold set for the given algorithm and the intensity in the edge image output from the algorithm. In later years automatic feature extraction has become more popular. Deep learning, a subset of machine learning that uses multi-layer neural networks, has recently had more publications than traditional machine learning approaches in terms of crack detection [6].

Different detection tasks exist that may be adopted for crack detection. They range from being more coarse to granular in how they represent their detection. The crudest is image classification, where an entire image is true or false for whether it contains, i.e. a crack or not. A more granular image classification approach is to section the image into smaller images and then classify them individually. This inherently gives an element of localization. Even more granular is object localization, which aims to localize objects of interest in the image, often drawing a bounding box around them. Object detection takes this further by first localizing objects of interest and then classifying the object that has been localized. Finally, the most granular tasks are segmentation tasks, where each pixel in the image is evaluated. Examples of these tasks for crack detection are illustrated in figure 1.2.

Synthetic Data for Supervised Learning
One of the general underpinnings for the success of supervised deep learning methods in recent years

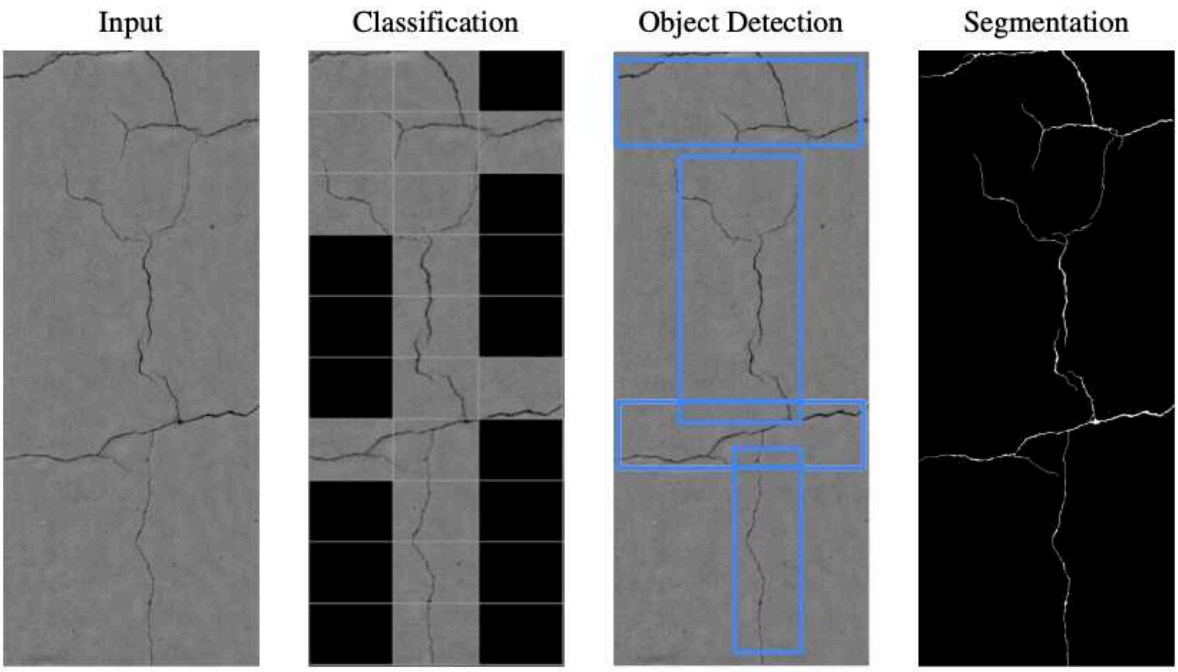


Figure 1.2.: In the leftmost image, *Input* is an image containing cracks that will be given to three different detectors, with varying detection tasks. In the *Classification* output, if the sectioned image contains a crack, they are not left as a black rectangle. The *Object Detection* output displays cracks that are first localized and have drawn a bounding box around them and are classified as cracks. In the *Segmentation* output, each image pixel has been evaluated for whether it is part of a crack. Figure from Hsieh and Tsai [6].

is access to large-scale labelled data [7]. Access to large-scale labelled data may be difficult depending on the task to be solved. Moreover, in many cases, the labelled data does not exist. However, one can find public datasets, use data annotation services, label them in-house, or synthetically generate data. From the inception of the field, computer vision researchers have employed 3D models as an instrument to enhance the performance on actual images [8]. And in more recent years, synthetic data have been used for the task of, i.e. object detection as in Sun and Saenko [9], Movshovitz-Attias et al. [10].

When training a detector, the goal is to match the source domain with the target domain and its underlying distributions. What follows is that when synthetic data is generated, it is the intent that the synthetic data, as part of the source domain, match the target domain, where the detector is expected to function optimally. Matching the source and target domain may be termed "bridging the reality gap" as in Tobin et al. [11], where the synthetic data is simulated, and the target domain is in reality. It is illustrated as a Venn diagram in figure 1.3 where the greatest overlap of the two sets D_s, and D_t, $D_s \cap D_t$ is the aim.

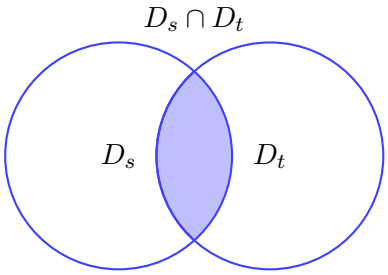


Figure 1.3.: Venn diagram of the two sets source domain D_s, and target domain D_t. When creating a synthetic dataset as part of the source domain D_s, the goal is to have the largest overlap $D_s \cap D_t$, with the target domain D_t.

1.2. Goals of the Thesis

The primary objectives of this thesis, which also serve to shape its structure, can be distilled into two key components.

Goal 1 *Build a synthetic data generation pipeline for cracks on ship surfaces.*

In a supervised learning task, an essential component is to have training data available. It is often laborious to create this data, as human annotators are involved. Instead, synthetically generated datasets may be made, rendering a semantic segmentation label alongside a colour image. The first goal of this thesis is to build a pipeline for rendering photorealistic colour images of cracks on ship surfaces, along with annotations.

Goal 2 *Train detectors on the synthetic data generated from the pipeline and examine the results.*

After synthetic data has been generated, detectors will be trained on this dataset, in addition to authentic images that have been annotated. Experiments are then performed to see whether the synthetic data improves the detector.

1.3. Requirements for the Synthetic Data Generation Pipeline

This section details the requirements for the synthetic data generation pipeline from goal 1.

Requirement	Description
R.1	The images produced by the process shall be photorealistic.
R.2	The geometric models used in the scene shall accurately represent the geometry of the objects they represent.
R.3	The textures and materials used on the geometric models shall be high resolution and accurately represent the objects' surface details.
R.4	The geometric models of the cracks shall have realistic dimensions. This includes the width, depth and overall shape of the crack.
R.5	The rendering engine used shall be capable of accurately simulating the behaviour of light, shadows, and reflections in the scene.
R.6	The lighting used in the scene shall simulate the lighting on the REDHUS drone.
R.7	The camera settings should accurately simulate the REDHUS camera.
R.8	The annotations shall be available in common formats, i.e. COCO.

Table 1.1.: Requirements related to the synthetic data generation pipeline from goal 1.

Part I.

Generation of Synthetic Dataset

2. Overview of the Chosen Approach for Image Generation

2.1. General Overview of the Pipeline

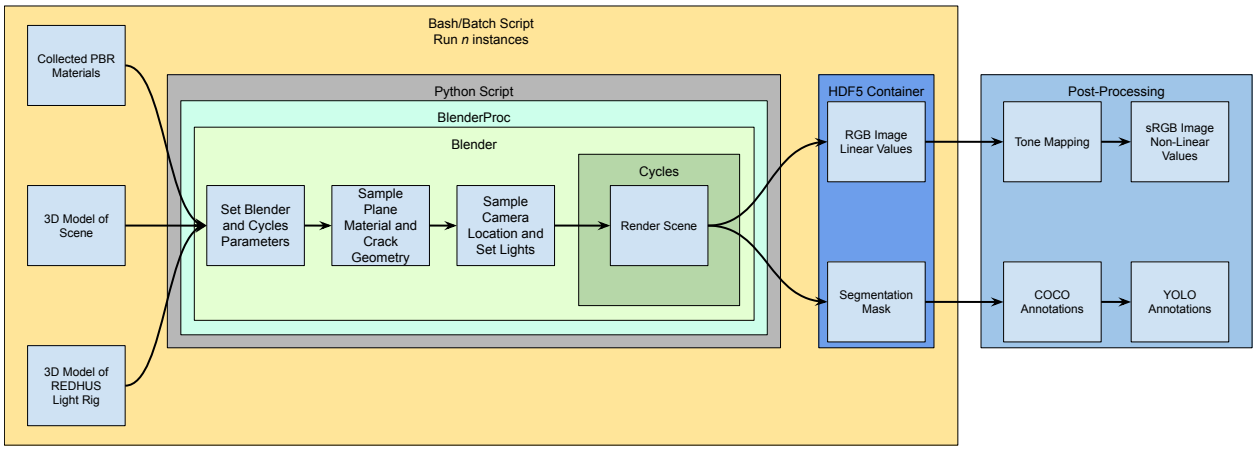


Figure 2.1.: Overview of the pipeline created to generate synthetic data for detectors to detect cracks in ship surfaces.

Figure 2.1 illustrates the general overview of the pipeline implemented. At its core is Blender, while the other components wrap around it. BlenderProc offers features needed for generating synthetic data, such as segmentation masks, and facilitating easier interaction with Blender via Python scripts.

The general flow in the pipeline is that a Python script collects pre-made 3D models and materials stored on the hard disk. A script sets the desired scene parameters before rendering, utilising the Cycles render engine. One of three materials is uniformly sampled and applied to the objects in the scene. One of the objects in the scene is then selected and replaced with an object that contains a crack, whose parameters are randomised within suitable intervals that make a realistic crack geometry. Afterwards, a camera location is sampled within the scene, oriented towards the crack, and the camera is placed along with lights to illuminate the scene.

The scene set-up is rendered using Cycles and exported to HDF5 containers that contain a pair consisting of a colour image and a segmentation mask. The colour image is post-processed with tone mapping to create an sRGB image. COCO annotations can be exported using the segmentation mask, which can be further processed to export YOLO annotations.

2.2. Description of Recurring Tools in the Pipeline

Blender
Blender [12] is an open-source, free 3D creation suite under the GNU General Public License. It is a

cross-platform application running on Linux, macOS, and Windows. Blender gives access to many 3D creation tools, such as compositing, texturing, simulations, modelling, animation and rigging, rendering, video editing, and VFX. In addition, a large and active online community of Blender users leads to a large degree of community support. In this thesis, the 3.3 LTS version of Blender has been utilised.

BlenderProc
BlenderProc [13, 14] is a procedural pipeline that aims to render photorealistic images and various annotations used to train neural networks based on Blender [12]. BlenderProc attempts to facilitate the interaction with Blender via an easy and intuitive Python API. The renders of a scene that serve as annotations with the help of BlenderProc are colour, depth, distance, surface normals, semantic segmentation, optical flow and normalised object coordinates. HDF5 containers, BOP, or COCO formats are directly usable storage options for the rendered data.

Shader Editor
A shader editor is a tool to create and edit materials for 3D objects in computer graphics. In Blender, the shader editor is a node-based interface that allows users to create complex materials by connecting different nodes. Each node in the shader editor represents a specific function, such as a texture, colour input, or mathematical operation. By connecting nodes, users can create complex combinations of these functions to achieve the desired look for their object and scene.

For example, a user might create a simple diffuse material node and then add a texture node to create a more complex pattern. They could then connect these nodes to a normal map node, adding depth and detail to the object's surface. In addition to these essential nodes, the Blender shader editor also includes more advanced features such as displacement mapping, subsurface scattering, and volumetrics.

3. Geometrical Modeling of the Scene including Cracks

3.1. Background of Geometric Modelling in Computer Graphics

Geometric modelling in computer graphics involves creating and manipulating virtual 3D objects using mathematical representations. It is a fundamental aspect of computer graphics utilised in various fields like animation, video games, virtual reality, industrial design, and architecture. Geometric modelling represents 3D objects with geometric primitives such as points, lines, curves, and surfaces. These primitives are used to construct more complex structures like meshes, which are collections of polygons. Triangles are commonly used polygons due to their simplicity and efficiency in rendering and computation. Polygons are comprised of vertices and edges and are used to define the faces of a mesh. Each face of a mesh consists of one polygon, and the polygons collectively form the surface of the 3D object. For example, four polygons can be seen on the left of figure 3.1.

Modelling objects is time-consuming, especially if a seemingly smooth mesh were to be created by hand. An approach to overcome this is the subdivision surface. *Subdivision surfaces* are a technique used in 3D modelling to create smooth and detailed surfaces from simpler base meshes. The basic idea behind subdivision surfaces is to iteratively refine and subdivide the base mesh by adding more vertices and edges. Each subdivision step divides each face of the mesh into smaller faces and adjusts the positions of existing vertices based on specific rules or algorithms [15, p.181].

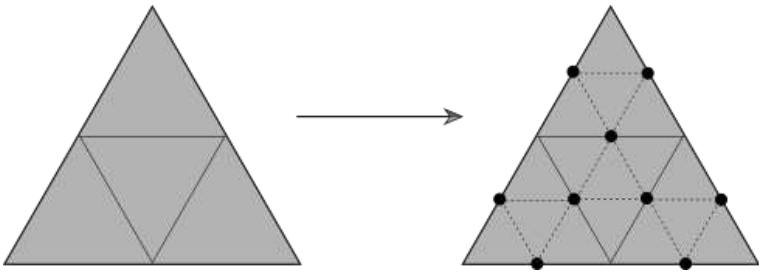


Figure 3.1.: **Left:** A mesh consisting of four polygons. **Right:** Same mesh after one subdivision step. Figure from Pharr et al. [15, p.184].

Subdivision surfaces follow a subdivision algorithm, such as the Catmull-Clark subdivision algorithm [16]. This algorithm subdivides each face by creating a new face in the centre and connecting it to the existing vertices and edges. The positions of the existing vertices are then adjusted based on averaging neighbouring vertices, resulting in a smoother surface. This process may be repeated multiple times to achieve higher levels of detail. A subdivision step is illustrated in figure 3.1.

A static subdivision surface could be more computationally wasteful, leading to adaptive subdivisions that lessen the computational burden. Adaptive subdivision is a technique used to dynamically control the level of detail in a mesh or surface. It allows for efficient rendering by allocating computational resources more selectively and optimising the level of detail in different areas of a 3D model. In adaptive subdivision, the mesh is subdivided and refined to prioritise areas where more detail is needed, such as regions with high curvatures, fine surface details, or areas closer to the

camera as in figure 3.55. On the other hand, areas with fewer detail requirements, such as flat or distant regions, are subdivided less or not at all, reducing the computational overhead. By adaptively subdividing the mesh, adaptive subdivision techniques strike a balance between visual quality and computational efficiency.

Bump mapping is a computer graphics method that simulates bumps on a surface by altering the surface normals, affecting how it reflects light. It does not change the object's geometry, only the illusion of the surface detail using a grayscale texture map. [15, p.584]. *Normal mapping* is an enhancement to bump mapping where a colour texture map is used to store surface normals for more detailed representation directly. Like bump mapping, it does not alter the object's geometry, only the light reflection. *Displacement mapping* is another method. Unlike the previous techniques, displacement mapping alters the object's geometry using a grayscale image. Pixel values physically displace the surface geometry in the direction of the surface normal, allowing a realistic alteration of the object's silhouette, but at a higher computational cost [17].

3.2. Geometric Modelling of the Scene

When generating a synthetic dataset for a supervised learning task, it is of desire to simulate the environment in which the detector will be operating. In the ideal case of the REDHUS drone, these would be scenes of ship tanks with realistic dimensions concerning the physical dimensions of the drone. A simplification of creating a ship tank is to create an enclosed cube. The enclosed cube serves as a geometric approximation of the ship tank. While a ship tank can have complex shapes, the cube provides a simple and uniform structure that can represent the overall geometric shape of the tank. Another aspect is that the enclosed cube removes external light sources, such as background illumination, as ship tanks are commonly used to store liquids, gases, or other substances.

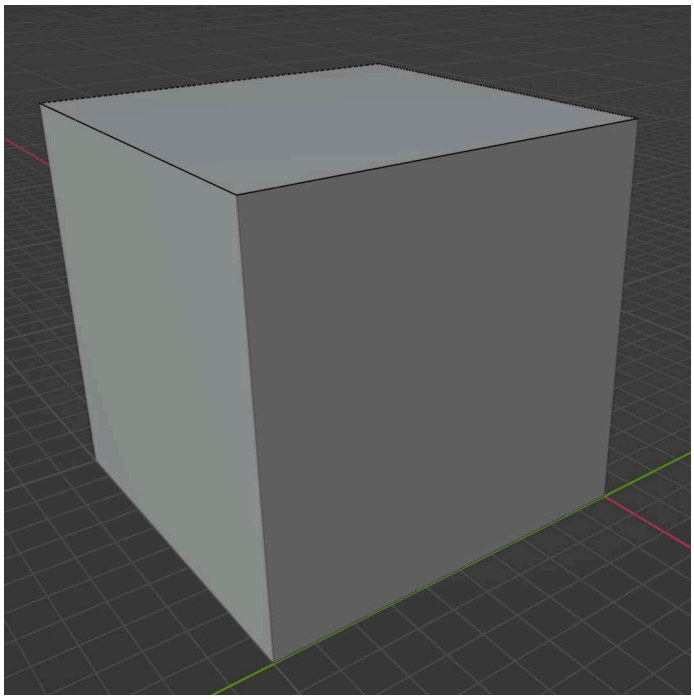


Figure 3.2.: Screenshot of Blender viewport solid shading, displaying an enclosed cube to simulate a ship tank. All renders are performed inside the volume of the cube. The walls of the cube are dimensionless, having no thickness.

The cube has a volume of $(10 \times 10 \times 10)m^3$ and is depicted in figure 3.2. The six planes are $10m \times 10m$, and all meet perpendicularly at their outer edges. Welding seams served to smooth out the intersections of the planes to remove the abrupt look. Twelve planes were placed at a 45° angle between the intersections of the walls to create the welding seams. The twelve planes were displaced using a displacement image to create the visual look of a welding seam. The intersection of three welding seams in a corner from the viewport view is depicted in figure 3.3 and rendered in figure 3.4. Figure 3.5 displays the image used to displace the weld, and figure 3.6 depicts the node setup using the weld displacement image.

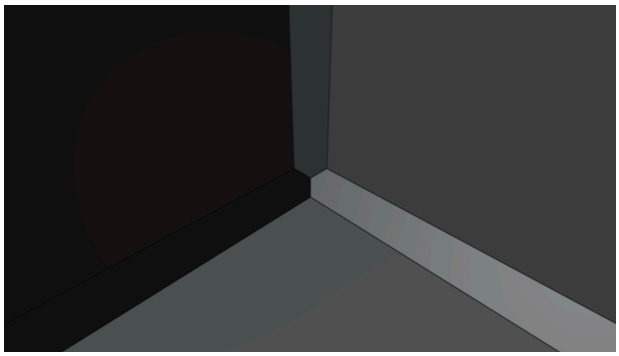


Figure 3.3.: Viewport screenshot of the intersection of welding seam planes in a corner.

Figure 3.4.: Render of the intersection of welding seam planes in a corner.

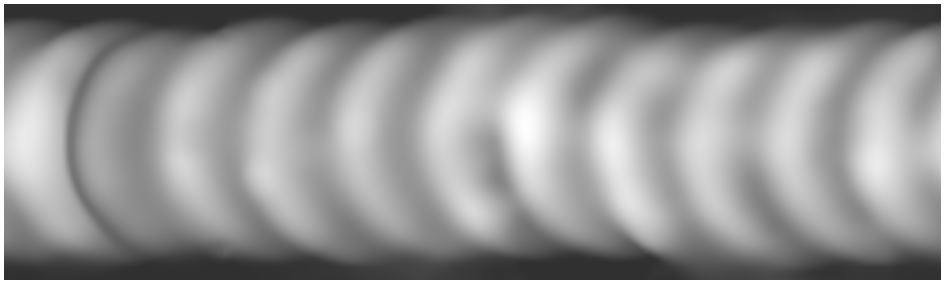


Figure 3.5.: Image from [18] used to create weld displacement depicted in figure 3.4 using the node setup in figure 3.6.

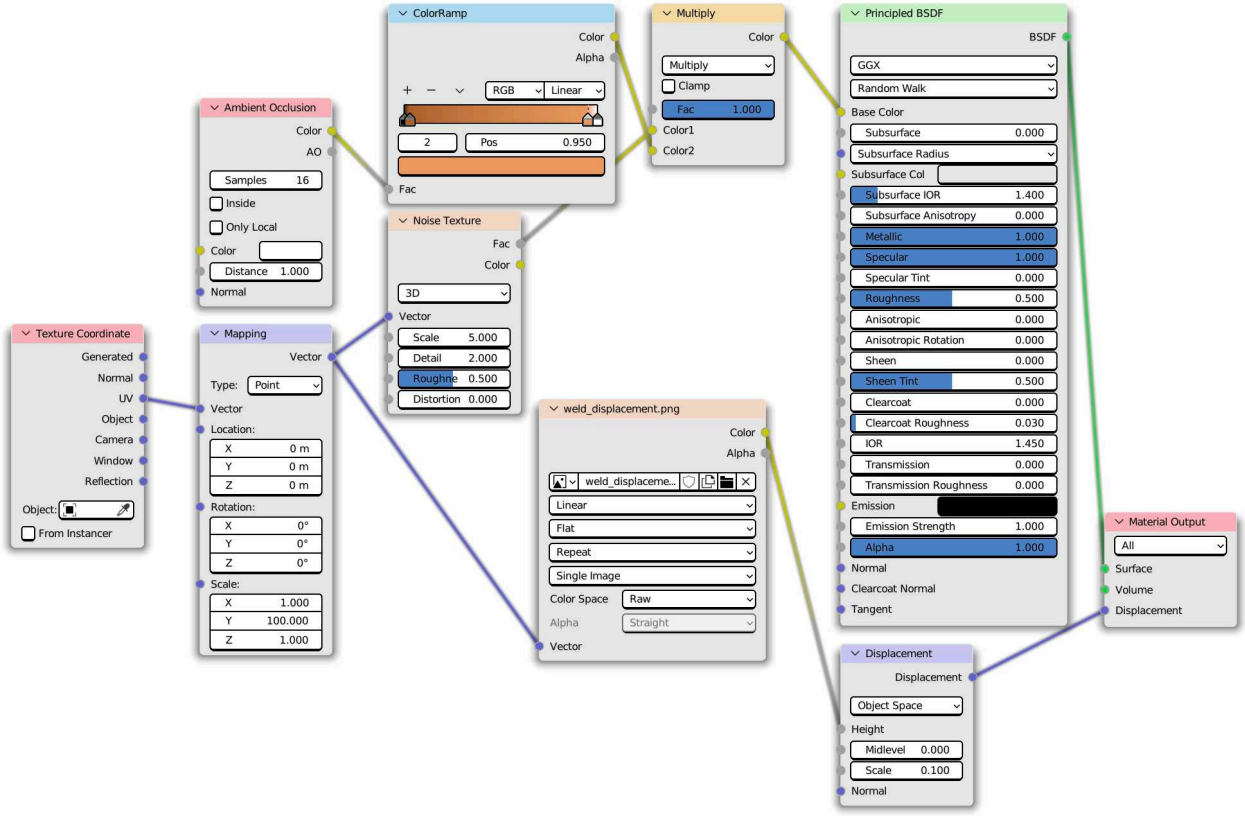


Figure 3.6.: Node setup to create weld displacement using the image in figure 3.5. The image is used in the node labeled `weld_displacement.png`.

3.3. Generating Artificial Cracks

The approach to creating cracks used in the shader editor of Blender has been adapted from a YouTube video detailing how to create a crack in the Earth's surface [19].

The Blender shader editor generates cracks by combining various nodes to create a single crack in a plane. The overall approach to creating cracks in the shader editor is to select the desired plane where one wishes to apply the crack. Then create a material where a line with a stochastic curvature is created. Stochastic noise is then applied to the curved line to give roughness to the cracks as if something has corroded away. A slightly heightened edge may be added to the crack on the plane surface.

This noisy curved line is then height displaced in the plane, creating a crack's visual appearance. Next, the depth of the crack is coloured in a corrosive colour. This colour bleeds into the surrounding surface texture, giving an appearance of corrosion spreading. The node setup depicted in figure 3.7 shows how the cracks are created. Note that the node setup does not contain the node setup that creates, i.e. the PBR (Physically Based Rendering) texture.

The various parameters that can be modified in the node setup in figure 3.7 to create cracks of varying shapes and sizes are listed in table 3.1. The name (label) of the node is given, and its relevant modifiable parameters are listed. A specific value or an interval of values is given. The values and intervals have been found via experimentation and have been shown to produce desirable cracks.

Creating Cracks
A suggested approach is given below to recreate cracks geometrically.

1. Get familiar with basic usage of Blender, i.e. how to open and use the shader editor.
2. Create a plane mesh object and scale it to the desired size. Apply the scale to alter the plane's dimensions.
3. Apply a `Subdivision Surface` modifier to the plane, enable `Adaptive Subdivision`, and set the subdivision algorithm to `Simple`, set the `Dicing Scale` to i.e. `0.5`.
4. Create a new material in the shader editor and recreate the node setup in figure 3.7.
5. Set the parameters listed in table 3.1. Set them to a value that lies in the suggested intervals.
6. Experiment with the parameters until a desired crack aesthetic has been achieved.

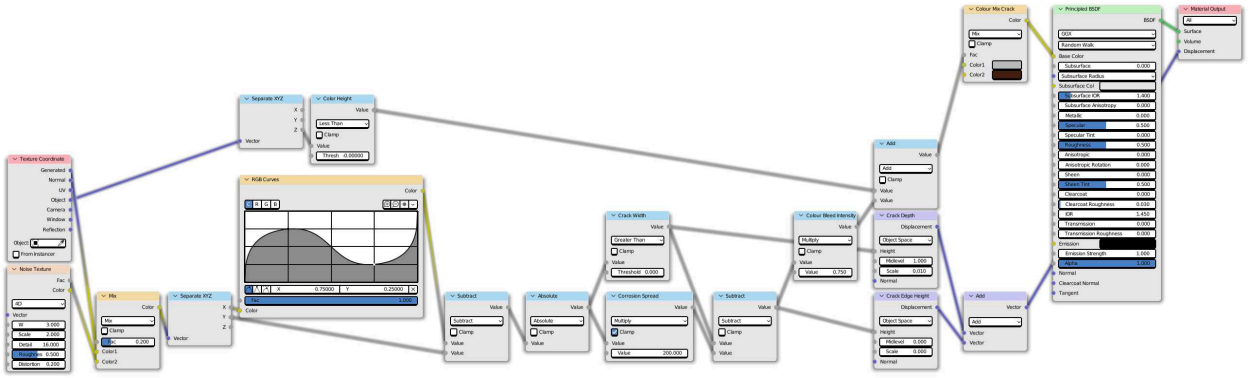


Figure 3.7.: Depicted is the Blender shader editor where the node setup for creating cracks in a plane is shown. The node setup does not include the setup of a procedural or PBR texture, only the crack geometry.

3.4. Visual Impact of Parameters on Artificial Cracks

Understanding how the parameters in table 3.1 affect the visual appearance is, in some instances, relatively trivial, i.e. the crack width and crack depth. However, how the noise texture impacts the visual appearance is not as trivial. Examples of the given parameters will be given here. These will be parameters inside the ranges in the table, giving (mostly) realistic cracks.

Node Name/Label	Parameter	Parameter Description	Interval
Crack Width	Threshold	The width of the crack. The value set for the threshold is half the crack width.	$[0.00005, 0.0075]$
Crack Depth	Scale	The depth of the crack.	$[0.005, 50]$
Crack Edge Height	Scale	How tall the crack edge shall be.	$[0.001, 0.01]$
Corrosion Spread	Value	How large of an area the corrosion shall spread from the crack.	$[25, 250]$
Corrosion Bleed Intensity	Value	How intense the corrosions colour shall bleed from the crack to the surrounding plane	$[0.01, 1.5]$
Noise Mix	Factor	The mix between the generated (actual) coordinates of the plane and the noise added to the crack	$[0.0, 0.25]$
Noise Texture	W	The texture coordinate to evaluate the noise at.	$[0.0, 100.0]$
	Scale	The base noise octave.	$[0.0, 3.0]$
	Detail	The noise detail (number of noise octaves).	$[1.0, 20]$
	Roughness	The noise roughness.	$[0.0, 0.5]$
	Distortion	The amount of distortion.	$[0.0, 0.25]$
RGB Curves	Factor	How much the mapped RGB curve impacts the curvature of the line.	$[0.0, 1.0]$
	Curve, C channel	The underlying shape of the curve. This parameter decides the overall look.	-
Colour Mix Crack	Color2	The colour of the cracks' corrosion. This colours the inside of the crack and the corrosion spreading from the crack.	-

Table 3.1.: Modifiable parameters for nodes in figure 3.7 that affects the visual crack appearance.

Node Name/Label	Parameter	Value
Noise Mix	Factor	0.25
Noise Texture	W	50
	Scale	2.5
	Detail	20
	Roughness	0.5
	Distortion	0.25
Crack Width	Threshold	0.0025
Crack Depth	Scale	1
Crack Edge Height	Scale	0.01
Corrosion Spread	Value	250
Corrosion Bleed Intensity	Value	1.2

Table 3.2.: Parameters used for node setup as in figure 3.7 for visualizations.

Impact of Noise Mix on Visual Appearance

Here we will observe the impact of varying the `Noise Mix`. All crack parameters except `Noise Mix` are as listed in table 3.2. The camera is located $2.5m$ above the plane.

Figure 3.8.: Rendered image of a crack with a `Noise Mix` of `0`.

Figure 3.9.: Rendered image of a crack with a `Noise Mix` of `0.15`.

Figure 3.10.: Rendered image of a crack with a `Noise Mix` of `0.25`.

Figure 3.11.: Rendered image of a crack with a `Noise Mix` of `0.6`. Here we observe that the noise added to the crack is too large, causing an unwanted crack representation.

Impact of Noise Texture W on Visual Appearance

Here we will observe the impact of varying the `Noise Texture W`. All crack parameters except `Noise Texture W` are as listed in table 3.2. The camera is located $2.5m$ above the plane.

Figure 3.12.: Rendered image of a crack with a `Noise Texture W` of `0`.

Figure 3.13.: Rendered image of a crack with a `Noise Texture W` of `20`.

Figure 3.14.: Rendered image of a crack with a `Noise Texture W` of `30`.

Figure 3.15.: Rendered image of a crack with a `Noise Texture W` of `40`.

Impact of Noise Texture Scale on Visual Appearance

Here we will observe the impact of varying the `Noise Texture Scale`. All crack parameters except `Noise Texture Scale` are as listed in table 3.2. The camera is located $2.5m$ above the plane.

Figure 3.16.: Rendered image of a crack with a `Noise Texture Scale` of `0`.

Figure 3.17.: Rendered image of a crack with a `Noise Texture Scale` of `1`.

Figure 3.18.: Rendered image of a crack with a `Noise Texture Scale` of `2`.

Figure 3.19.: Rendered image of a crack with a `Noise Texture Scale` of `2.75`.

Impact of Noise Texture Detail on Visual Appearance
Here we will observe the impact of varying the `Noise Texture Detail`. All crack parameters except `Noise Texture Detail` are as listed in table 3.2. The camera is located $2.5m$ above the plane.

Figure 3.20.: Rendered image of a crack with a `Noise Texture Detail` of `0`.

Figure 3.21.: Rendered image of a crack with a `Noise Texture Detail` of `5`.

Figure 3.22.: Rendered image of a crack with a `Noise Texture Detail` of `10`.

Figure 3.23.: Rendered image of a crack with a `Noise Texture Detail` of `20`.

Impact of Noise Texture Roughness on Visual Appearance
Here we will observe the impact of varying the `Noise Texture Roughness`. All crack parameters except `Noise Texture Roughness` are as listed in table 3.2. The camera is located $2.5m$ above the plane.

Figure 3.24.: Rendered image of a crack with a `Noise Texture Roughness` of `0`.

Figure 3.25.: Rendered image of a crack with a `Noise Texture Roughness` of `0.25`.

Figure 3.26.: Rendered image of a crack with a `Noise Texture Roughness` of `0.35`.

Figure 3.27.: Rendered image of a crack with a `Noise Texture Roughness` of `0.55`.

Impact of Noise Texture Distortion on Visual Appearance

Here we will observe the impact of varying the `Noise Texture Distortion`. All crack parameters except `Noise Texture Distortion` are as listed in table 3.2. The camera is located $2.5m$ above the plane.

Figure 3.28.: Rendered image of a crack with a `Noise Texture Distortion` of `0`.

Figure 3.29.: Rendered image of a crack with a `Noise Texture Distortion` of `0.1`.

Figure 3.30.: Rendered image of a crack with a `Noise Texture Distortion` of `0.2`.

Figure 3.31.: Rendered image of a crack with a `Noise Texture Distortion` of `0.3`.

Impact of Crack Width on Visual Appearance

Here we will see the impact of varying the `Crack Width`. All crack parameters except `Crack Width` are as listed in table 3.2. The `Crack Width` is not precisely as specified, as the noise added does create random perturbations to the width. Random perturbations are, however, a desired outcome. The camera is located $0.5m$ above the plane.

Figure 3.32.: Rendered image of a crack with a `Crack Width` of $\simeq 1mm$.

Figure 3.33.: Rendered image of a crack with a `Crack Width` of $\simeq 2mm$.

Figure 3.34.: Rendered image of a crack with a `Crack Width` of $\simeq 5mm$.

Figure 3.35.: Rendered image of a crack with a `Crack Width` of $\simeq 10mm$.

Impact of Crack Depth on Visual Appearance

Here we will see the impact of varying the `Crack Depth`. All crack parameters except `Crack Depth` are as listed in table 3.2. The camera is located $0.5m$ above the plane.

Figure 3.36.: Rendered image of a crack with a `Crack Depth` of $25mm$.

Figure 3.37.: Rendered image of a crack with a `Crack Depth` of $50mm$.

Figure 3.38.: Rendered image of a crack with a `Crack Depth` of $100mm$.

Figure 3.39.: Rendered image of a crack with a `Crack Depth` of $1000mm$.

Impact of Crack Edge Height on Visual Appearance

Here we will see the impact of varying the `Crack Edge Height`. All crack parameters except `Crack Edge Height` are as listed in table 3.2. The camera is located $0.15m$ above the plane.

Figure 3.40.: Rendered image of a crack with a `Crack Edge Height` of $5mm$.

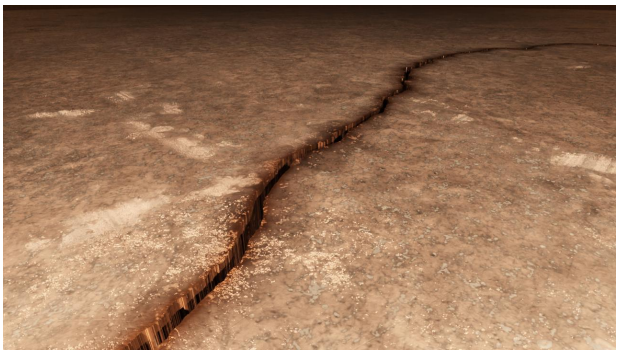


Figure 3.41.: Rendered image of a crack with a `Crack Edge Height` of $10mm$.

Figure 3.42.: Rendered image of a crack with a `Crack Edge Height` of $50mm$.

Figure 3.43.: Rendered image of a crack with a `Crack Edge Height` of $100mm$.

Impact of Corrosion Spread on Visual Appearance

Here we will see the impact of varying the `Corrosion Spread`. All crack parameters except `Corrosion Spread` are as listed in table 3.2. The camera is located $0.5m$ above the plane.

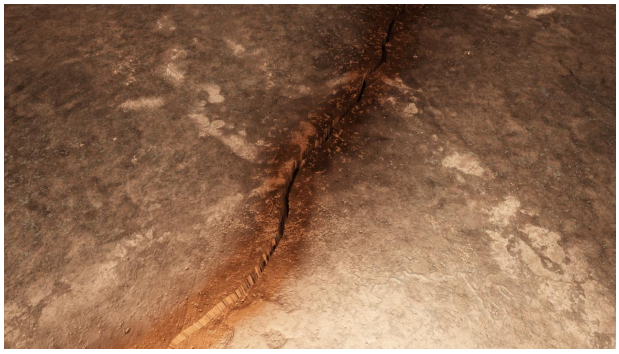


Figure 3.44.: Rendered image of a crack with a `Corrosion Spread` of `50`.

Figure 3.45.: Rendered image of a crack with a `Corrosion Spread` of `150`.

Figure 3.46.: Rendered image of a crack with a `Corrosion Spread` of `250`.

Figure 3.47.: Rendered image of a crack with a `Corrosion Spread` of `350`.

One thing to note is that the `Corrosion Spread` parameters also impact the area that the edge of the crack occupies. Compare figure 3.43 with a `Corrosion Spread` of `250`, and figure 3.48 with a `Corrosion Spread` of `750`.

Figure 3.48.: Rendered image of a crack with a `Crack Edge Height` of $100mm$, and a `Corrosion Spread` of 750.

Impact of Corrosion Bleed Intensity on Visual Appearance

Here we will see the impact of varying the `Corrosion Bleed Intensity`. All crack parameters except `Corrosion Bleed Intensity` are as listed in table 3.2. The camera is located $0.5m$ above the plane.

Figure 3.49.: Rendered image of a crack with a `Corrosion Bleed Intensity` of `0.75`.

Figure 3.50.: Rendered image of a crack with a `Corrosion Bleed Intensity` of `1`.

Figure 3.51.: Rendered image of a crack with a `Corrosion Bleed Intensity` of `1.25`.

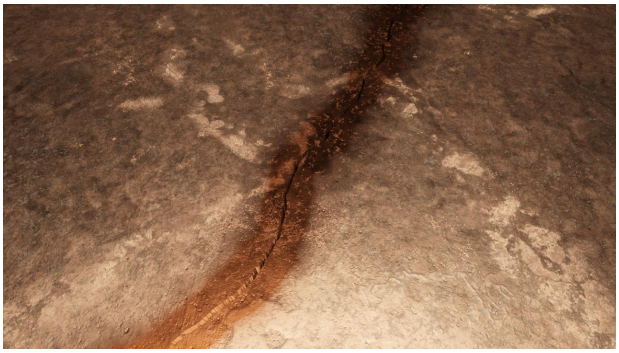


Figure 3.52.: Rendered image of a crack with a `Corrosion Bleed Intensity` of `1.5`.

3.5. Visual Impact of Subdivision Parameters

The subdivision parameters used in Cycles significantly impact the visual appearance of the generated cracks and other displaced objects in the scene. This section will give examples of how the subdivision impacts the geometry of a generated crack. For each example, a wireframe and colour images have been rendered. The wireframe is analogous to the meshes in figure 3.1.

In the following figures, the crack parameters are kept static. The static parameters used are listed in table 3.3.

Node Name/Label	**Parameter**	**Value**
Noise Texture	W	50
	Scale	2.5
	Detail	20
	Roughness	0.5
	Distortion	0.25
Noise Mix	Factor	0.25
Crack Width	Threshold	0.0025
Crack Depth	Scale	1
Corrosion Spread	Value	100
Crack Edge Height	Scale	0.01
Corrosion Bleed Intensity	Value	1.25

Table 3.3.: Parameters used for the `Subdivision` visualizations.

The adaptive subdivision parameters are accessed and set up through several parameters. `Dicing Rate` is the main setting for adaptive subdivision. The dicing rate corresponds to the size of the individual micropolygons[1] in pixels that the original geometry will be divided into. A lower value will increase render detail but will require more memory and could increase render times. The `Max Subdivisions` parameter defines the maximum subdivision level for the adaptive algorithm. Increasing this value will allow more detail in the mesh at the cost of rendering time and memory. Both these properties are found in the `Render Properties`. `Dicing Scale` can be found in `Subdivision` under the `Modifier Properties` of a specific object. It modifies the dicing scale for the specific object the modifier is applied to.

[1] A polygon roughly the size of a pixel or smaller.

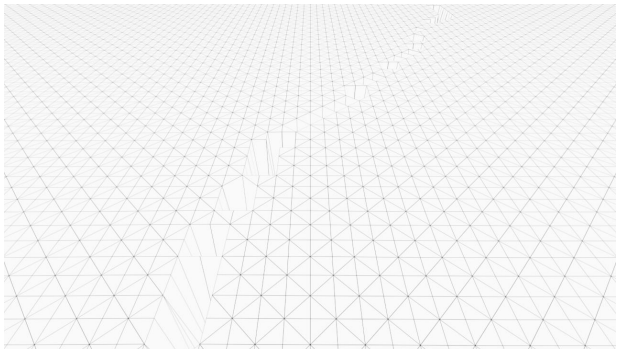


Figure 3.53.: Rendered wireframe of the subdivided crack plane. Here we can see the subdivided mesh with its edges and vertices.

Figure 3.54.: Rendered colour image. Parameters: `Dicing Rate: 1.00 px`, `Max Subdivisions: 1`, `Dicing Scale: 20.00`. Here we can observe the low degree of subdivision, making the cracks' geometry look unrealistic.

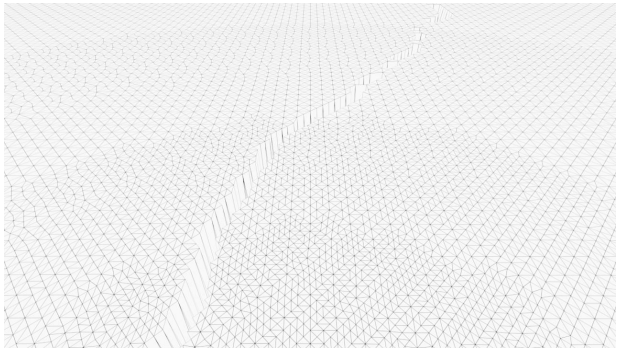


Figure 3.55.: Rendered wireframe of the subdivided crack plane. Here we can see the subdivided mesh with its edges and vertices. What is of interest to observe is that the adaptive subdivision has subdivided the mesh closer to the camera to a higher degree. The mesh farther away from the camera has a lower degree of subdivision. However, the size of the polygons appears to be roughly the same due to the distance.

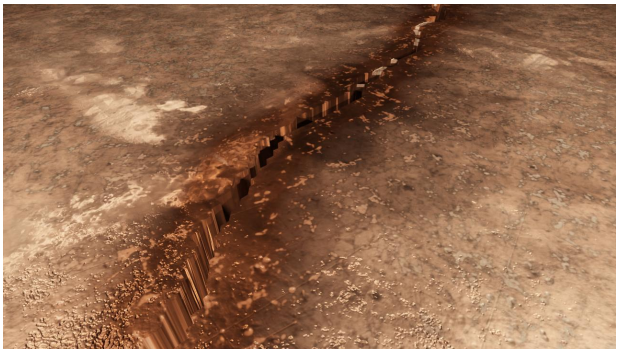


Figure 3.56.: Rendered colour image. Parameters: Parameters: `Dicing Rate: 1.00 px`, `Max Subdivisions: 12`, `Dicing Scale: 20.00`. The appearance of the crack has improved from figure 3.54, but it still appears unrealistic.

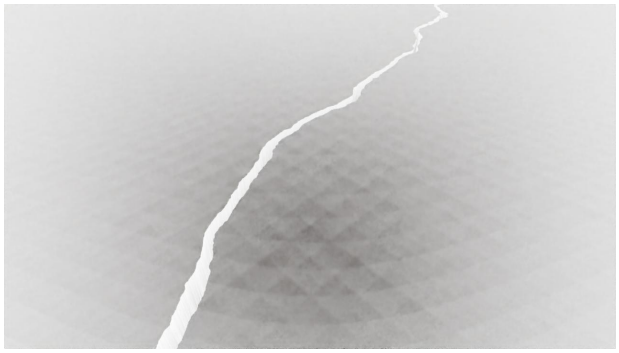


Figure 3.57.: Rendered wireframe of the subdivided crack plane. Here we can see the subdivided mesh with its edges and vertices. It is impossible to see the individual subdivided polygons as the final size of a polygon has become approximately half a pixel.

Figure 3.58.: Rendered colour image. Parameters: `Dicing Rate: 1.00 px`, `Max Subdivisions: 12`, `Dicing Scale: 0.5`. The crack now appears much more realistic than in figure 3.56.

4. Appearance and Texture Generation in the Scene

The goal of photorealistic rendering is to generate images from a 3D scene that are virtually indistinguishable from a real photo captured from the same perspective [15, p.4]. At its core, photorealistic rendering is based on the physics of light and its interaction with matter. When light hits an object, some is absorbed, some is reflected, and some is transmitted through the material. How light is reflected and absorbed by different surfaces can vary widely, depending on factors such as the surface material that describes its reflectivity, roughness, and transparency.

4.1. Background of Photorealistic Rendering

Texturing in Computer Graphics
Textures are the visual attributes applied to the surfaces of 3D models, responsible for imparting intricate details that enhance realism. Without textures, a 3D object would be a monochromatic, flat, and lifeless shape. The rich information of real-world surfaces as the scratches on a metal surface, the grain of a wooden object, and the minute patterns on a fabric would be more challenging to portray geometrically, especially given the computational constraints. Furthermore, textures offer a strategic approach to operational efficiency. Attempting to model minute details on every object in a scene geometrically is laborious and could lead to a steep demand for computational resources. The essence of textures lies in their ability to represent complex surface information while maintaining a lean geometry count.

Texture mapping applies two-dimensional images to three-dimensional models, enhancing computer graphics' perceived detail and realism. In a simplistic model, each vertex in a 3D object is assigned a corresponding point in a 2D texture image. These corresponding points are defined in texture space, usually referred to as UV coordinates, ranging from (0,0) to (1,1), with (u, v) analogous to (x, y) in Cartesian coordinates. After assigning UV coordinates to the vertices, the next step is interpolating these coordinates for the interior points of the polygons, a process conducted during rasterisation. A standard method for this is Barycentric interpolation, which determines the value of an attribute within a triangle given its values at the vertices. Once the UV coordinates are interpolated, these are used to sample the texture image. However, since the coordinates usually do not correspond to an exact pixel in the texture image, some form of texture filtering, such as bilinear or trilinear filtering, is applied to determine the resulting colour [15, p.597].

Materials in Computer Graphics
Materials govern how these textured surfaces respond to light, defining their appearance under different lighting conditions. While textures provide surface details, materials dictate how those surfaces reflect or absorb light, simulating the physical properties of real-world materials. The importance of materials becomes more evident when considering the vast range of material properties in the real world. From the rough, matte surface of a concrete wall to the smooth, reflective finish of polished metal, materials are how i.e. reflections are digitally synthesised [15, p.571].

In essence, a texture can be considered the skin applied onto a 3D model, while the material is the

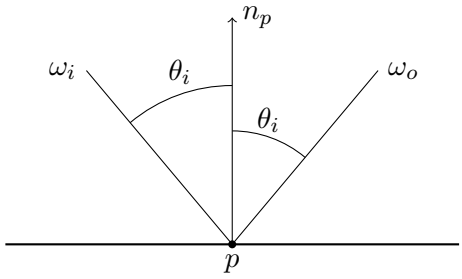


Figure 4.1.: Illustration of a perfect specular reflection. The incident radiance ω_i is perfectly specularly reflected at point p and becomes the outgoing radiance ω_o. Note that the angle θ_i of the incident and outgoing radiance is identical in relation to the surface normal n_p.

set of rules that define how that skin interacts with light. Textures are combined with materials when rendering an object in a 3D application.

The fundamental principle behind materials is the *rendering equation* [20]:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i d\omega_i. \tag{4.1}$$

Where $L_o(p, \omega_o)$ is the total outgoing radiance from a point p in direction ω_o. $L_e(p, \omega_o)$ is the emitted radiance from the surface at point p in direction ω_o. The integral is the incident radiance from all directions on the sphere S^2 around point p, determining how much of the light is scattered. It is scaled by the *bidirectional scattering distribution function* (BSDF) $f(p, \omega_o, \omega_i)$. $L_i(p, \omega_i)$ is the incident radiance. The BSDF describes how light scatters on a surface in different directions and how light is transmitted and scattered through an object. The cosine term ensures that the incident radiance is weighted by the angle between the incident direction and the surface normal so that surfaces reflect more light in the direction perpendicular to their surface normal. Analytically solving the integral in the rendering equation is generally not possible. Therefore simplifying assumptions or numerical techniques are used [15, p.863]. An illustration of a perfectly specular reflection can be seen in figure 4.1.

The BSDF often encompass a *bidirectional reflectance distribution function* (BRDF) that describes scattering at the surface. And a *bidirectional transmittance distribution function* (BTDF) that describes the distribution of transmitted light. There are functions such as *bidirectional scattering-surface reflectance distribution function* (BSSRDF) that describe how light enters a surface, is sub-scattered and exits the surface at a different point than the origin.

Physically Based Rendering
Physically Based Rendering (PBR) is an approach to material and light modelling that aims to simulate the physical interaction between light and materials more accurately. In PBR, textures are used to define surface colour and material properties. These include the base colour (albedo) texture, representing the surface's inherent colour. In addition, the metallic texture determines which parts of the surface are metallic and non-metallic, and the roughness texture indicates how rough or smooth the surface is. Combined with a BRDF model, these maps allow for highly realistic rendering of various materials under various lighting conditions.

Computer graphics software must simulate the behaviour of light in a virtual environment to create photorealistic images. Simulating the behaviour of light has typically been performed using ray tracing. Ray tracing is a technique that follows the path of light rays from the virtual camera to the objects in the scene and then calculates the colour of each pixel by tracing the path of the rays as

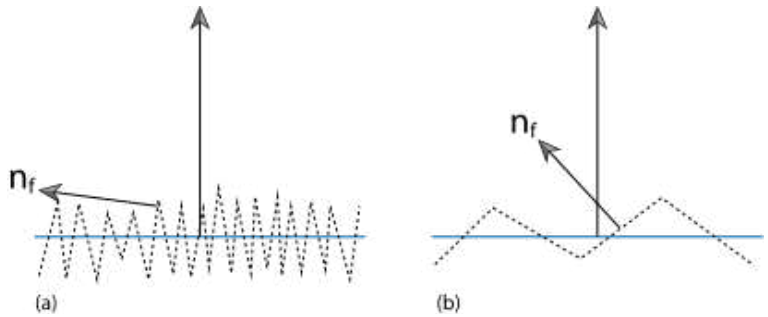


Figure 4.2.: (a) Rough microfacet, (b) smoother microfacet. Figure from Pharr et al. [15, p.533].

they bounce off objects in the scene and interact with different materials. It is a reverse process of how light would behave in real life, where it starts from the light source and bounces off the surfaces to reach the camera.

A further development of ray tracing is path tracing [20]. Path tracing is a ray tracing variant based on the Monte Carlo method. It simulates the behaviour of light in the scene by tracing many random paths from the virtual camera to the objects in the scene. Path tracing samples multiple paths per pixel and averages them to estimate the final colour. As the number of samples increases, the image becomes progressively more accurate.

Central to PBR is the use of physically-accurate BRDFs. Examples are, i.e. the Lambertian reflection, where light scatters equally in all directions when hitting the surface, to more advanced models, such as the Torrance–Sparrow model [21, 22]. Torrance-Sparrow considers the surface's microstructure by modelling it as a collection of small facets or microfacets. These microfacets have different orientations and surface normals, affecting reflected light's direction. See figure 4.2. The model also includes parameters that describe the surface roughness and the distribution of microfacets. The general form of the Torrance-Sparrow model is:

$$f_r(\omega_o, \omega_i) = \frac{D(\theta_h)G(\theta_o, \theta_i)F(\theta_d)}{4\cos\theta_o \cos\theta_i} \tag{4.2}$$

- where $D(\theta_h)$ is the microfacet distribution function, which describes the probability density of microfacets oriented in direction ω_h. See figure 4.3.
- $G(\theta_o, \theta_i)$ is the geometric attenuation factor, which accounts for the fact that other microfacets occlude some microfacets and cannot contribute to the reflected light.
- $F(\theta_d)$ is the Fresnel reflectance at the interface between air and the surface, which depends on the angle θ_d between ω_i and the micro normal ω_h and the refractive index of the surface.

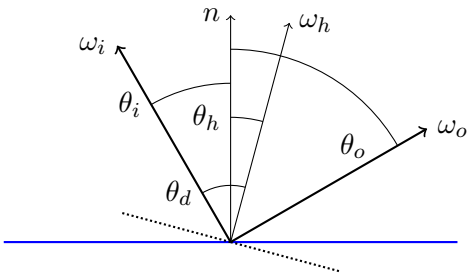


Figure 4.3.: The microfacets are specular, causing only specular reflection of the ω_i to ω_o where the surface normal n is equal to the half-angle vector ω_h. Illustration of incident light ω_i, scattered light ω_o, the half angle vector ω_h, and the surface normal n.

Physically Based Rendering Materials

Physically Based Rendering (PBR) materials refer to the digital representation of materials, which closely mimic their real-world counterparts by following the physical laws of light reflection and energy conservation. PBR materials aim to accurately simulate how light interacts with surfaces in the real world, resulting in more realistic and consistent renders. They use mathematical models to describe how light interacts with a surface based on the material's physical properties. PBR materials involve several key texture maps to define the material properties. Maps of particular interest are described:

The *albedo* (colour) map is a texture that represents the base colour of an object's surface, free from any shadows or lighting which might otherwise affect the perception of the material's colour. It is intended to capture only the intrinsic property of the material, the colour that the material would appear under neutral lighting. It can be thought of as the raw colour of the object. The albedo map should not contain any high-frequency details like ambient occlusion or shadows, as other maps will handle these details.

The *metallic* map defines which parts of the model are metallic and non-metallic. This map is typically grayscale, with white representing a fully metallic surface and black representing a non-metallic surface. It is grayscale because in a PBR workflow, materials are typically categorised into either metals or dielectrics (non-metals), and there's typically no intermediate or partial metal value.

A *normal* map is a texture type that represents the normal vector of the surface of a 3D model, used for faking the lighting of bumps and dents. Normal maps are employed to enhance details without the need for additional polygons. Their unique purple hue arises because each pixel's colour values are interpreted as the X, Y, and Z coordinates of the surface normal at that specific location.

A *displacement* map is a grayscale texture that alters the actual geometry of a surface based on the grayscale values in the map. Lighter values represent areas where the surface is raised, and darker values represent areas where the surface is lowered. When a displacement map is applied, the geometry of the surface changes and adds more detail to the model's mesh. However, this increased realism comes at the cost of significantly increased computation, as more geometry means more complexity in the rendering calculations.

The *roughness* map is a grayscale texture that defines how rough or smooth the model's surface is. A value of white is fully rugged, scattering reflected light in many directions and giving a diffused look. A value of black is entirely smooth, reflecting light in a single direction and giving a shiny look.

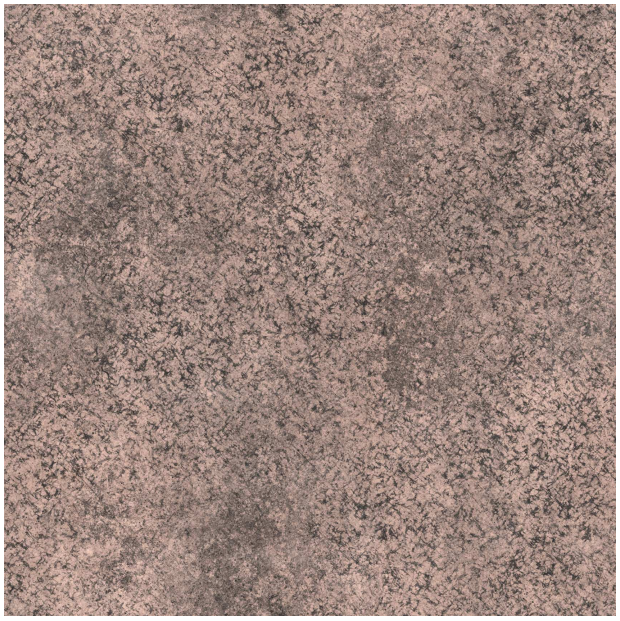


Figure 4.4.: Color map for the PBR material shown in figure 4.10.

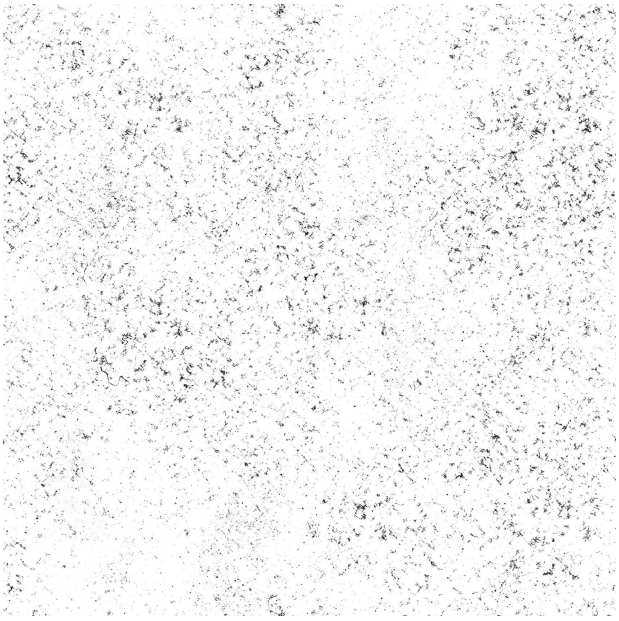


Figure 4.5.: Metallic map for the PBR material shown in figure 4.10.

Figure 4.6.: Normal map for the PBR material shown in figure 4.10.

Figure 4.7.: Zoomed in normal map from figure 4.6 for the PBR material shown in figure 4.10. Here the variation in colours that are used to represent X, Y, and Z coordinates of the surface normal at that point is observed.

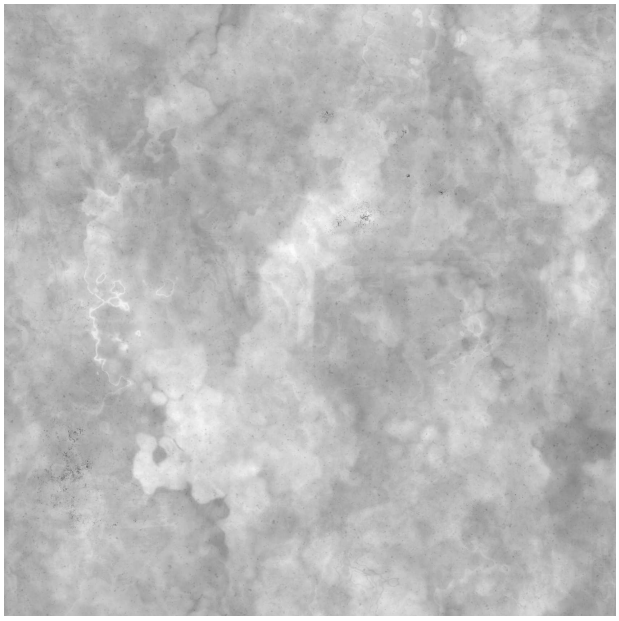


Figure 4.8.: Displacement map for the PBR material shown in figure 4.10.

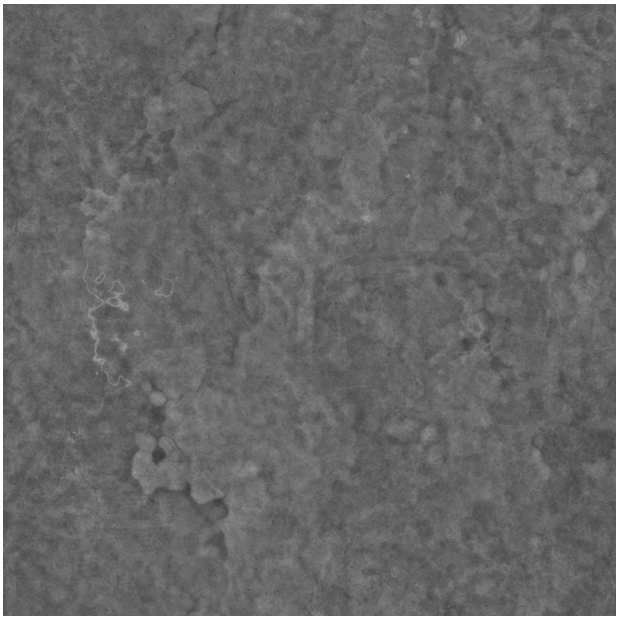


Figure 4.9.: Roughness map for the PBR texture and material shown in figure 4.10.

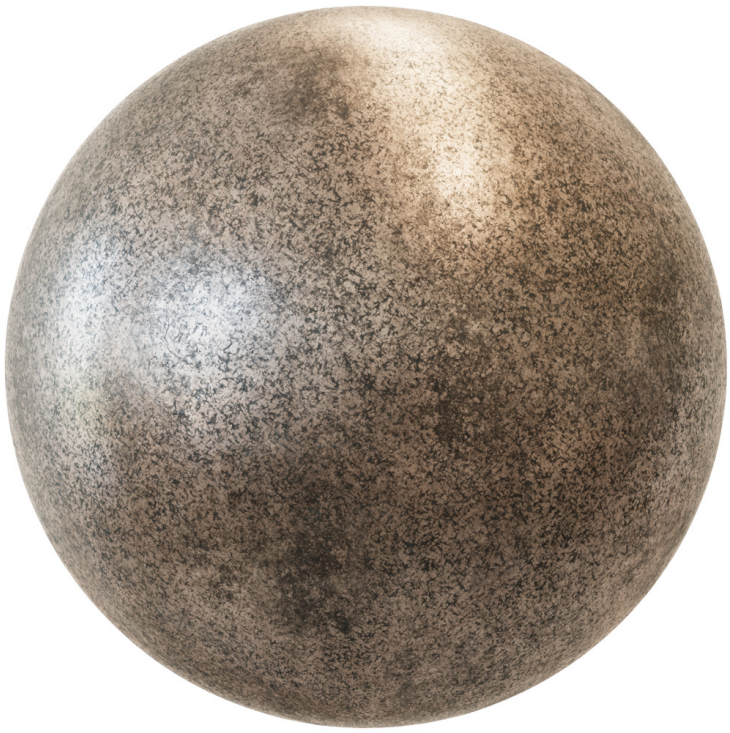


Figure 4.10.: The combined maps from figures 4.4, 4.5, 4.6, 4.8, 4.9 applied to a sphere.

Combining these maps provides a more realistic portrayal of a surface's appearance under varying lighting conditions. The albedo map gives the base colour, the metallic map differentiates between metal and non-metal parts, the roughness map determines how shiny or diffused the reflection will be, and the normal map provides the finer surface details. The displacement map alters the actual geometry of the surface. Together, these make up the essential components of a PBR material.

Photorealistic Rendering in Blender
Photorealistic rendering in Blender can be achieved by selecting specific tools that Blender offer. Among the tools that Blender offers are the Cycles render engine and the Principled BSDF shader. The Cycles render engine is physically-based and designed to produce high-quality, photorealistic images. Cycles simulate light in a physically accurate manner, considering the intricate details of how light rays bounce around a scene, how materials absorb and reflect light, and how light interacts with the atmosphere. Cycles also support global illumination, which considers direct and indirect light that bounces off surfaces. This inclusion of indirect light adds depth and realism to a scene, leading to more believable renders [23].

Complementing Cycles is the Principled BSDF shader based on the Disney Principled BRDF. The Disney Principled BRDF Burley [24] is, in turn, based on the Torrance-Sparrow model, which was described in equation 4.2, albeit with modifications. The Principled BSDF shader is a comprehensive all-in-one shader introduced in Blender to provide a standardised, physics-based shader across various renderers. It is a consistent and artist-friendly tool for defining a broad range of physically-based materials. Like Cycles, the Principled BSDF shader is based on physical models of how light interacts with surfaces. The shader is versatile, representing different materials—from dielectric materials like plastic and glass to metallic surfaces. Despite its versatility, the Principled BSDF shader is designed to simplify creating complex shaders. Instead of demanding a deep understanding of the physics of light and materials, it presents intuitive controls that artists can use to achieve the desired appearance.

4.2. Using Physically Based Rendering Materials for Photorealistic Ship Tank Surfaces

Three approaches were taken for creating photorealistic textures and materials for ship surfaces. The first was utilising PBR materials that were gathered online. The second was to implement a procedural stained material in the shader editor of Blender. And the third was to implement a procedural painted metal surface with corrosive spots, also in the shader editor.

Examples of this material can be seen in section 8.1. The first approach used to create photorealistic ship tank surfaces was using PBR materials. These are described in section 4.1. All PBR materials were gathered from ambientCG [25], licensed under the Creative Commons CC0 1.0 Universal License. The collected materials were in a pixel resolution of 4096×4096, and the included files used were colour (albedo), metalness, roughness, normal, and displacement. Note that `NormalGL` is used, not `NormalGX`. This is because Blender uses the OpenGL format and not the DirectX format.

The approach used to apply a gathered PBR material in Blender was to select the desired object to which a material will be applied. Then, create a new material in the `Shader Editor`. Afterwards, recreate the node setup shown in figure 4.11.

Adjusting the size of the PBR material is performed by altering the `X` and `Y` under `Scale` in the `Mapping` node. The drawback to this mapping approach is that textures quickly display repetitive patterns. In figure 4.12, a plane approximately 10 metres wide with material using the texture in figure 4.10 is applied to it. When scaling it down ten times, a repetitive pattern arises. This is observed in figure 4.13 in which the identical texture is applied to the plane, only scaled down ten times.

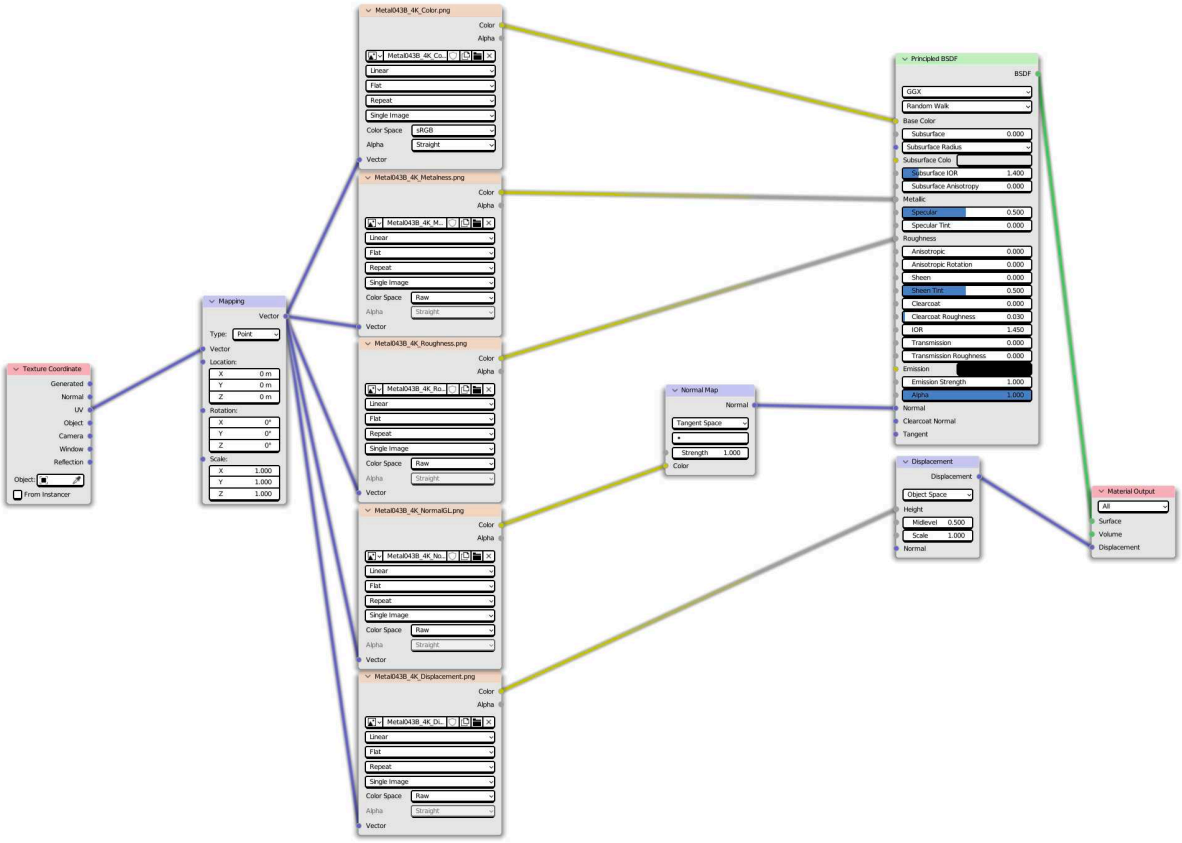


Figure 4.11.: Node setup using PBR materials in the Blender `Shader Editor`.

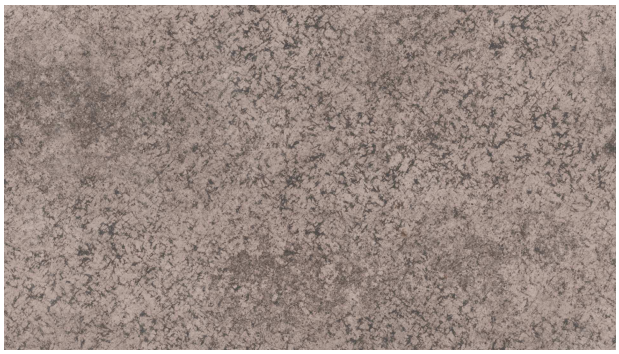


Figure 4.12.: PBR texture from figure 4.10 applied to a plane 10 metres wide, with a `X` and `Y` `Scale` of `1`.

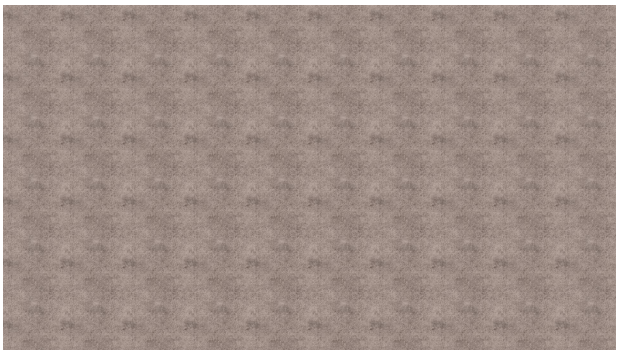


Figure 4.13.: PBR texture from figure 4.10 applied to a plane 10 metres wide, with a `X` and `Y` `Scale` of `10`. Observe the repetitive pattern that arise.

To alleviate the repetitive pattern, we can use texture bombing. In texture bombing, textures are blended to create a more complex surface appearance [26]. For example, it's often used in video game terrain to mix different textures like grass, dirt, and rock more realistically. In Blender, this can be achieved by an extensive node setup in the `Shader Editor` but can be efficiently utilised through a Blender-addon such as `Scattershot` [27]. The node setup using Scattershot is observed in figure 4.14. Inside the `Scatter Fast` node is an extensive node setup to facilitate texture bombing. The results of texture bombing are observed in figure 4.16, where no repetitive pattern is seen.

Instead of a square pattern occupied by the texture, the pattern is mixed and "bombed" to create an illusion of a procedural texture. This can be observed in figure 4.15, where each cell has been visualised by giving it a random colour. All while the texture has been scaled down.

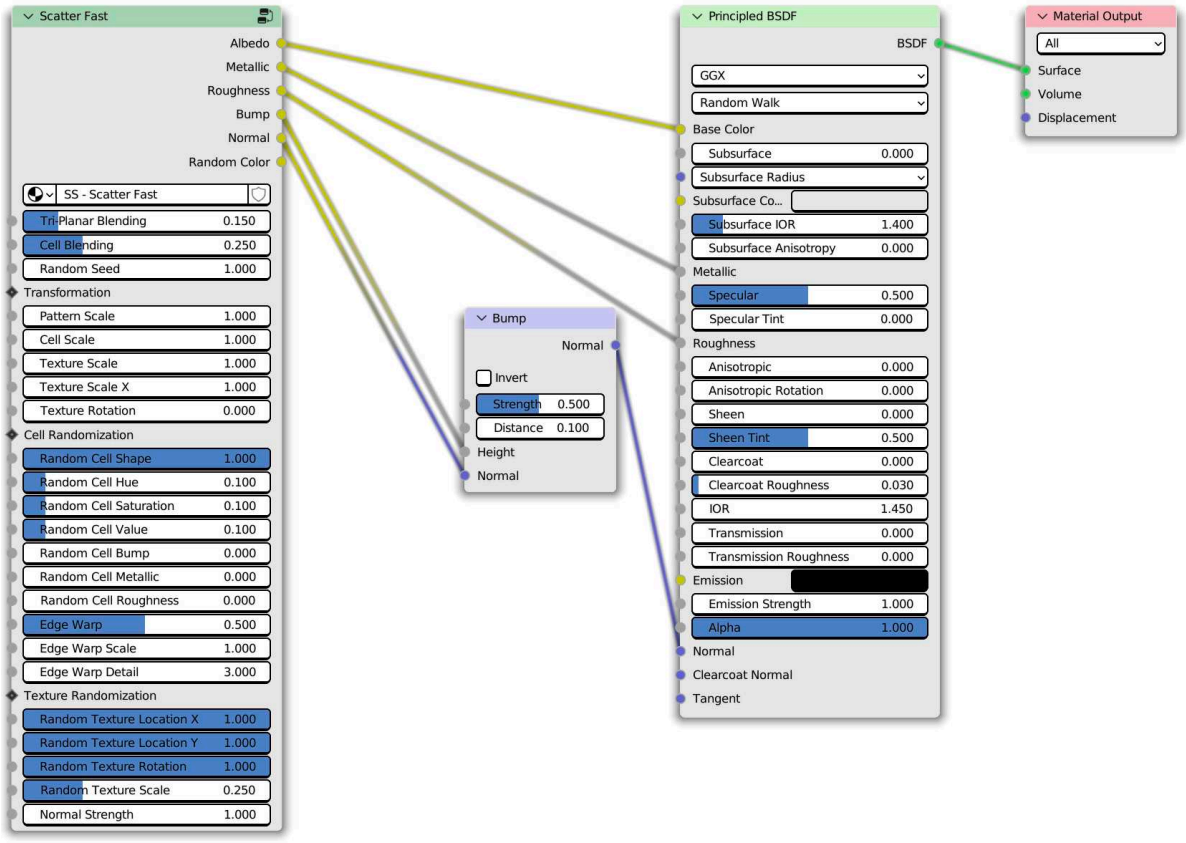


Figure 4.14.: `Scatter Fast` node setup in the Blender `Shader Editor`.

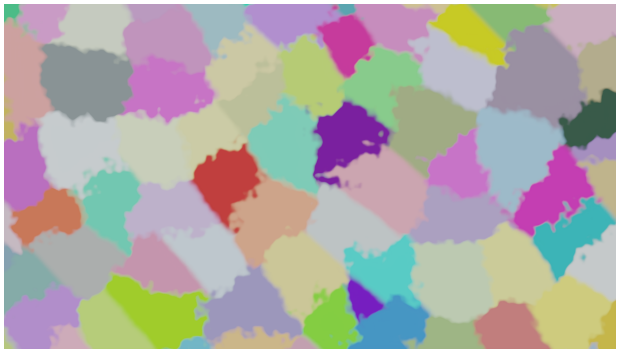


Figure 4.15.: The same plane as in figure 4.12 and 4.13, where the node setup in figure 4.11 has been replaced with the `Scatter Fast` node setup as in figure 4.14. The `Random Color` from the `Scatter Fast` node has been visualised in this figure.

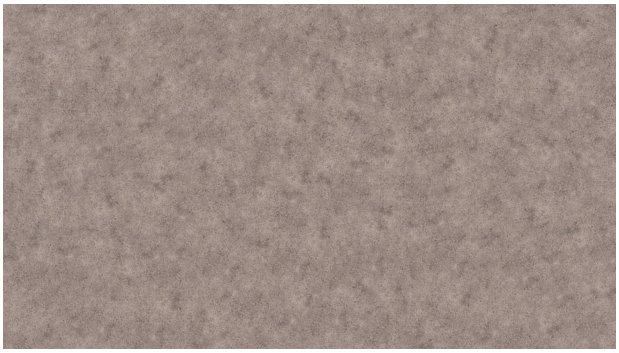


Figure 4.16.: The same plane as in figure 4.12 and 4.13, where the node setup in figure 4.11 has been replaced with the `Scatter Fast` node setup as in figure 4.14.

4.3. Generating Procedural Stained Metal Material for Photorealistic Ship Tank Surfaces

Examples of this material can be seen in section 8.2. The second approach was to create a procedural stained metal material wanting to make more variation and see whether this was feasible. A method was found on YouTube [28]. This material was implemented in the shader editor. Figure 4.17 shows the shader node setup. A noise texture and musgrave texture are used to create the stained look of the material. Musgrave textures produce a wide range of natural-looking patterns, making them useful for creating complex surfaces like terrain, marble, or other organic materials. Colour ramps are then used to control the texture and look inserted into the Principled BSDF node. Except for the colour ramps, most parameters are observed in figure 4.17. As set in the node setup, colour ramps parameters are given in table 4.1. When rendering images with this material, the `Stops` in `ColorRamp2` was set to two random colours taken from the colour chart in figure 4.19.

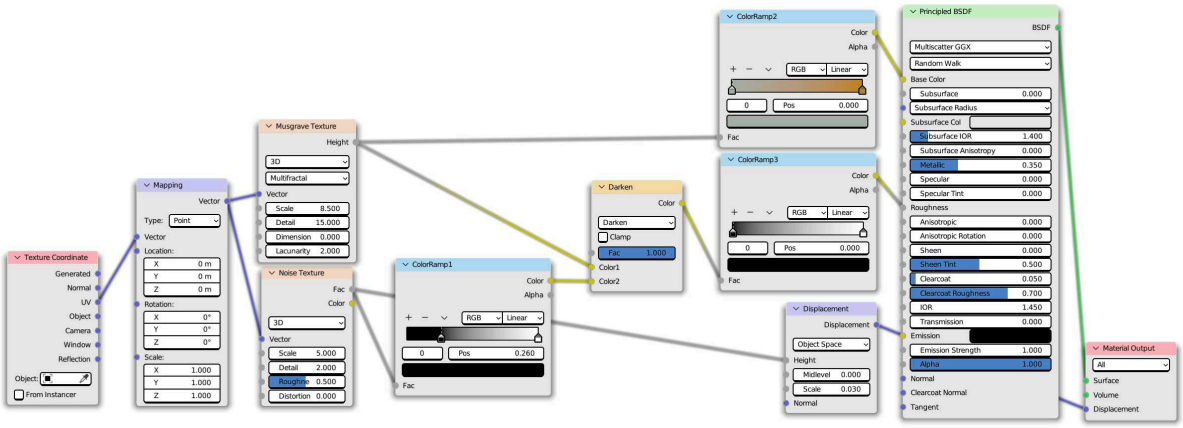


Figure 4.17.: Node setup for a procedural stained metal in the Blender `Shader Editor`.

Node Name/Label	Stop	Position	Hex Value
ColorRamp1	0	0.260	000000
	1	1.000	FFFFFF
ColorRamp2	0	0.000	A8AEA9
	1	1.000	B8862B
ColorRamp3	0	0.000	000000
	1	1.000	FFFFFF

Table 4.1.: `Stop`, `Position`, and color values for `ColorRamp` nodes in figure 4.17.

4.4. Generating Procedural Painted Metal Material for Photorealistic Ship Tank Surfaces

Examples of this material can be seen in section 8.3. The third approach was similar to the second. It was to create a procedural painted metal material. A method was found on YouTube [29]. This material was implemented in the shader editor. Figure 4.18 shows the shader node setup. Noise textures are fed into colour ramps. Colour ramps are then used to control the texture and look inserted into the Principled BSDF node. Most parameters are observed in figure 4.18, but the colour ramps are not. Their parameters, as set in the node setup, are given in table 4.2. When

rendering images with this material, `stop` 0 and 1 in `ColorRamp2` were set to the same random colour taken from the colour chart in figure 4.19.

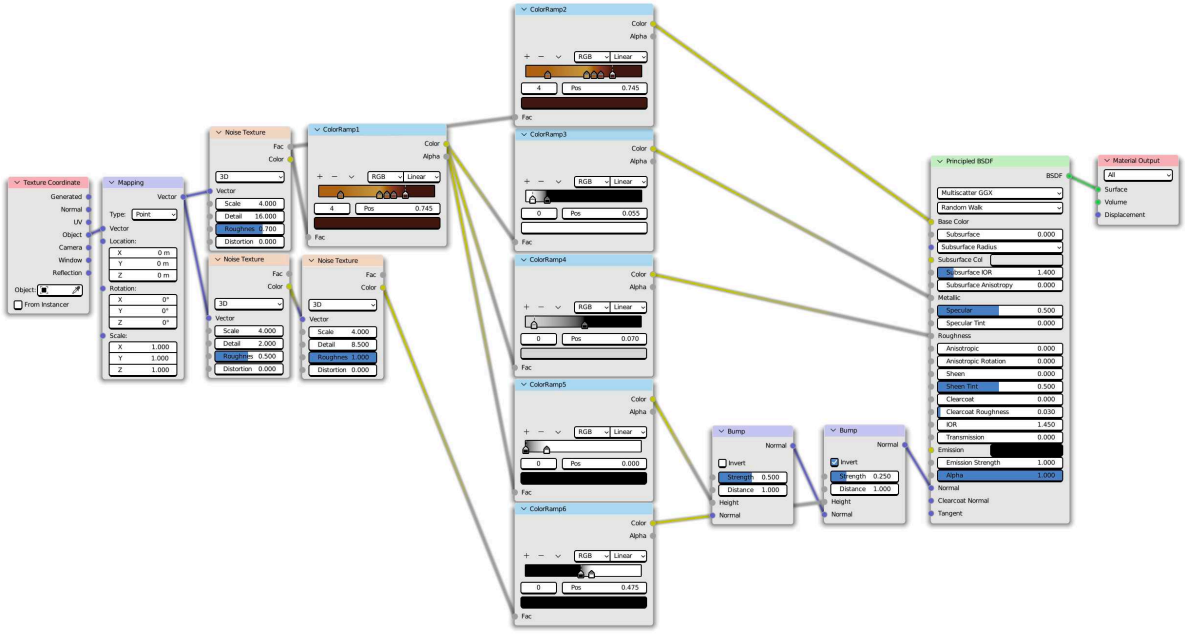


Figure 4.18.: Node setup for a procedural painted metal in the Blender `Shader Editor`.

Node Name/Label	Stop	Position	Hex Value
ColorRamp1	0	0.185	A6692B
	1	0.525	C19B4D
	2	0.585	A25F22
	3	0.645	843D2B
	4	0.745	3E1C14
ColorRamp2	0	0.185	A6692B
	1	0.525	C19B4D
	2	0.585	A25F22
	3	0.645	843D2B
	4	0.745	3E1C14
ColorRamp3	0	0.055	FFFFFF
	1	0.180	000000
ColorRamp4	0	0.070	D7D7D7
	1	0.000	000000
ColorRamp5	0	0.000	000000
	1	0.180	FFFFFF
ColorRamp6	0	0.475	000000
	1	0.570	FFFFFF

Table 4.2.: `Stop`, `Position`, and color values for `ColorRamp` nodes in figure 4.18.

Figure 4.19.: RGB triplets gathered from DNV crack images using K-Means clustering. These RGB triplets were captured to have representative colours found in ship tanks during surveys. These are used to colour the PBR and procedural textures.

4.5. Generation and Rendering of the Segmentation Mask

The segmentation masks for training and serving as the ground truth are created by rendering the scene with new materials applied to objects. When a colour image is rendered, a segmentation mask is rendered alongside it. What occurs is that the desired colour image is rendered, and then the same scene is rendered again with a crucial modification.

After the colour image has been rendered and before the segmentation mask is rendered, all objects are stripped of their materials and replaced with a new material suitable to create a segmentation mask. In the default implementation, the new material created is seen in figure 4.20. This material is a single emission shader and is the material used for all objects except the crack plane. Desired in the segmentation masks are objects with distinct, separable colours as depicted in figure 4.21. This results in equation 4.1 being reduced to:

$$L_o(p, \omega_o) = L_e(p, \omega_o) \tag{4.3}$$

where $L_e(p, \omega_o)$ is the emitted radiance from the surface at point p in direction ω_o.

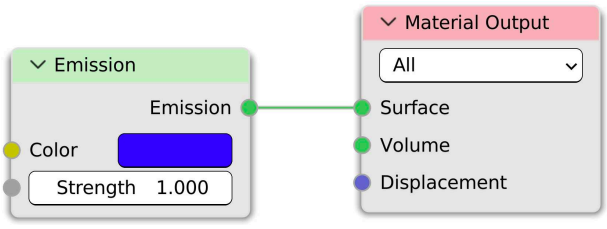


Figure 4.20.: Default material created for an object with the intent of rendering a segmentation mask. In this instance, the object will be coloured blue in the segmentation mask.

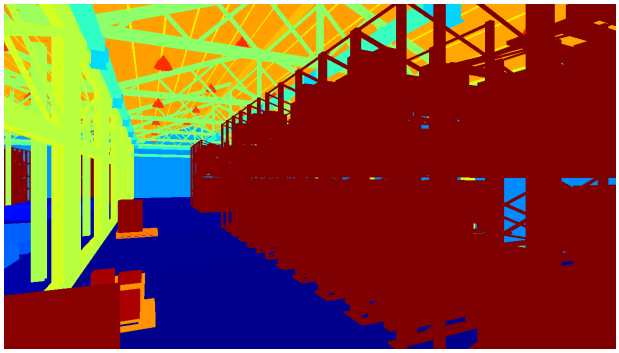


Figure 4.21.: Rendered instance segmentation mask of colour image in figure 4.22.

Figure 4.22.: Rendered colour image, with its rendered instance segmentation in figure 4.21.

BlenderProc provides segmentation masks of a scene with a single function call in a Python script. Unfortunately, it is unsuitable for application on the crack plane in its default form. In the default implementation, the segmentation mask is rendered for the entirety of the object and its mesh. This implies that if a mesh volume is displaced, it is coloured as part of the object. So in the case of the crack plane, it means that the entirety of the plane is rendered as a single colour, not creating a separate colour for the crack.

The solution to this is that the original material is partially replaced. The crack geometry, as i.e. in figure 3.7, is kept as it were in the colour image, so what is being replaced are the surface textures. The final material can be seen in figure 4.23, where the node group `Segmentation Shader` is added during modification, replacing the procedurally generated or PBR nodes. The node group `Crack Geometry` is kept from the existing material.

The colours set for objects in the segmentation mask, i.e. the blue colour in the `Emission` shader in figure 4.20, are created by a function. This function generates many equidistant RGB triplets and returns as many distinct colours as objects in the render. When the scene is loaded in the script, each object can be assigned a `category_id` that decides which object will share segmentation colours. I.e. if car objects are loaded into the scene in the script, they can be assigned the same `category_id`, and all share, i.e. the colour green in the segmentation mask. Examples of the crack segmentation mask can be seen in chapter 8.

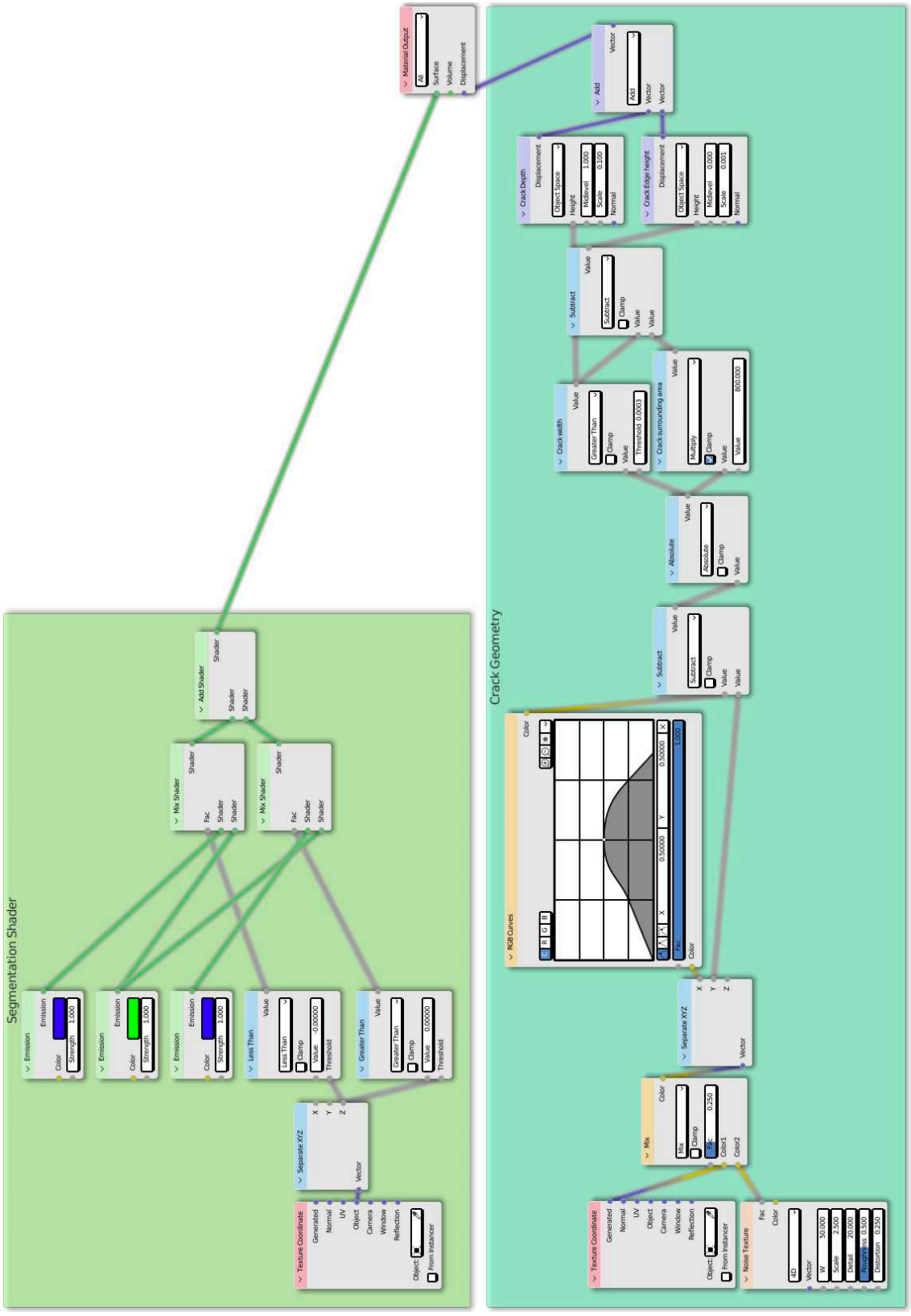


Figure 4.23.: material created for the crack plane that is created for a segmentation mask. In this instance, the non-crack parts of the object will be coloured green, and the crack coloured blue in the segmentation mask.

5. Illumination of the Scene and Virtual Camera Setup

5.1. Background of Illumination in Computer Graphics

Illumination in computer graphics is deeply rooted in radiometry and photometry, branches of physics that measure electromagnetic radiation. These disciplines describe how light behaves and interacts with the world and objects within a rendered scene when using a physically based renderer such as Cycles, contributing to depth, colour, and texture perception.

Radiometry focuses on the measurement of all electromagnetic radiation, including visible light. Within computer graphics, radiometry aids in computing how light interacts with surfaces in a scene, enabling realistic rendering of visuals. Two relevant radiometric units are radiant energy and radiant flux. Radiant energy is the total amount of energy emitted, transferred, or received as electromagnetic radiation and is measured in joules (Q). Radiant flux defines the total power of electromagnetic radiation, capturing the rate at which radiant energy changes. The unit for radiant flux is the watt (W), emphasising its focus on energy per unit of time.

While radiometry considers all electromagnetic radiation, photometry narrows the scope down to studying visible light as the human eye perceives it. This makes photometry important for realistic rendering in computer graphics, as it mimics the human eye's variable sensitivity to different wavelengths of light. Photometric quantities parallel their radiometric counterparts, but they weigh the power of light according to its effect on the human eye. Hence, we have luminous energy, measured in lumen seconds (also called talbot (T)), representing the perceived energy of light. Luminous flux, in lumens (lm), analogously defines the perceived power of light or the rate of change of luminous energy per unit of time.

Both radiometry and photometry have distinct roles in computer graphics. Radiometry calculates how light propagates and interacts within a scene, informing how light sources affect surfaces. While photometry brings in the human element, shaping how an observer perceives the illuminated scene, making it vital for achieving high realism in computer-generated imagery [15, p.334-342].

5.2. Simulating the Illumination of the REDHUS Drone in Blender

Two obstacles had to be resolved in order to recreate the illumination setup of the REDHUS drone in Blender. The first was creating the illumination objects to simulate the actual LEDs as accurately as possible, and the second was setting the location and orientation of illumination objects in the scene. Six LEDs are mounted on the front of the drone to illuminate its work environment. The LEDs are exhibited in figure 1.1, where three LEDs are mounted to each red bracket. A side view of a bracket is illustrated in figure 5.1.

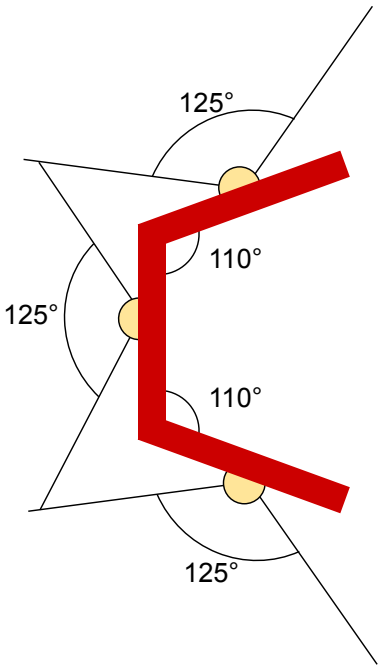


Figure 5.1.: A side view of a REDHUS LED bracket. The drone has two of these on each side of the camera.

Simulating the REDHUS LEDs

Simulating the LEDs on the drone as physically accurately as possible was desired, as the intent was to achieve a realistic simulation of the REDHUS drone. The LEDs used on the drone are from the manufacturer `Cree LED` and have the part number `XHP70B-00-0000-0D0BP240E`. Relevant characteristics of the LED are listed in table 5.1.

Characteristic	Value
Physical Size	$7.0mm \times 7.0mm$
Viewing Angle	125°
Colour Temperature	4000 Kelvin (K)
Luminous flux	1830 Lumen (lm)

Table 5.1.: Relevant characteristics of the `Cree LED XHP70B-00-0000-0D0BP240E`.

In Blender, there are four general categories of light objects:

- *Point light* mimics a lightbulb or any light source that emits light omnidirectionally from a single point. Light intensity diminishes with distance, following the inverse square law in real-world physics.
- *Sun light*, unlike a point light, casts parallel rays across the entire scene. It mimics the behaviour of sunlight, as the sun is so far away that its rays reach us almost parallel.
- *Spot lights* emit light in a specific direction, shaped like a cone. Resembling how many real-world lights like stage spotlights or desk lamps work.
- *Area light* emits light from a surface (a rectangle or a disk), unlike the other types emitting from a point. The surface area affects the softness of shadows, with larger surfaces resulting in softer shadows.

The spotlight was chosen to simulate the LEDs because one can control the direction of light and the shape of the light cone. The light emitted by an LED is usually brighter and more concentrated than other light sources, and these are factors you can adjust in Blender. The LED's colour can also be simulated by adjusting the light source's colour in Blender.

Setting the physical size and viewing angle in Blender was straightforward and was performed by two object variable assignments. A list of RGB triplets that represents their blackbody colour temperature was found from Charity [30]. Since Cycles operates using linear RGB values, the RGB triplet `1.0000, 0.6636, 0.3583` representing a colour temperature of 4000K could be set directly in the Python script.

Converting the luminous flux to radiance took more work. Light objects in Cycles use radiant flux. Therefore, the luminous flux of the LED had to be converted. The relationship between the two depends on the wavelength of the light and how the human eye perceives it. The luminosity function describes the sensitivity of the human eye. It peaks at about 555 nm (green light), which is assigned a relative value of 1. The function decreases for wavelengths longer or shorter than 555 nm, meaning the eye is less sensitive to these colours, and thus, less luminous flux is perceived from the same amount of radiant flux. The typical conversion factor for light at 555 nm is about 683 lumens/watt. This value will vary widely depending on the exact spectrum of the light [31].

To simulate the LED in table 5.1 with a luminous flux of 1830 lumens and producing light strictly at 555 nm, the radiant flux would be approximately $1830(lm)/683(lm/W) \approx 2.68W$. In practice, most light sources emit across a broad spectrum of wavelengths, so more information about the light source, like its spectral power distribution, is needed to make an accurate conversion. In Cycles, a spotlight only limits the emission angle and does not compensate for the loss of energy emitted [32]. To compensate for the energy loss, the radiant flux was multiplied by the compensation factor: $\frac{360^\circ}{125^\circ} \Rightarrow 7.72W$. This radiant flux was set for each of the six LEDs.

Translation and Rotation of the LEDs with the Camera

Translation and rotation in Blender, especially BlenderProc, are trivial in a Python script. First, one grabs the desired object in the script and sets its location and Euler angles, or rotation matrix. When scripting the image generator, we want the LEDs to maintain their relative position to the camera as on the drone. Moving the LEDs with the camera was more troublesome than one might think. The first attempt tried to move all the individual light objects about the camera's local coordinates. Moving the light objects this way did not place them at their intended locations.

Instead, a light rig was created, depicted in figure 5.2. A to-scale 3D model of the drone was supplied, where measurements were taken to recreate the dimensions. The six planes represent the locations of the LEDs. In the script, the light rig's location and rotation are identical to the camera's global coordinates. The six small planes on the light rig are then in exact relation to the camera object as the LEDs are on the drone. Afterwards, the LEDs are set to the global coordinates of the planes, and the light rig is deleted to not interfere with the LEDs.

Figure 5.2.: REDHUS light rig created in Blender. The orientation of the light rig is approximately equal to the drone depicted in figure 1.1.

5.3. Background of Image Formation

The pinhole camera model is a simple model of image formation. The basic idea is that light rays from a point in the scene pass through a small hole and form an image on the image plane or sensor. The main elements of a pinhole camera model are the pinhole, the image plane, and the points in 3D space. When a ray of light from a 3D point passes through the pinhole, it intersects the image plane at a corresponding 2D point.

The geometry of the pinhole camera is such that a projective transformation relates the 3D points and their corresponding 2D points. This transformation depends on the intrinsic and extrinsic parameters of the camera. Intrinsic parameters include the focal length and sensor size, while extrinsic parameters involve the camera's position and orientation in space.

The Camera Matrix is a mathematical model that encapsulates the projective transformation from 3D world coordinates to 2D image coordinates. For a pinhole camera, the camera matrix $\mathbf{P}$ is a 3×4 matrix that can be factored into the product of an intrinsic matrix $\mathbf{K}$ and an extrinsic matrix $[\mathbf{R}|\mathbf{t}]$, where $\mathbf{R}$ represents rotation and $\mathbf{t}$ represents translation:

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \tag{5.1}$$

Homogeneous coordinates introduce an additional coordinate for 2D (x, y, w) and 3D (X, Y, Z, W) representations. Homogeneous coordinates allow representing infinity and translation with matrix operations. The camera matrix transforms the homogeneous coordinates of a 3D point to a 2D image point, $x = PX$, and the actual 2D image coordinates are then dehomogenized $u = x/w$, $v = y/w$.

Extrinsic parameters, encapsulated in the extrinsic matrix, describe the camera's position and orientation and external factors of the camera. The matrix includes a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$ for the transformation from the world coordinate system to the camera coordinate system:

$$[\mathbf{R}|\mathbf{t}] = \begin{bmatrix} R_{11} & R_{12} & R_{13} & t_1 \\ R_{21} & R_{22} & R_{23} & t_2 \\ R_{31} & R_{32} & R_{33} & t_3 \end{bmatrix} \tag{5.2}$$

Where R_{ij} represents the rotation matrix elements, which align the world's coordinates with the camera's coordinates, and t_i represents the elements of the translation vector, which describes the camera's location in the world.

The intrinsic matrix $\mathbf{K}$ describes the internal parameters of the camera and lens system that affect the image formation process. The parameters it encapsulates are called intrinsic because they are inherent to the camera and do not change with the camera's position or orientation in the world. These parameters include the camera's focal length, the pixel scale factors, the centre of projection, and the skew coefficient if the pixel axes are not perpendicular. One possibility of writing the upper-triangular form of the intrinsic matrix $\mathbf{K}$ as a 3×4 matrix represented as:

$$\mathbf{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

Where f_x and f_y are the focal lengths of x and y pixels. s is the skew coefficient, which accounts for the possibility of non-perpendicular axes. This is often set to zero, as modern cameras usually have

perpendicular axes. Finally, c_x and c_y are the coordinates of the principal point in pixels, typically assumed to be at the image centre. To roughly initialise f_x and f_y, we can use equations 5.4 and 5.5.

$$f_x = \frac{W}{2}\left(\tan\frac{\theta_H}{2}\right)^{-1} \tag{5.4}$$

$$f_y = \frac{H}{2}\left(\tan\frac{\theta_V}{2}\right)^{-1} \tag{5.5}$$

Where W and H are the width and height of the image, respectively, θ_H and θ_V are the lens's horizontal and vertical fields of view, respectively [33, p.36-60].

5.4. Virtual Camera Setup in Blender

The camera setup used on the REDHUS drone is `Leopard Imaging LI-IMX274-MIPI-M12`. It comprises a `Sony IMX274` image sensor and a `Sunny Optical SYD1201A` lens. Relevant characteristics of the camera setup are found in table 5.2.

Component	Characteristic	Value
Sensor	Active Pixels	$3864H \times 2196V$
	Pixel Size	$1.62\mu m \times 1.62\mu m$
Lens	Focal Length	$3.7mm$
	FOV[1]	$92°/83°/53°$

Table 5.2.: Relevant characteristics of the `Leopard Imaging LI-IMX274-MIPI-M12`.

Cycles use a pinhole camera model when the depth of field is turned off in camera settings. BlenderProc allows for easily setting the intrinsic matrix $\mathbf{K}$ in Blender. The $\mathbf{K}$ matrix in 5.6 is identical to equation 5.3, except for there being no skew factor, s, as the camera assumes rectangular pixels. This is not an issue when simulating the REDHUS camera setup. From table 5.2, the `Pixel Size` infers it has rectangular pixels, leading to a reasonable assumption that the skew factor is zero. Moreover, Szeliski [33, p.56] states "solid-state manufacturing techniques render this negligible".

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

[1] Field of View (Diagonal/Horizontal/Vertical)

6. Exposure Control and Tone Mapping of Rendered Colour Images

6.1. Background of Exposure Control

Exposure is a vital concept in the field of photography and videography. It refers to the amount of light that reaches the camera sensor, which is vital to the quality and aesthetic of the resulting image or video. Exposure control refers to the ability of a photographer or videographer to manage and manipulate the amount of light that enters the camera.

Three fundamental parameters control exposure, forming the *exposure triangle* consisting of aperture, shutter speed, and ISO. *Aperture* denotes the size of the opening in the lens that allows light to pass into the camera, typically measured in f-stops, i.e. f/2.8, f/5.6, f/8. A lower f-stop number means a larger aperture, more light entering the camera, resulting in a brighter image and vice versa. The aperture also affects the depth of field. A larger aperture provides a shallower depth of field, focusing on the subject and blurring the background.

Shutter Speed indicates the time the camera's sensor is exposed to light, generally measured in fractions of a second, i.e. 1/60, 1/250, 1/1000. A faster shutter speed will freeze action but let in less light, while a slower shutter speed will blur motion but let in more light. *ISO* refers to the sensitivity of the camera's sensor to light. Lower ISO values mean lower sensitivity to light. They are used in bright conditions to prevent overexposure, while higher ISO values increase sensitivity to light, which is helpful in darker conditions. However, higher ISO values also increase the noise or grain in an image, decreasing its overall quality [34, p.16-20]. Balancing these three parameters is crucial to achieving the desired exposure.

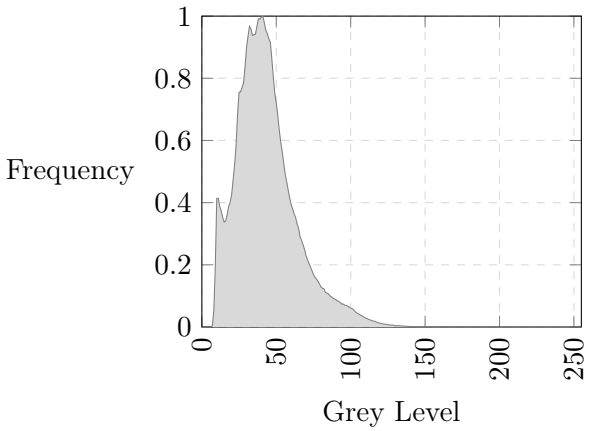
6.2. Exposure Control for Rendered Colour Images

Performing exposure control in Blender differs from how it is used in a physical camera, as rendering a 3D scene differs from capturing a physical scene with a camera. The concept of *aperture* in Blender relates primarily to the depth of field effect. A smaller F-stop value corresponding to a larger physical aperture leads to a smaller depth of field, making the scene appear to have more blur in areas that are not in focus.

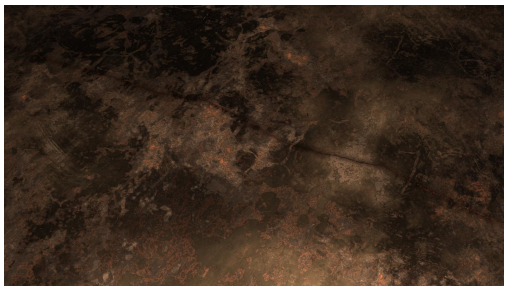
In the context of Blender, *shutter speed* is mainly used when creating animations, as it influences the amount of motion blur. The motion blur settings control the shutter speed in Blender. Setting it to a lower value would result in less motion blur and vice versa. In Blender, *ISO* is generally simulated by adjusting the strength of light sources or the exposure of the final render. Higher ISO values would result in a brighter image, and lower ISO values would result in a darker image.

When rendering images with light sources simulating the actual REDHUS LEDs as described in 5.1, the images came out looking underexposed and with a significant skew in their grey scale histograms as in figure 6.1a. The solution was to withdraw the approach to precisely simulate LEDs and set the lights from an artistic and histogram perspective to create desirably illuminated images. This

is illustrated in figure 6.2a, where the illumination was set from an artistic perspective. We also observe in the figure that the grey levels are more uniformly distributed.

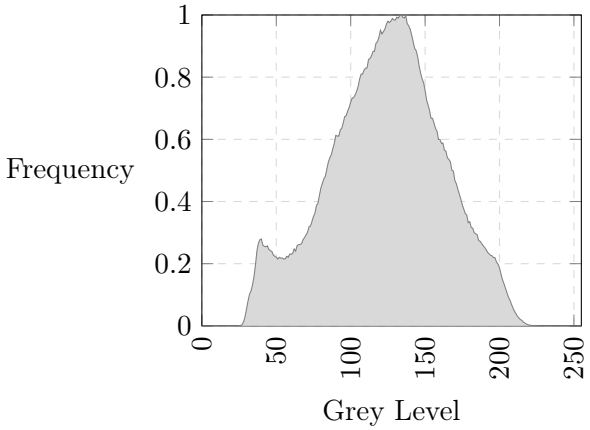


(a) Greyscale histogram with REDHUS LED power of $\approx 15W$.

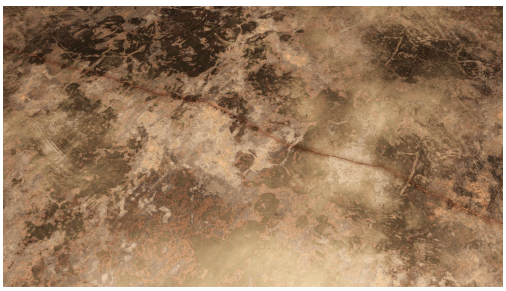


(b) Renderd image with with REDHUS LED power of $\approx 15W$.

Figure 6.1.: Grey scale histogram in figure 6.1a of the image in figure 6.1b. We can see from the combination of the image and histogram that the image is underexposed, as the histogram is skewed to the lower grey levels in the histogram.



(a) Greyscale histogram with REDHUS LED radiant flux of $\approx 100W$.



(b) Rendered image with with REDHUS LED radiant flux of $\approx 100W$.

Figure 6.2.: Grey scale histogram in figure 6.2a of the image in figure 6.2b. We can see from the combination of the image and histogram that the image is more correctly exposed, as the grey levels in the histogram are more uniformly distributed.

6.3. Background of Tone Mapping

The human visual system can perceive a wide range of brightness levels, from very dim to very bright, and can adapt to changes in lighting conditions. However, digital cameras and displays have a limited range of brightness levels that they can capture or reproduce, often much smaller than the range of brightness levels in a real-world scene.

This means that our eyes can perceive details in both very bright and very dark areas of a scene simultaneously, referred to as high dynamic range (HDR). However, most displays can only represent

a low dynamic range (LDR). This means they cannot display the full range of brightness levels that the human visual system can see in an HDR scene. This discrepancy between HDR images or scenes and LDR display capabilities is where tone mapping comes in. *Tone mapping* is a technique used in image processing to map one set of colours to another to approximate HDR images' appearance in media with a lower dynamic range [33, p.620-634].

The goal of tone mapping is to preserve as much of the perceived detail, contrast, and colour from the original HDR scene as possible in the final LDR image. Tone mapping involves a trade-off between compressing the dynamic range and preventing the loss of detail or the introduction of artefacts. Tone mapping does not just apply to brightness levels but also to colours.

The Reinhard algorithm [35] is a technique for tone mapping. It is a global operator meaning that the same operation is applied to all pixels in an image, based on the overall characteristics of the image, rather than considering each pixel in its local context. The algorithm is based on a model of human perception and adapts to the overall luminance of the image. A few parameters can control it to adjust the brightness and contrast of the output.

The basic form of the Reinhard algorithm can be expressed as follows:

$$L(d) = \frac{L(w)(1 + \frac{L(w)}{L_{max}^2})}{1 + L(w)} \tag{6.1}$$

where

- $L(w)$ is the world luminance or the actual brightness of the pixel in the HDR image.
- $L(d)$ is the display luminance or the pixel's brightness in the LDR image.
- L_{max} is the maximum luminance in the scene.

Working with HDR images and performing operations such as tone mapping are performed in a linear colour space such as Linear RGB. Operations are performed in linear colour spaces because many image operations assume a linear relationship between pixel values and light intensity, which holds in a linear colour space. Operations like blending, blurring, and tone mapping, like the Reinhard algorithm, all expect this linear relationship.

However, displays used to view images are generally designed around the sRGB colour space, which is non-linear. Furthermore, these displays expect image data to be gamma corrected for better utilization of data based on human visual perception. As a result, before the tone-mapped image is displayed, it is typically converted from the linear colour space to the sRGB colour space. For example, in a typical workflow, one might start with an HDR image in a linear colour space, apply the Reinhard tone mapping algorithm to reduce its dynamic range, and then convert the resulting image to the sRGB colour space before display.

A simple approximation from Linear RGB to sRGB is to apply a power function with a gamma value of 2.2. The conversion from linear RGB to sRGB would then be as simple as raising each colour channel to the power of 1/2.2:

$$c_{\text{sRGB}} = c_{\text{Linear}}^{\frac{1}{2.2}} \tag{6.2}$$

6.4. Tone Mapping of Rendered Colour Images

In the pipeline, rendered images from Blender are in the OpenEXR format [36]. OpenEXR is an HDR image file format that stores a wide range of colours and brightness values in Linear RGB.

Post-processing an image in OpenEXR format with colours in linear RGB involves tone mapping and converting it to a lower bit depth format for display on standard monitors with LDR. In the pipeline, the process is performed using the Python library OpenCV.

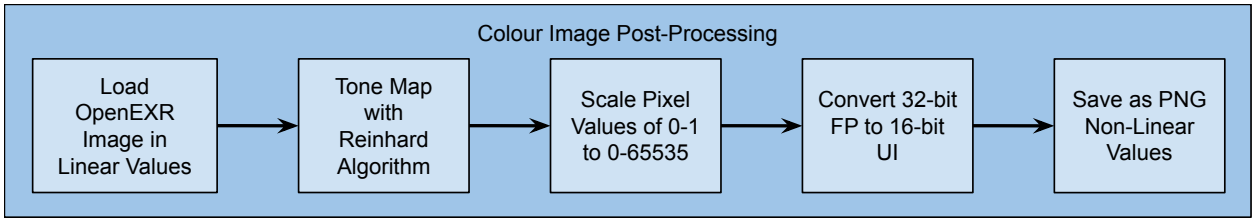
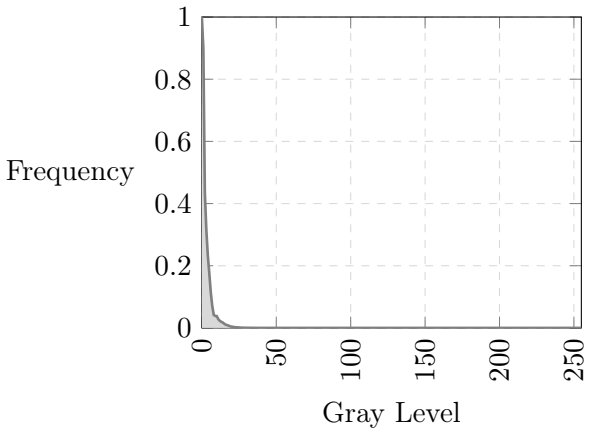


Figure 6.3.: Overview of the post-processing of colour images in the pipeline

A rendered image in OpenEXR is loaded in the pipeline, and the Reinhard algorithm is applied to the image. Once tone mapped, the image is still in 32-bit floating-point format, but the brightness range is compressed, and the image can be displayed on a standard monitor. To save the image in a more suitable format, it must be converted to a lower-bit depth format such as 16-bit PNG. The image is saved with a bit depth of 16 using unsigned integers. The post-processing code can be found in appendix A.

An example of an image in linear values that have not been tone mapped is depicted with its greyscale histogram in figure 6.4. From the histogram, it might appear that the conversion from 32-bit floating-point (0-1) to 16-bit unsigned integer (0-65535) has not been performed. This is not the case. This is how the linear values appear. The same image has been successfully tone mapped in figure 6.2 where the pixel values are non-linear, which is desired.



(a) Greyscale histogram with of an image in linear RGB.

(b) Rendered colour image in linear RGB.

Figure 6.4.: Grey scale histogram in figure 6.4a of the image in figure 6.4b. Here we observe an image with linear values. The image in figure 6.4b appears completely dark but contains information if inspected closely.

7. Scripting the Image Generator

7.1. Scripts for the Image Generator

The algorithm in 1 is the main script that renders a single pair of an RGB image and its segmentation mask. This algorithm should be thought of as the `Python Script` in figure 2.1 and is what is executed in the `Bash/Batch Script` also seen in the figure. In the algorithm, 2 object materials are selected and set using a uniformly distributed variable. Furthermore, in algorithm 3 the camera and light positions are sampled and then set. The parameters set for rendering (`RENDER_PARAMS`) are given in table 7.1.

Algorithm 1 Image Generator

```
function MAIN()
    BlenderProc.initialize()          ▷ Initializes Blender settings and cleans the scene
    LoadScene(cube.blend)                                  ▷ Loads the created blender scene
    SetRenderParameters(RENDER_PARAMS)                     ▷ See table 7.1 for RENDER_PARAMS
    SetCameraParameters()             ▷ The K matrix is set as described in section 5.4
    SetObjectMaterial()
    ReplaceRandomPlane()              ▷ Replaces ones of the cubes sides with crack plane
    SetCrackParameters()                                   ▷ See table 3.1 for crack parameters
    SetCameraAndLights()
    data ← RenderScene()
    WriteHDF5(data)                   ▷ Writes the rendered data to HDF5 containers
end function
```

Algorithm 2 Setting Object Material

```
function SETOBJECTMATERIAL()
    MeshObjects ← GetMeshObjects()          ▷ Retrieves all mesh objects in the scene
    u ← 𝒰(0, 3)                  ▷ Sample a pseudorandom value from a uniform distribution
    if u < 1 then
        SetPBRMaterial(MeshObjects)
    else if u ≥ 1 and u ≤ 2 then
        SetStainedMetalMaterial(MeshObjects)
    else if u > 2 then
        SetPaintedMetalMaterial(MeshObjects)
    end if
end function
```

Algorithm 3 Setting Camera and Light Positions

```
function SetCameraAndLights()
    CrackPlaneLoc ← CrackPlane.GetLocation()
    CameraLoc ← SampleCameraLoc()                    ▷ Samples a location within the cube
    CameraRot ← RotFromForwardVec(CrackPlaneLoc, CameraLoc) ▷ Calculates the rotation matrix given the camera location and crack planes location
    AddCameraPose(CameraLoc, CameraRot)   ▷ This is where the camera is positioned in the render
    LightRig ← LoadLightRig()
    LightRig.SetLocAndRot(CameraLoc, CameraRot) ▷ The light rig object has the identical position as the camera. The light objects are not positioned at the precise camera location
    LightRig.SetLights()                  ▷ Creates and sets the light objects on the light rig
    LightRig.Delete() ▷ Deletes the light rig to not interfere with the light. The light objects are not deleted
end function
```

Cycles Parameters	Value
cycles.max_subdivisions	12.0
cycles.dicing_rate	1.0
CrackPlane.cycles.dicing_rate	0.5

BlenderProc Function	Parameter	Value
set_render_devices	use_only_cpu	False
	desired_gpu_device_type	OPTIX
enable_experimental_features		
set_max_amount_of_samples	samples	256
set_light_bounces	max_bounces	128
	diffuse_bounces	1024
	glossy_bounces	1024
	transmission_bounces	1024
	volume_bounces	1024
	ao_bounces_render	1024
	transparent_max_bounces	1024
set_output_format	file_format	OPEN_EXR
	color_depth	32
	enable_transparency	False
set_world_background	color	$(0, 0, 0)$
	strength	1

Table 7.1.: RENDER_PARAMS set in algorithm 1.

8. Examples of Generated Images

8.1. Physically Based Rendering Materials for Photorealistic Ship Tank Surfaces

Materials created with the approach from section 4.2 are presented in this section. Images with cracks have the accompanying segmentation mask displayed. The ones without cracks do not have the segmentation mask displayed.

Examples of Generated Images with Cracks

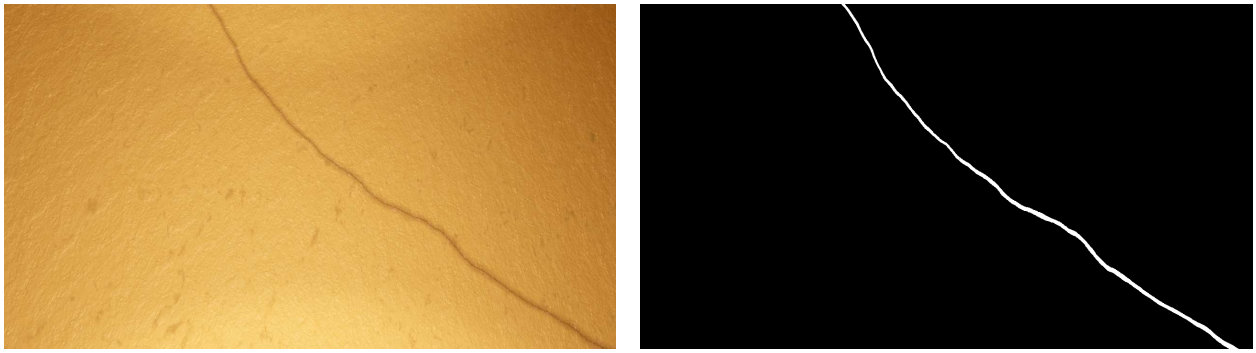


Figure 8.1.:

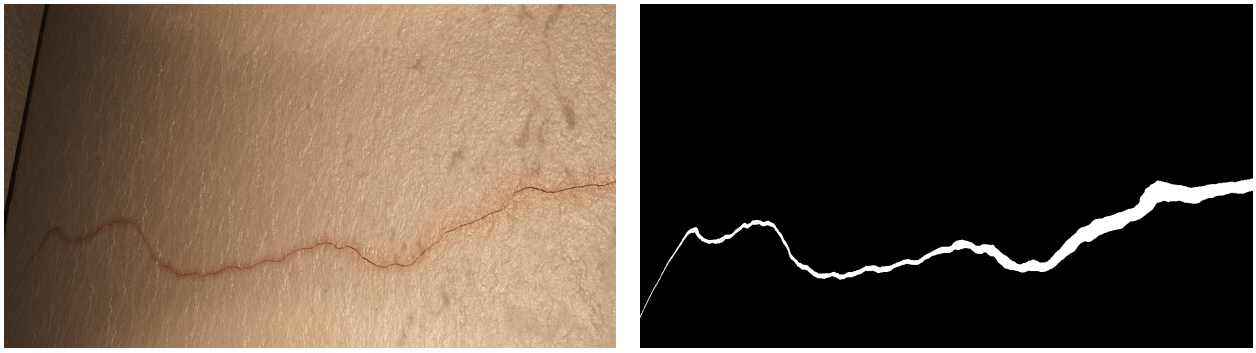


Figure 8.2.:

Figure 8.3.:

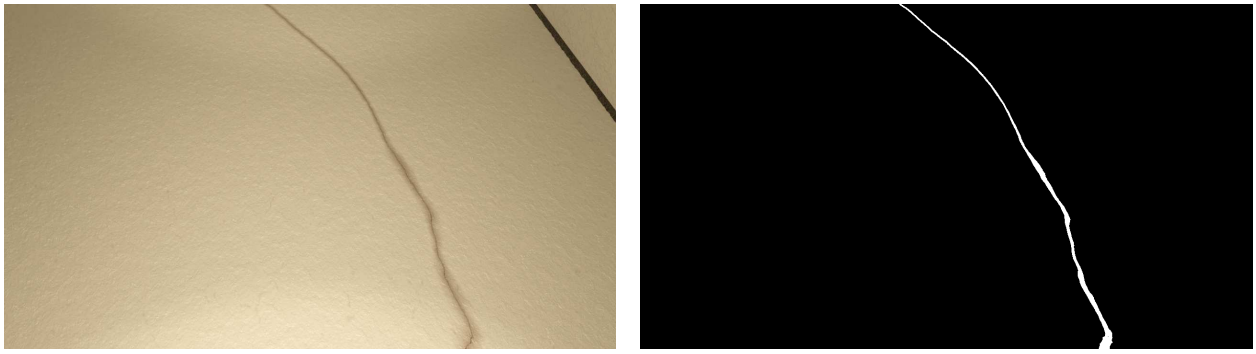


Figure 8.4.:

Figure 8.5.:

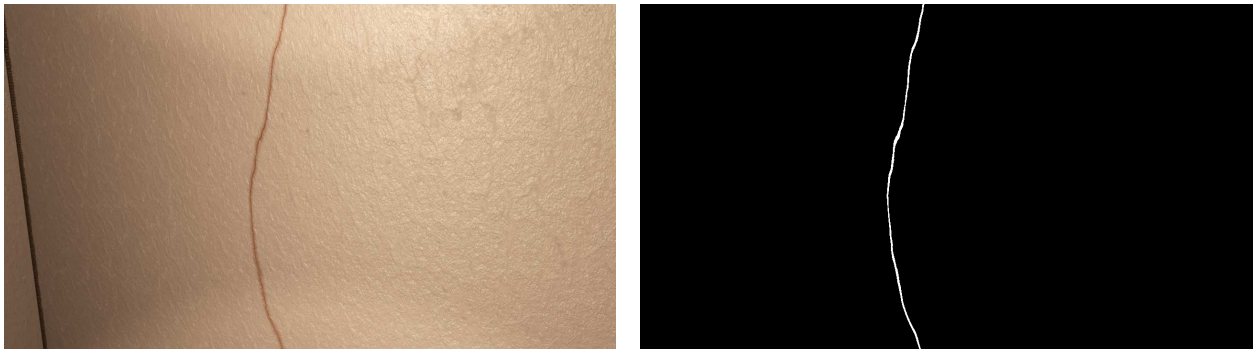


Figure 8.6.:

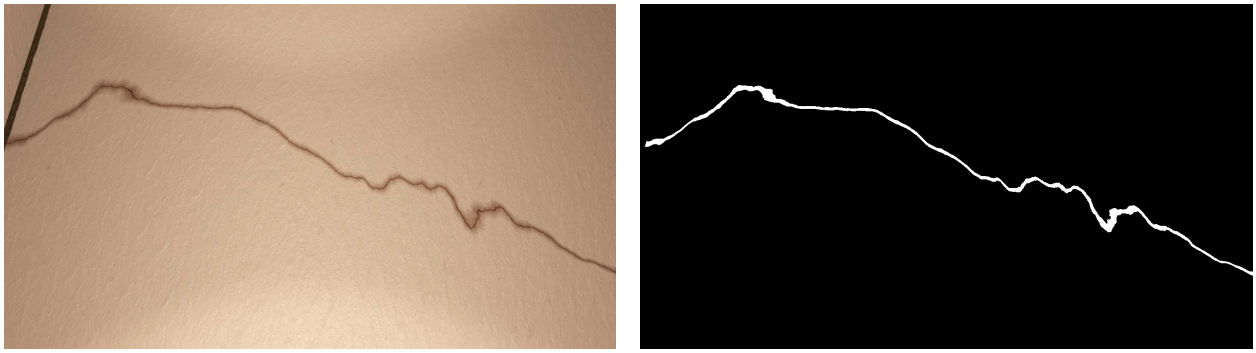


Figure 8.7.:

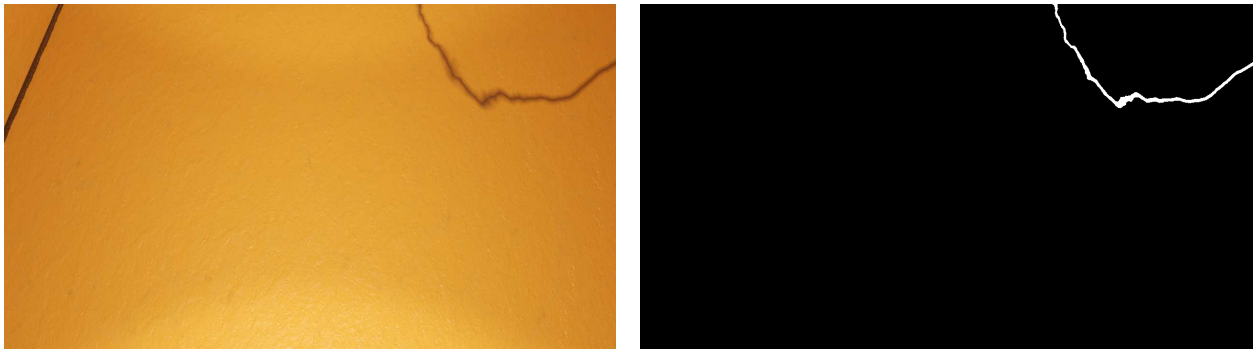


Figure 8.8.:

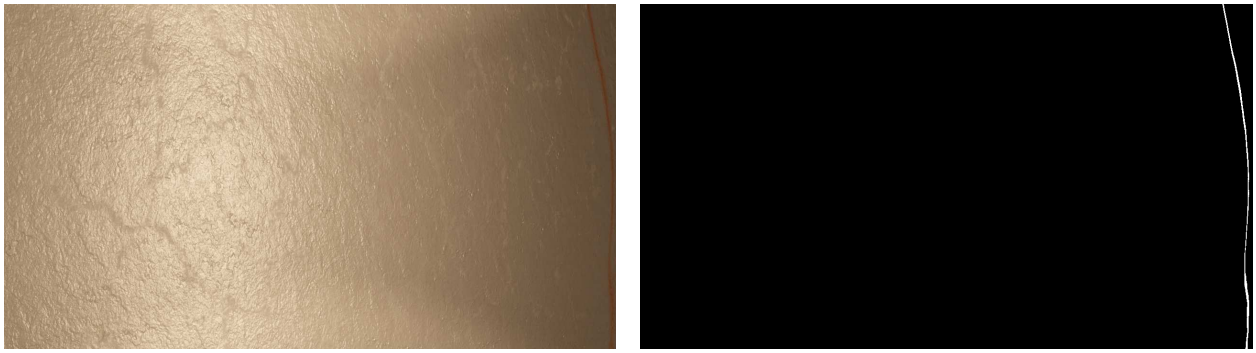


Figure 8.9.:

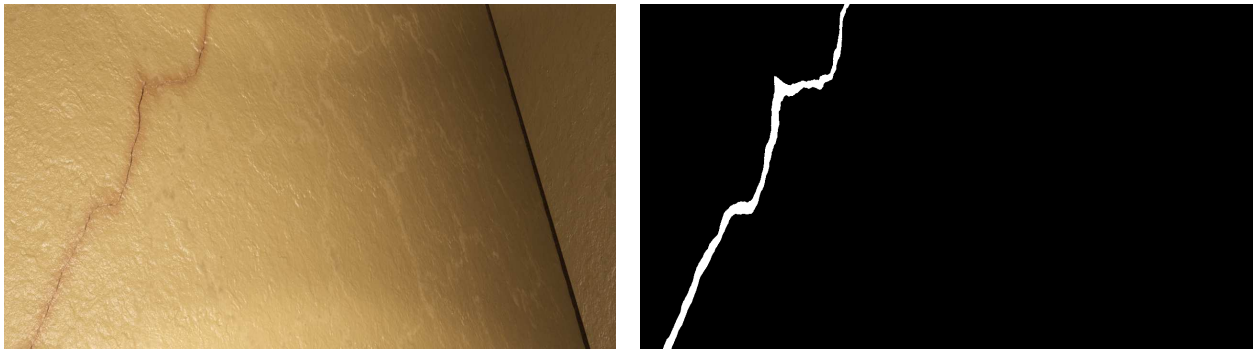


Figure 8.10.:

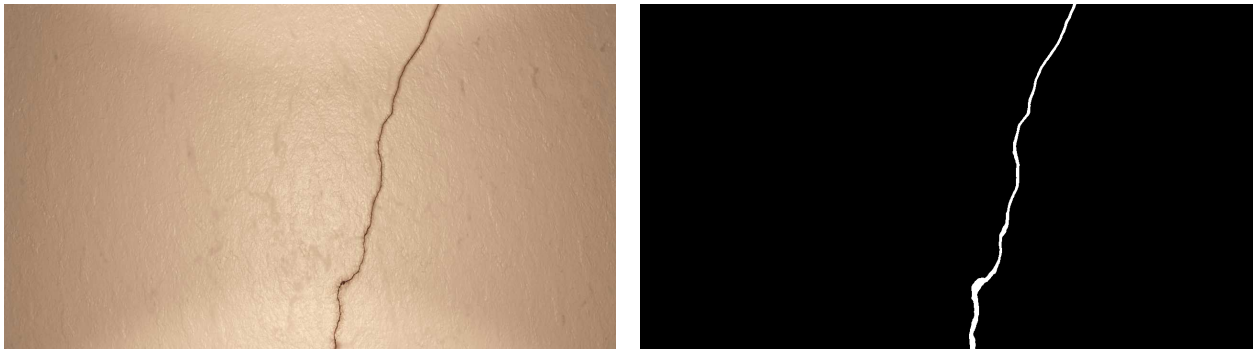


Figure 8.11.:

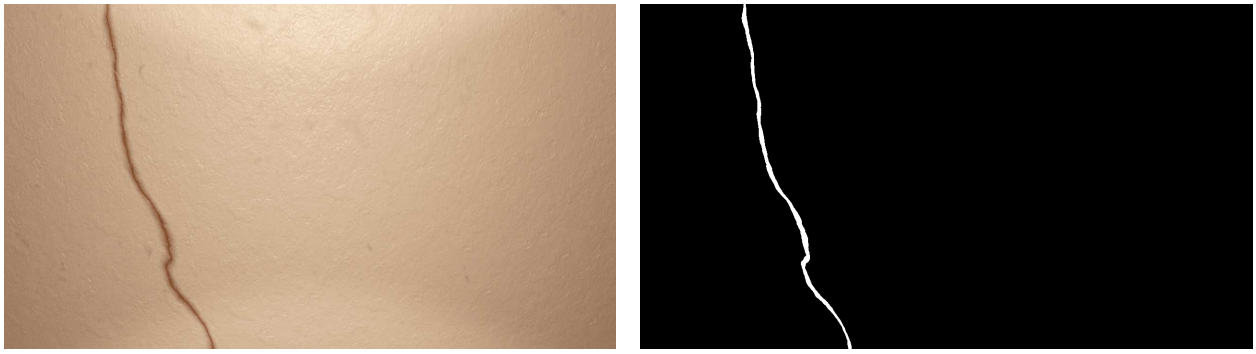


Figure 8.12.:

Figure 8.13.:

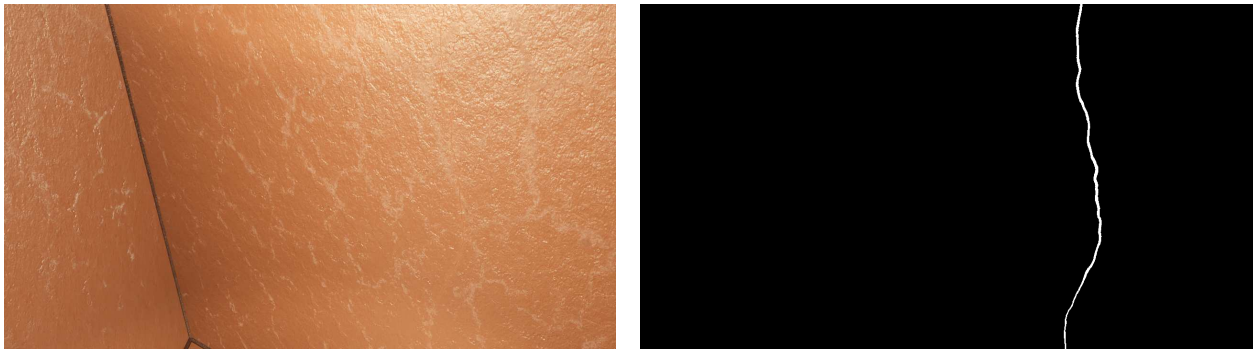


Figure 8.14.:

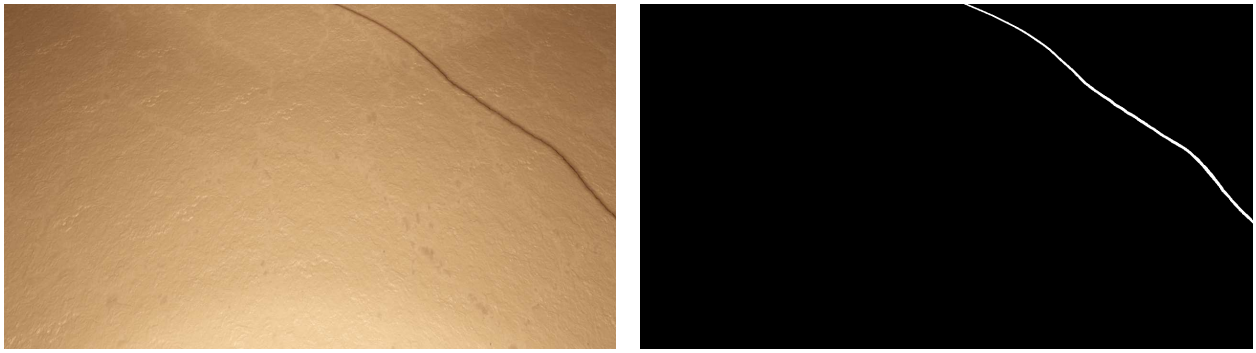


Figure 8.15.:

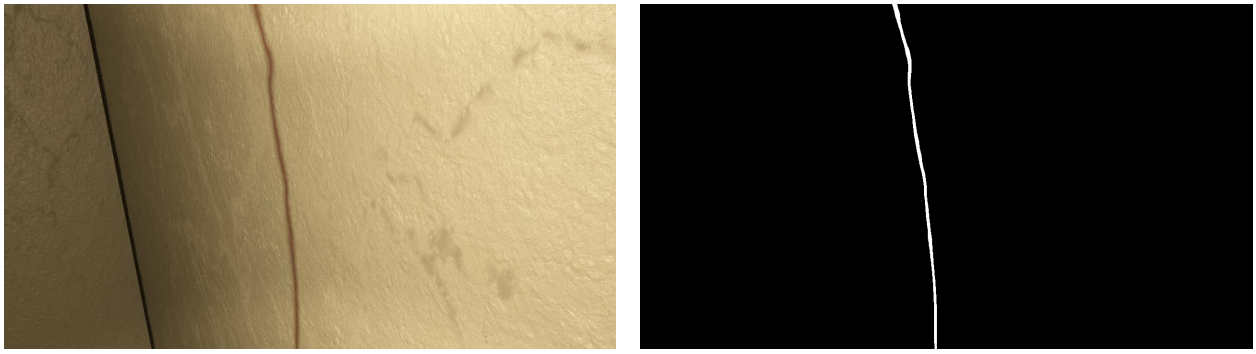


Figure 8.16.:

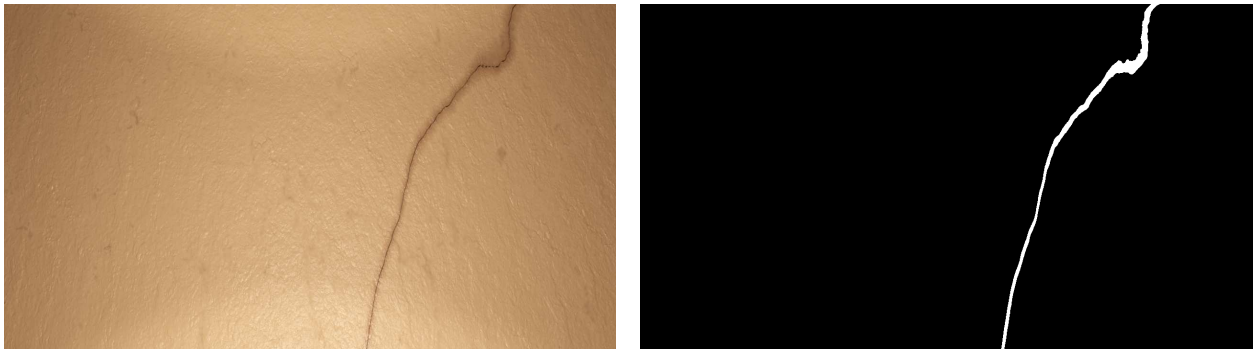


Figure 8.17.:

Figure 8.18.:

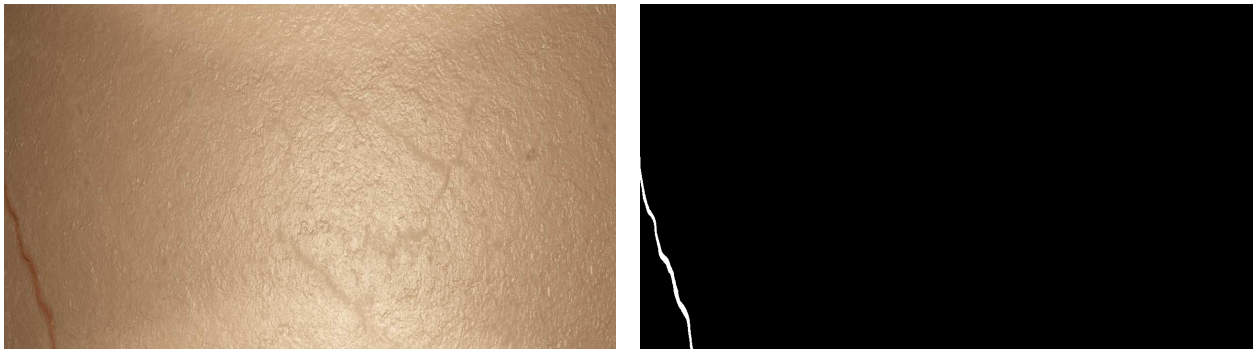


Figure 8.19.:

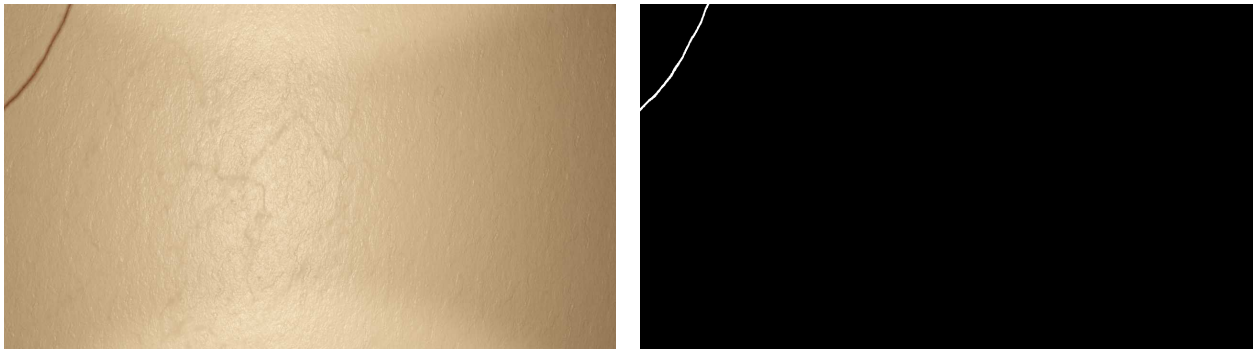


Figure 8.20.:

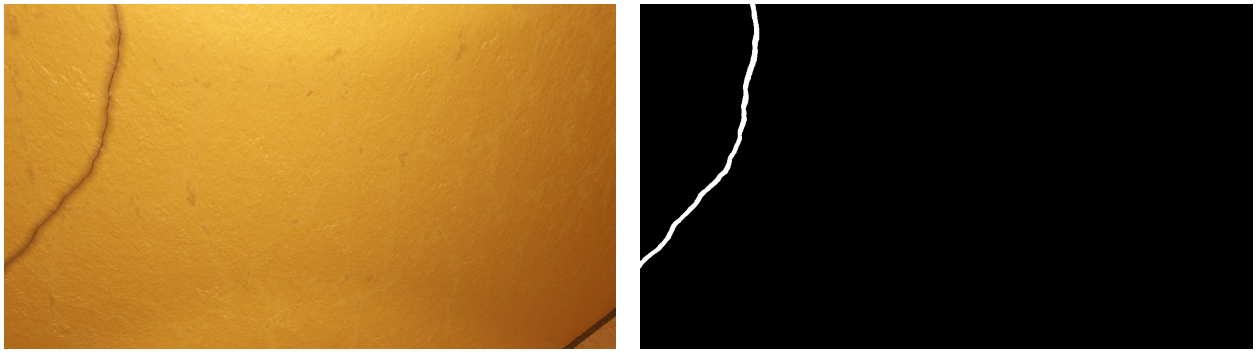


Figure 8.21.:

Figure 8.22.:

Examples of Generated Images without Cracks

Figure 8.23.:

Figure 8.24.:

8.2. Procedurally Generated Stained Metal Texture for Photorealistic Ship Tank Surfaces

Materials created with the approach from section 4.3 are presented in this section. Images with cracks have the accompanying segmentation mask displayed. The ones without cracks do not have the segmentation mask displayed.

Examples of Generated Images with Cracks

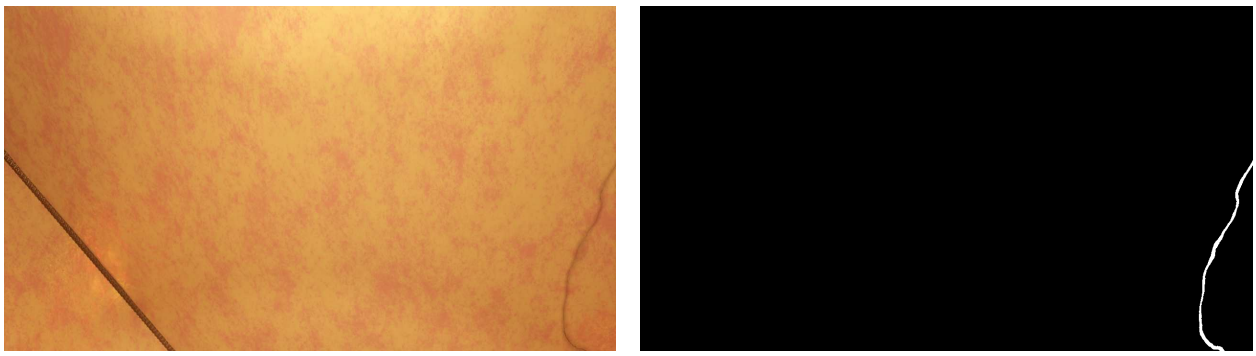


Figure 8.25.:

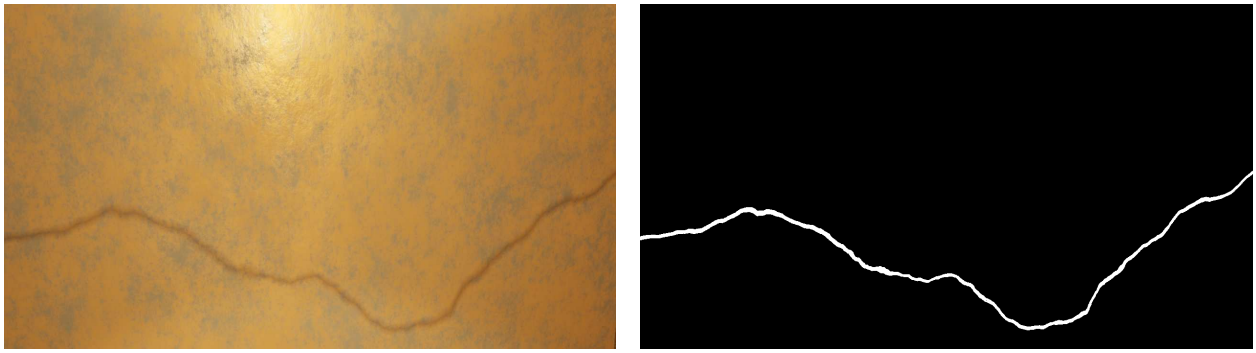


Figure 8.26.:

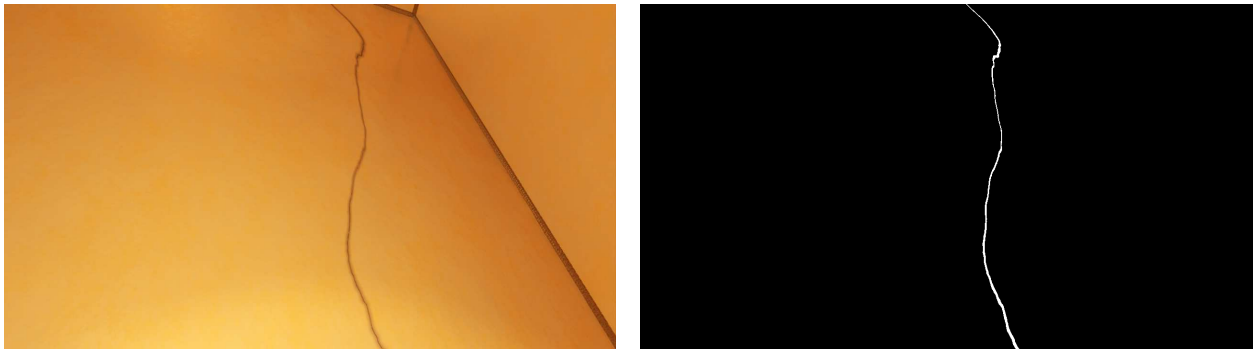


Figure 8.27.:

Figure 8.28.:

Figure 8.29.:

Figure 8.30.:

Figure 8.31.:

Figure 8.32.:

Figure 8.33.:

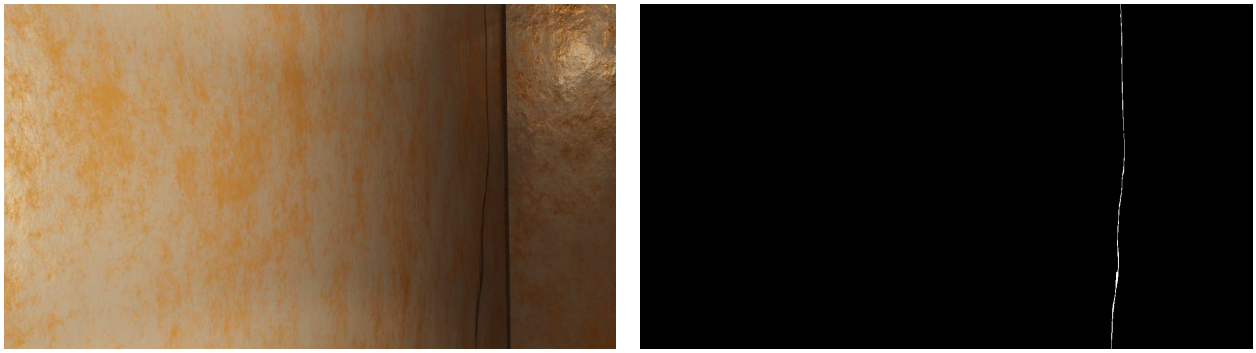


Figure 8.34.:

Figure 8.35.:

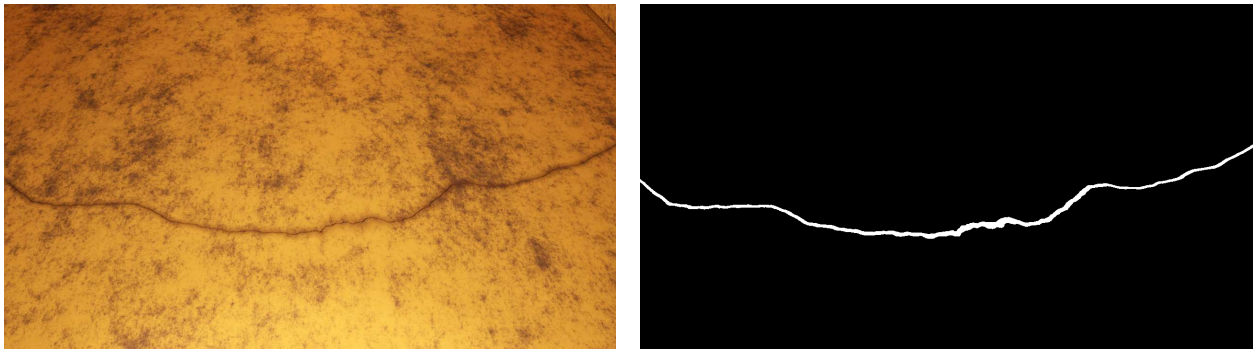


Figure 8.36.:

Figure 8.37.:

Figure 8.38.:

Examples of Generated Images without Cracks

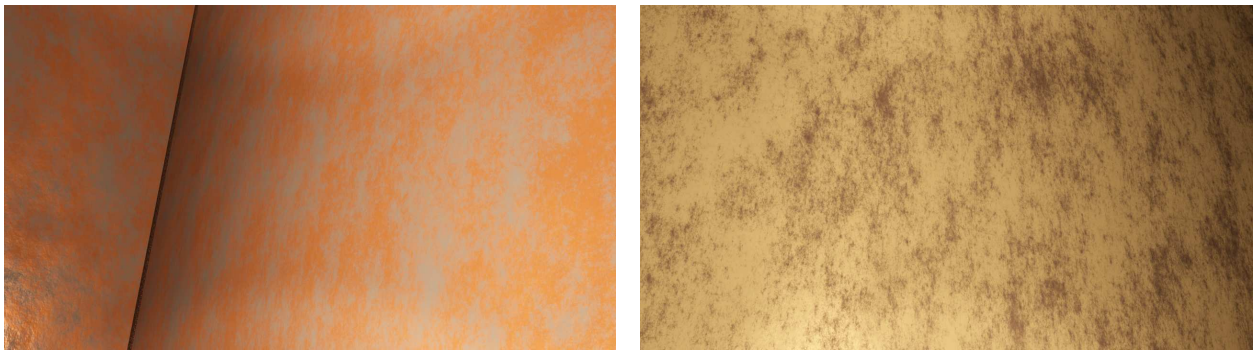


Figure 8.39.:

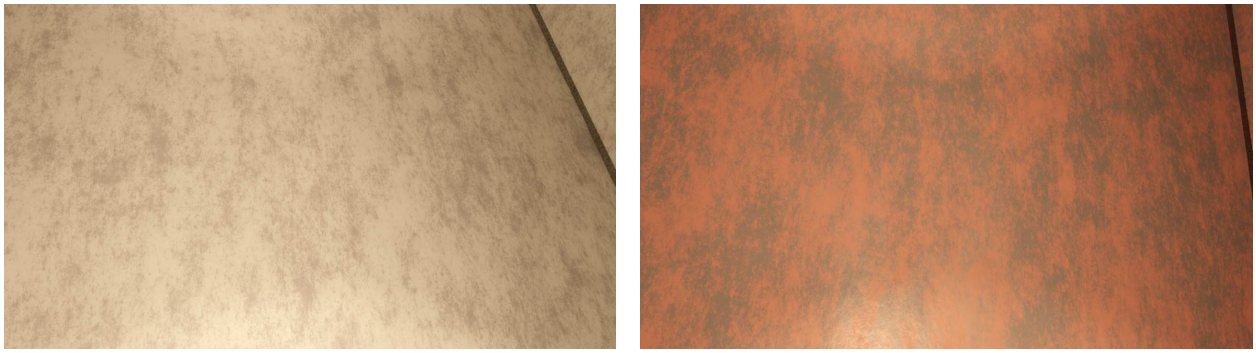


Figure 8.40.:

8.3. Procedurally Generated Painted Metal Texture for Photorealistic Ship Tank Surfaces

Materials created with the approach from section 4.4 are presented. Images with cracks have the accompanying segmentation mask displayed. The ones without cracks do not have the segmentation mask displayed.

Examples of Generated Images with Cracks

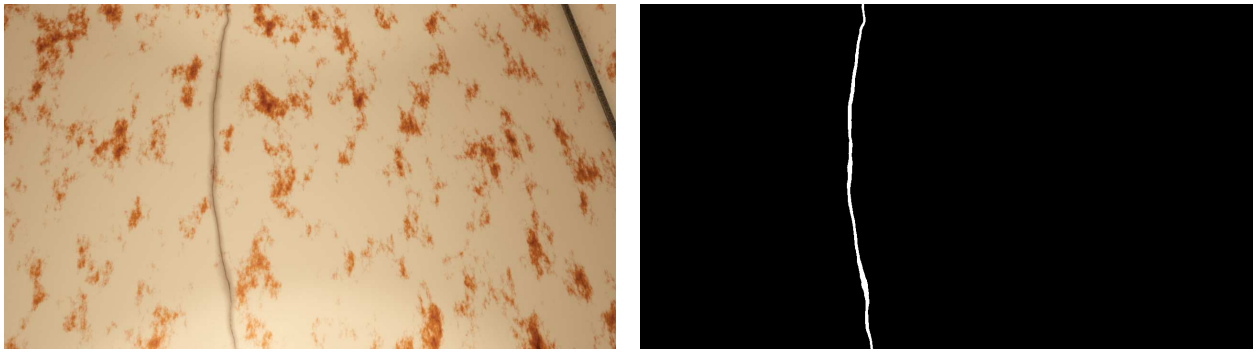


Figure 8.41.:

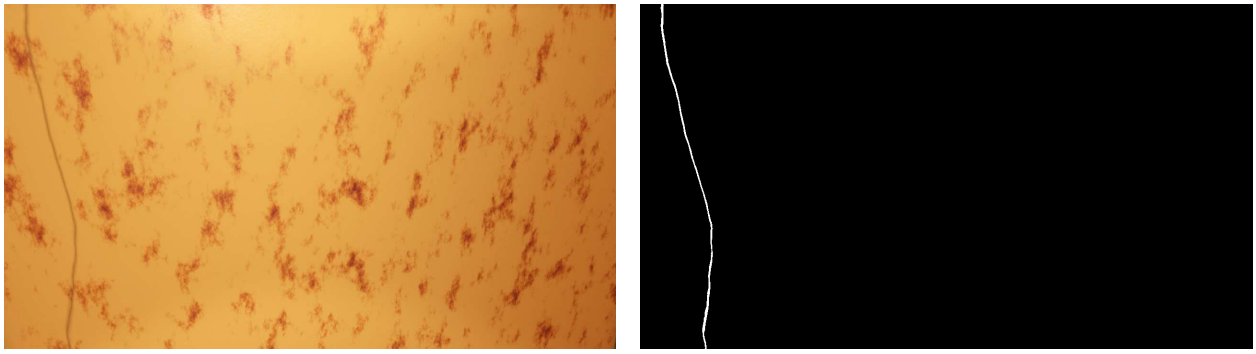


Figure 8.42.:

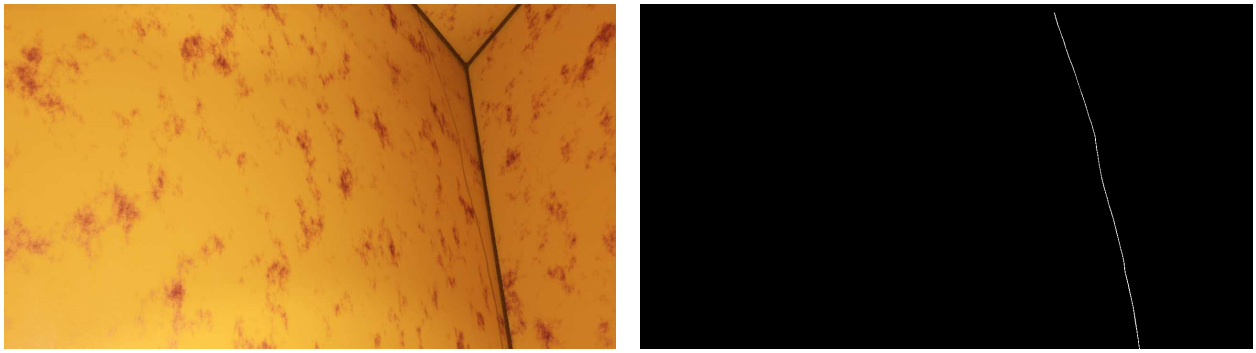


Figure 8.43.:

Figure 8.44.:

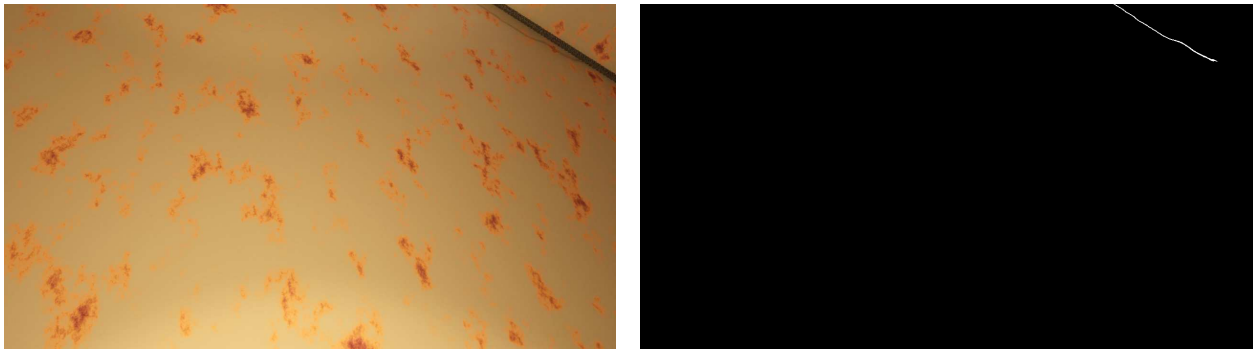


Figure 8.45.:

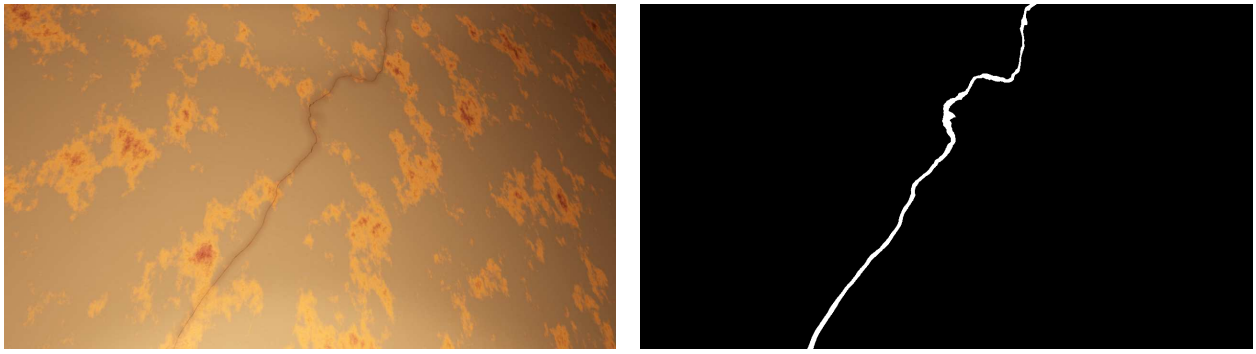


Figure 8.46.:

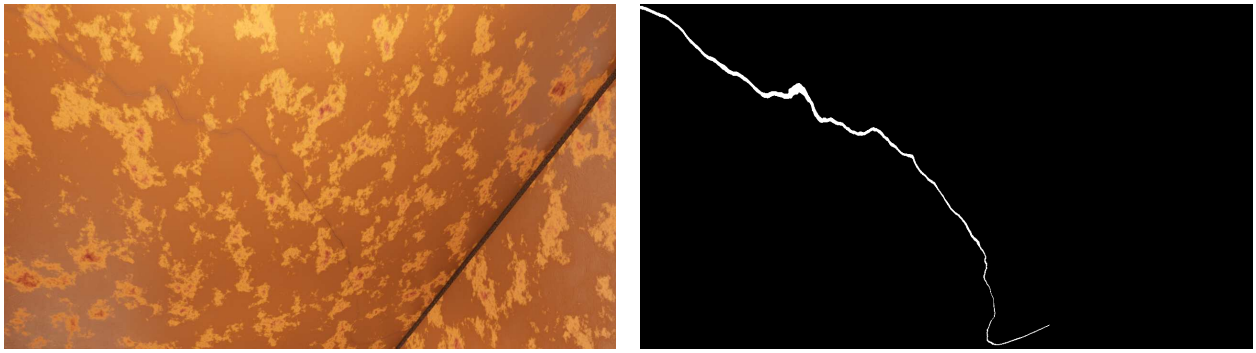


Figure 8.47.:

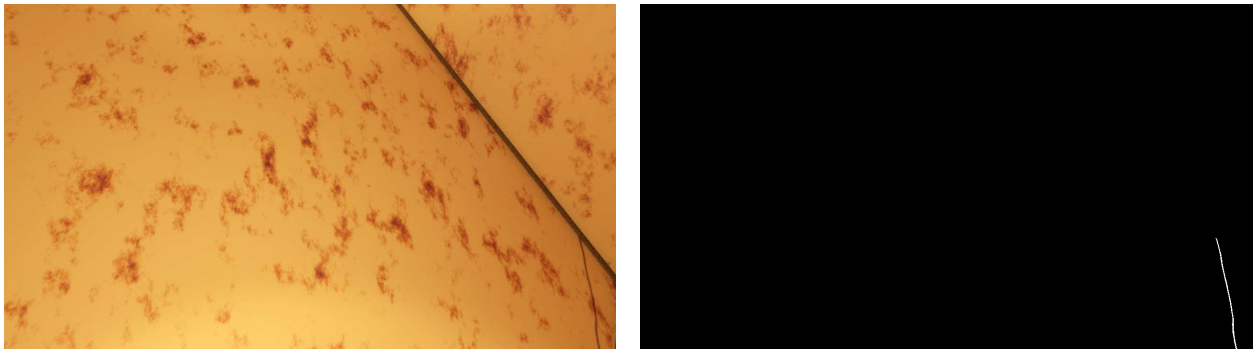


Figure 8.48.:

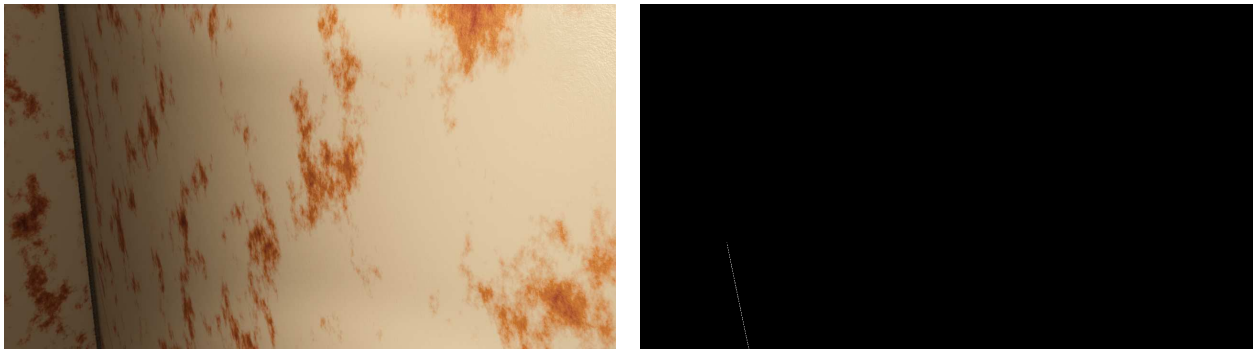


Figure 8.49.:

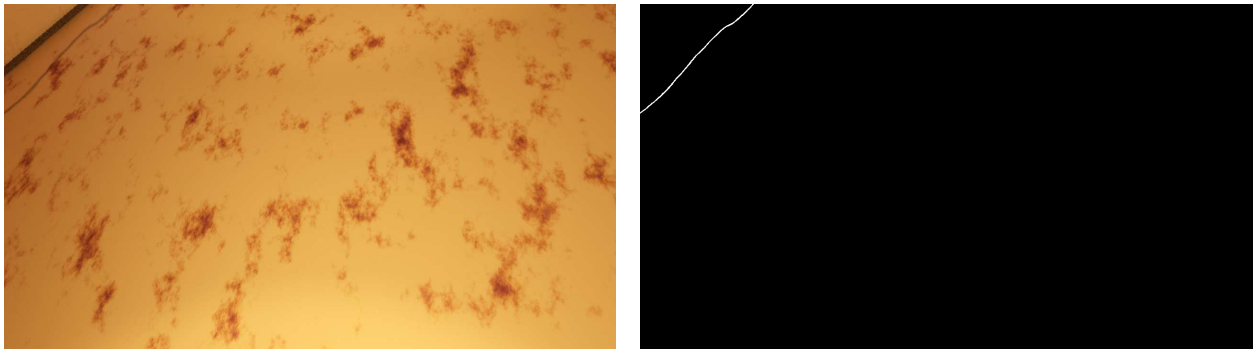


Figure 8.50.:

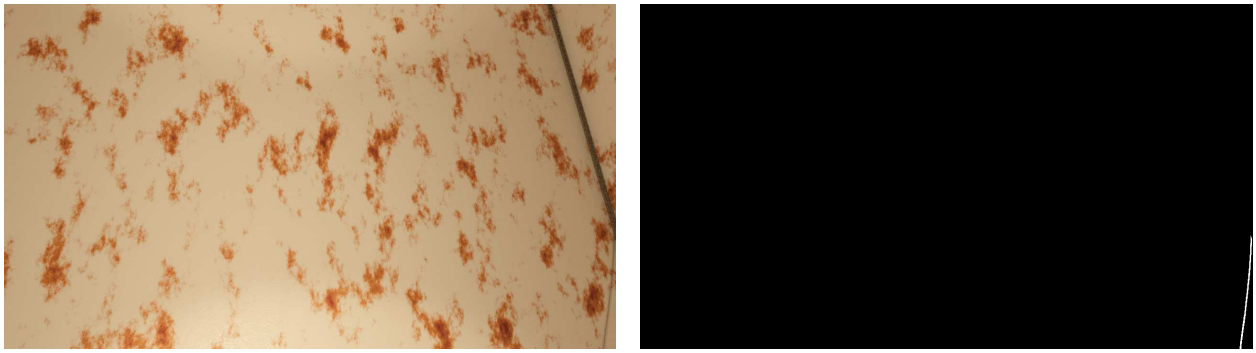


Figure 8.51.:

Figure 8.52.:

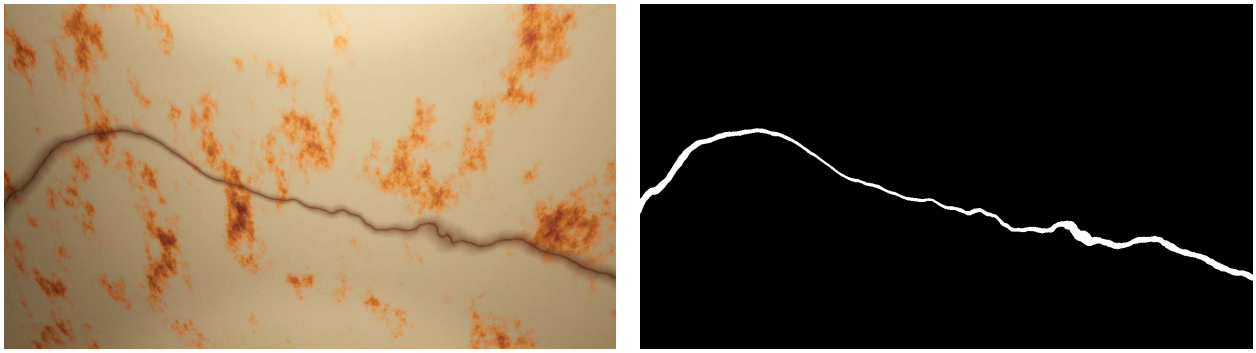


Figure 8.53.:

Figure 8.54.:

Examples of Generated Images without Cracks

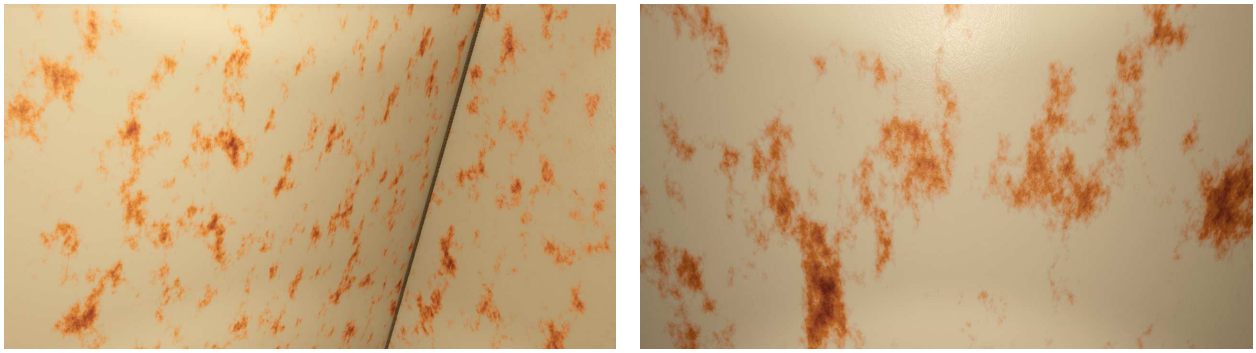


Figure 8.55.:

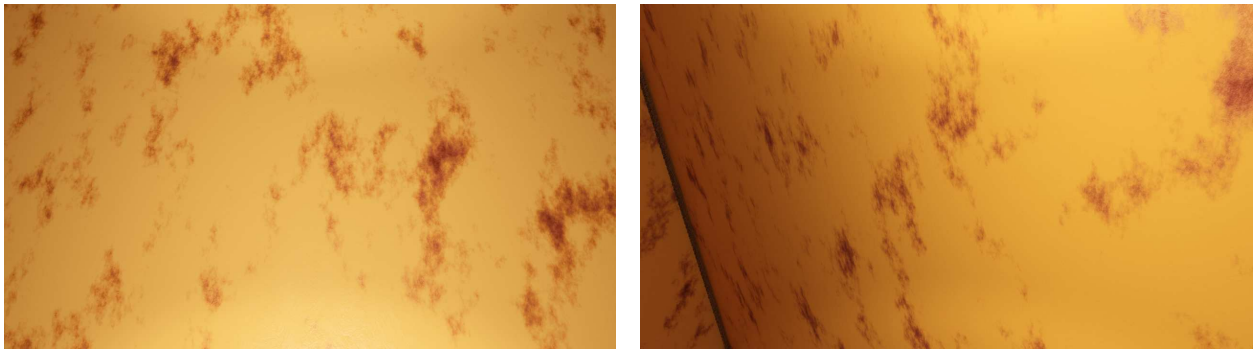


Figure 8.56.:

9. Attempted Approaches

Approaches that were attempted and unfortunately did not come to fruition are showcased here. This is to describe the attempted approaches and serve as examples of potential pitfalls to avoid.

9.1. Attempted Geometric Modelling of the Scene

The scene used in the rendering, described in section 3.2, is simple as it is only a cube. What would have been ideal is an actual 3D model of a ship tank or multiple ship tanks. DNV, unfortunately, could not share any of their 3D models, as they were under, i.e. non-disclosure agreements with their clients.

Not having a 3D model led to browsing online for a ship tank. Unfortunately, a ship tank model was not found, so an alternative choice had to be made. The choice landed on an industrial scene that could represent the geometry found in a ship tank. A 3D model of a warehouse was found online and attempted to serve as the scene. Renders of the warehouse are depicted in figure 9.1.

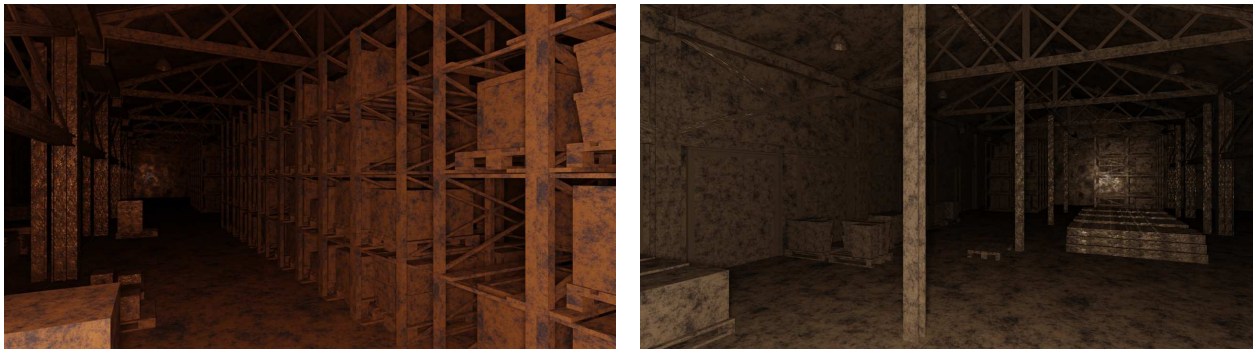


Figure 9.1.: Render from the warehouse scene. Please note that the materials applied to objects in the scene have not been properly UV mapped, as would have been performed if utilised.

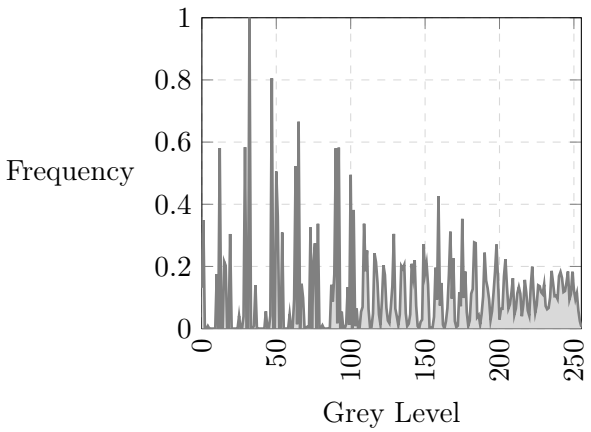
The warehouse scene has a significantly more exciting geometric structure. However, it had problems in terms of scripting the image generation. It was unclear how to apply a crack to existing objects in the scene due to their mesh structure and whether it was feasible without modifications to all mesh objects. It was believed that the issue was that objects already had complex mesh structures, causing the further subdivision to go mayhem. Subdivision quickly becomes very memory intensive and commonly uses upwards and occasionally double-digit GBs when rendering a scene utilising the cube in the current pipeline. When applying a crack and subdividing an object in the warehouse scene, the Cycles render engine attempted to allocate over 1TB of memory, eventually crashing.

There were also difficulties when adding a plane with a crack on it suitably in the scene, as the rotation of objects was not behaving as expected. I.e. when a warehouse wall was replaced with a crack plane, the crack plane's orientation was incorrect and not as expected. This could possibly be solved by various `Apply Transform` on the objects in Blender.

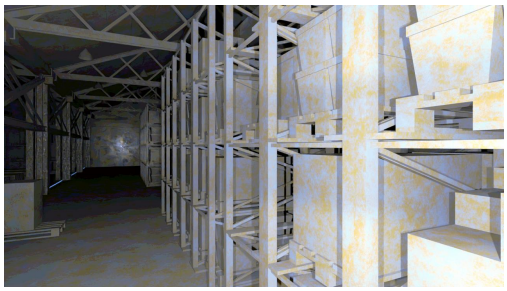
9.2. Attempted Exposure Control and Tone Mapping Approaches

Cycles render engine is not a perfectly radiometric render engine, resulting in that setting, i.e. radiant flux of physically correct values from the real world, does not result in realistic renders. This was before realising that the lights' radiant flux should be set from a more artistic perspective rather than from a radiometric and photometric perspective. Prior to this, several approaches were attempted to alleviate the underexposed images as indicated by their look and histograms. Histogram equalisation can be done in multiple colour spaces. It was attempted in RGB and approached where the images were first converted to YUV, YCrCb, and CIELAB (L*a*b*) colorspaces.

Y in YUV, Y in YCrCb, and L in L*a*b* was the image channels where equalisation was attempted. In addition to standard histogram equalisation, CLAHE (Contrast Limited Adaptive Histogram Equalization) was attempted, albeit unsuccessful.

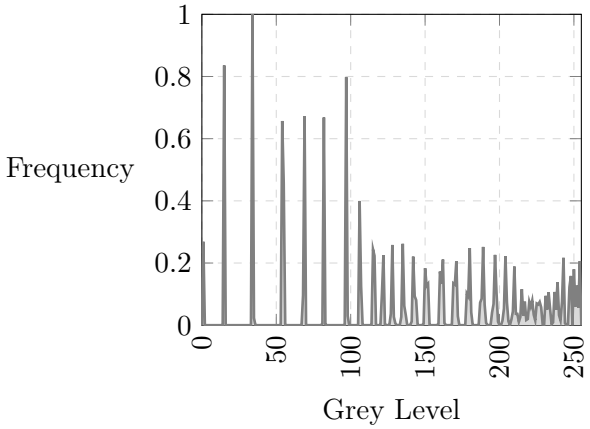


(a) Greyscale histogram of an image where histogram equalisation has been performed individually on the RGB channels. Notice the choppiness of the grey levels.

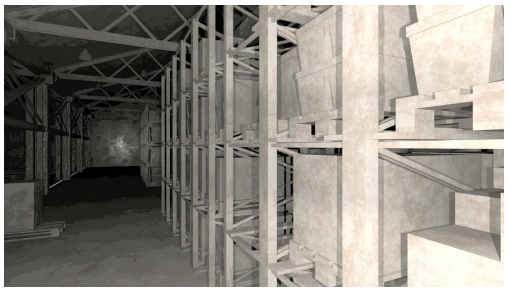


(b) Rendered image where histogram equalisation has been performed individually on the RGB channels. Notice the abrupt changes in colours on the floor of the scene.

Figure 9.2.: Grey scale histogram in figure 9.2a of the image in figure 9.2b. Histogram equalisation has been performed individually on the RGB channels. This image has better exposure and looks decent at first glance. However, looking at the greyscale histogram, one notices a choppiness in the grey levels. At closer inspection of the rendered image, we can spot these abrupt changes, especially on the floor of the scene.

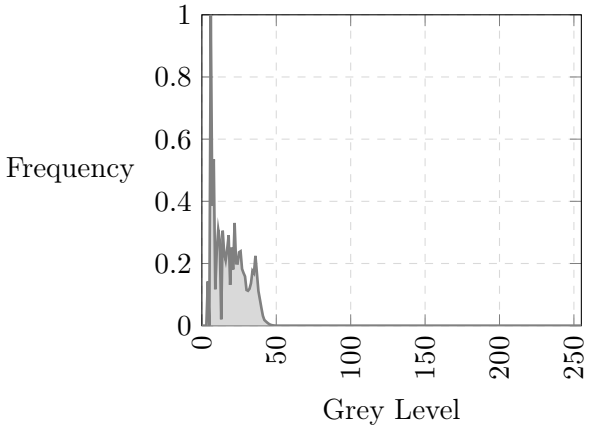


(a) Greyscale histogram of an image where histogram equalisation has been performed on the Y channel in the YCrCb colourspace. Notice the choppiness of the grey levels.



(b) Rendered image where histogram equalisation has been performed individually on the Y channel in the YCrCb colourspace. Notice the abrupt changes in colours on the floor of the scene.

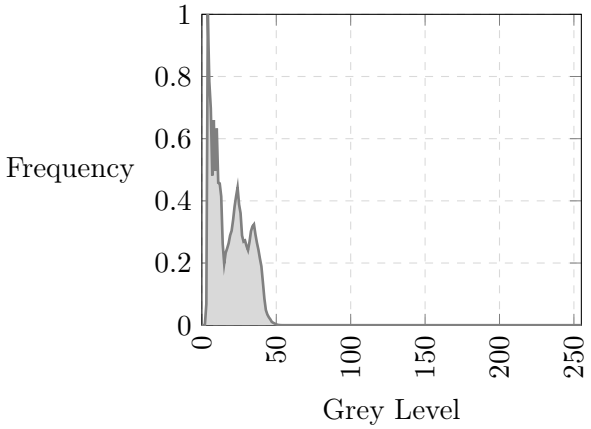
Figure 9.3.: Grey scale histogram in figure 9.3a of the image in figure 9.3b. Histogram equalisation has been performed on the Y channel in the YCrCb colourspace. This image has better exposure and looks decent at first glance. However, looking at the greyscale histogram, one notices a choppiness in the grey levels. At closer inspection of the rendered image, we can spot these abrupt changes, especially on the floor of the scene. It is similar to figure 9.2.



(a) Greyscale histogram of an image where CLAHE has been applied to the L channel in the CIELAB colourspace.

(b) Rendered image where CLAHE has been performed on the L channel in the CIELAB colourspace.

Figure 9.4.: Grey scale histogram in figure 9.4a of the image in figure 9.4b. CLAHE has been applied to the L channel in the CIELAB colourspace.



(a) Greyscale histogram of an image where CLAHE has been applied to the Y channel in the YCrCb colourspace.

(b) Rendered image where CLAHE has been applied to the Y channel in the YCrCb colourspace.

Figure 9.5.: Grey scale histogram in figure 9.5a of the image in figure 9.5b. CLAHE has been applied to the Y channel in the YCrCb colourspace.

Part II.

Experiments with Detectors on Generated Dataset

10. Experiments with Generated Dataset

10.1. Evaluation Metrics for Machine Learning Tasks

Machine learning models are mathematical functions that make predictions based on input data. To understand how well these models perform, measuring their performance using various metrics, collectively called evaluation metrics, is essential. Evaluation metrics in computer vision vary based on the specific task at hand. Tasks include image classification, object detection, semantic segmentation, and instance segmentation. Each of these tasks has its unique set of metrics to evaluate the performance of the models.

Precision and recall are two measures of a detector's performance originating from the Information Retrieval field. They offer a more nuanced assessment than merely considering the proportion of correct predictions (i.e., accuracy) as they consider the types of errors made. Precision attempts to answer the question of what proportion of identifications that were identified as correct actually were correct. Recall aims to determine the percentage of true positive cases that were correctly identified. These metrics originate from the confusion matrix illustrated in table 10.1. Precision and recall are given in equation 10.1, and 10.2.

		Predicted Condition	
		Positive	Negative
Actual Condition	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Table 10.1.: Confusion Matrix

$$Precision = \frac{TP}{TP + FP} \tag{10.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{10.2}$$

The F1 score, also known as the harmonic mean of precision and recall, is a metric that combines precision and recall. The benefit of the F1 score is that it provides a single scalar metric that balances the trade-off between precision and recall. It is defined as:

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{10.3}$$

In object detection, the goal is to classify and locate objects within an image. A common evaluation metric for this task is Average Precision (AP) and its variant Mean Average Precision (mAP).

Portions of this chapter have been copied from Preparatory Project: REDHUS Crack Detection, Ole Martin Ingebo, December 2022, and TDT4265 Assignment 4, Ole Martin Ingebo, March 2022.

These metrics consider the classification accuracy and the Intersection over Union (IoU) between the predicted and ground truth bounding boxes.

Intersection over union (IoU) is a measure of overlap between two bounding boxes. It calculates the area of overlap between the predicted bounding box and the ground truth bounding box, divided by the two boxes' union area. This metric is often used as a threshold to define whether a prediction is considered a match, usually at a 0.5 IoU threshold. To find the IoU, we use the formula:

$$IoU = \frac{|B_p \cap B_{gt}|}{|B_p \cup B_{gt}|} \tag{10.4}$$

where $B_p = (x_p, y_p, w_p, h_p)$ is the predicted bounding box, and $B_{gt} = (x_{gt}, y_{gt}, w_{gt}, h_{gt})$ is the ground truth box.

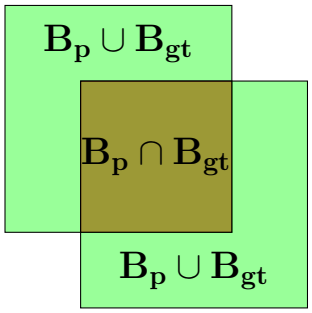


Figure 10.1.: Illustration of intersection over union. **add more**

Mean average precision@0.5 (mAP@0.5) is a common metric for evaluating the performance of object detection models that utilise the IoU. The @0.5 means this metric is calculated using an IoU threshold of 0.5. If the IoU for a prediction is greater than or equal to 0.5, the prediction is considered a true positive; otherwise, it is considered a false positive. The mAP@0.5 gives a single-number summary of the model's performance, averaging the precision of the model across all recall levels.

A metric that extends mAP@0.5 is mAP0.5:0.95. It is a more stringent variant of mAP@0.5 and is used in challenges such as the COCO object detection challenge. Instead of calculating the mean average precision at a single IoU threshold as 0.5, it calculates the mAP at multiple IoU thresholds from 0.5 to 0.95 with a step size of 0.05 and then takes the average. This provides a more holistic view of the model's performance, as the model must perform well across a range of IoU thresholds to achieve a high mAP@0.5:0.95 score.

10.2. Descriptions of Datasets used in Experiments

Description of DNV Dataset
The images in the dataset were sourced from surveyors at DNV. In total, 1.4 million images were collected. The images were taken with the intent of documenting surveys and not for computer vision-based tasks. Observing images with various types of markings and labelling is not uncommon. The images, for example, contain markings on the surrounding structures to mark where a crack is. This is performed with, for example, paint or chalk. Another type of labelling is performed digitally

Portions of this chapter have been copied from Preparatory Project: REDHUS Crack Detection, Ole Martin Ingebo, December 2022.

with an image editor program. What has been performed here varies. In some images, points of interest are marked with, i.e. a red circle, while others have regions of interest that are zoomed in on. Other images contain an object to assist with determining the scale of, i.e. a crack, just from the picture. This can, for example, be a ruler, pen, or hammer.

To a significant extent, many non-crack images contain more complex, varying scenes and backgrounds. This is, i.e. images of larger objects, such as an entire ship, and equipment from process engineering, such as valves, pumps and plumbing. Occasionally a city skyline can be observed in the background. The images that do contain cracks are often taken closer to the object or structure that has a crack, and there is more often uniform illumination. These images contain, to a greater extent, only the ship's structure where the crack appears and no other objects.

In some cases, surveyors have removed material to inspect the cracks further. Material removal is performed, i.e. with an angle grinder or hammer. The area where the angle grinder has been applied then changes various characteristics. This is, i.e. different texture and seeing the raw metal covered by corrosion. The scene may become brighter and have a greater degree of reflection.

There is likely an illumination difference in the drone footage compared to the images surveyors have captured. However, they share that the illumination source, i.e., a flash, is most likely rotating and translating with the camera. Examples of, i.e. the flash not being attached to the camera have not been observed. Even though the images share this commonality of illumination, there is a question of whether there are other impacts of this.

The image resolution of the observed images varies greatly. The smallest image has the dimensions 160×120, while the largest has the dimensions 4608×3456. The images that are currently available are all in JPEG format. A density heatmap of image resolutions in the DNV dataset can be seen in figure 10.2. The top five resolutions by count can be seen in table 10.2.

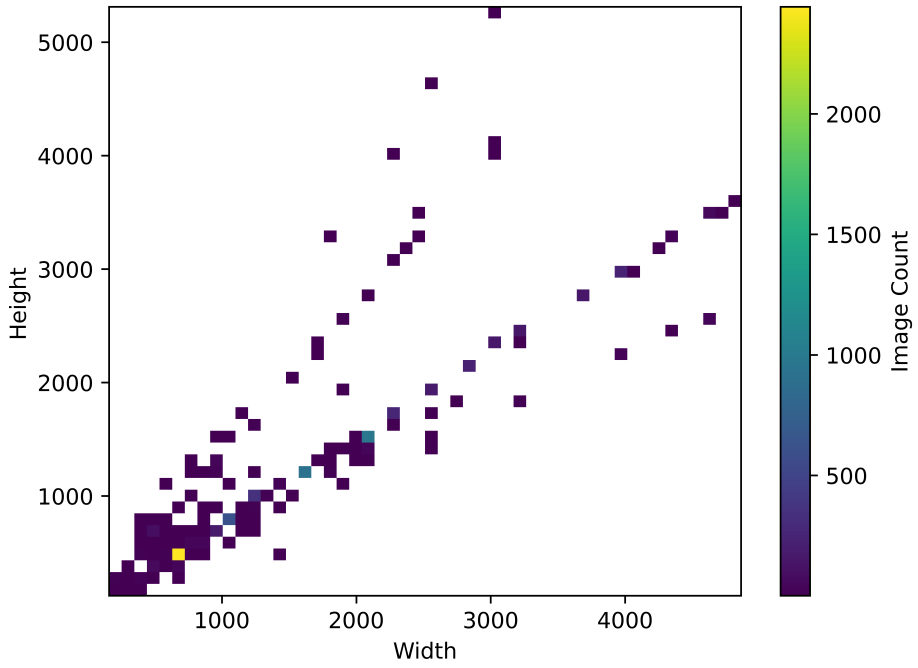


Figure 10.2.: Density heatmap of image resolutions in the DNV dataset.

Resolution	Count	Percentage of Dataset
640×480	2415	31.1%
2048×1536	897	11.5%
1600×1200	844	10.9%
1024×768	611	7.9%
1280×960	342	4.4%
Other	2656	34.2%

Table 10.2.: Top five image resolutions by their count in the DNV dataset.

The dataset currently has 3927 images labelled with cracks and 3838 without cracks. Resulting in a total of 7765 labelled images. The labels that exist or can be easily created are for image classification, patch-based image classification, object detection and semantic segmentation. The two classes for semantic segmentation are one for cracks and one for ambiguous areas. The reasoning behind the ambiguous class is that it can be ignored during training and evaluation for semantic models. An ambiguous area can vary, but it can, i.e. be areas surrounding cracks where it is difficult to assess whether it shall be deemed a crack. Ambiguous areas where corrosion has taken over and eroded material, creating more significant gaps and holes in the construction. Other examples include welding seams, holes that occur intentionally by construction, and areas with more significant corrosion. A few images from this dataset can be seen in section 11.2.

Description of Generated Dataset
A synthetic dataset was generated to evaluate whether the implemented pipeline could generate synthetic data that accurately emulated the attributes seen in real-world images. The composition of these renders closely matches the examples presented in Chapter 8, showcasing a consistent application of the pipeline.

This synthetic dataset is a foundational component of the experimental setup, which aims to assess how well the artificially created data mimics the characteristic features inherent to real images. The idea is to generate data and create a rich, realistic and representative data set that can be used effectively in machine learning models. By creating a synthetic dataset that parallels real-world images in terms of features, the hope is to expand the training material available for these models without labour-intensive and time-consuming data collection processes. A description of the synthetic dataset utilised for the experiments is provided in table 10.3.

Image Resolution	640×360
Image Count	10000
Image Count with Crack/No Crack	5000/5000

Table 10.3.: Synthetic dataset utilised for experiments.

10.3. Experimental Plan

After its creation, evaluating the authenticity and the representative capacity of a synthetically generated dataset is critical. In the context of this work, the aim was to emulate real-world tank cracks. The efficacy of the synthetic dataset in representing these real-world occurrences is tested against an actual dataset of tank crack images via an experiment.

This experiment compares evaluation metrics, as discussed in section 10.1, derived from the detector's predictions on both the real-world and synthetic datasets. If the detector displays high

performance on both sets, it validates the synthetic dataset as an accurate representation of the real-world data. However, should the detector's performance significantly decrease when applied to the real-world dataset compared to the synthetic dataset, it may suggest a need for more authenticity in its representation of tank cracks.

To ascertain the validity of the synthetic dataset, an experiment was designed to be carried out in three iterations. The initial step involves training a detector solely on DNV data. Subsequently, synthetic data is added to the training split of the DNV dataset, thus creating a mixed dataset on which the detector is trained once more. The dataset's identical validation and test splits, composed exclusively of DNV data, are then used to test the two trained detectors.

Ideally, a type of cross-validation would be implemented, practical constraints led to the decision to sample the DNV three times during the splitting process randomly. The experiment is repeated thrice to yield more robust and representative results.

10.4. Experimental Setup

Hardware, Software and Detector Used in Experiments
The experimental platform for the three conducted experiments comprised two Azure NCv3 instances provided by DNV. Each instance was outfitted with six vCPUs, a memory capacity of 112 GiB, and an NVIDIA Tesla V100 GPU with 16 GiB of memory. For software, Python 3.11 and PyTorch 2.0.1 were utilised.

The detector framework was designed to remain static and consistent throughout all experiments. This approach aimed not to compare different detectors but to assess the selected detector's performance across varying mixtures of the DNV and generated datasets. Consequently, the YOLOv5m6 [37] detector was chosen as recommended by DNV. This detector had exhibited high effectiveness in the past, along with a favourable balance between its performance metrics and the parameter count.

The training process was implemented with a batch size of 16 and an image resolution of $640px \times 640px$. All detectors underwent training for 300 epochs using the SGD (stochastic gradient descent) optimiser. However, some concluded prematurely due to early stopping. During the training phase, the backbone of the YOLOv5m6 detector remained frozen. This meant that only the detector's head was involved in learning, as per the recommendation of DNV. For detailed information regarding the YOLOv5m6 parameters used during training, please refer to table 10.4. Similarly, the parameters established for data augmentation during the training process are enumerated in table 10.5.

In total, six datasets were created. Initially, three `DNV` datasets were established, randomly splitting the DNV dataset into different training, validation, and testing subsets combinations. Then, an additional 10000 identical synthetic images were included in each `Train` split of the three datasets, creating three additional `Mixed` datasets. All experiments utilised datasets divided into training, validation, and testing subsets, adhering to a stratified split with an approximate ratio of $80:10:10$ *prior* to adding the generated data. Detailed descriptions of the datasets are found in tables 10.6, 10.7, 10.8, 10.9, 10.10, and 10.11. Note that `Crack/No Crack` does not indicate the total crack annotations but images that contain at least a single crack annotation, and that no synthetic data were added to the `Validation`, and `Test` split during training, and testing.

Parameter	Parameter Description	Parameter Value
lr0	Initial learning rate	0.01
lrf	Final OneCycleLR learning rate (lr0 * lrf)	0.1
momentum	SGD momentum	0.937
weight_decay	Optimizer weight decay	4e-4
warmup_epochs	Warmup Epochs	3.0
warmup_momentum	Warmup initial momentum	0.8
warmup_bias_lr	Warmup initial bias lr	0.1
box	Box loss gain	0.05
cls	Class loss gain	0.3
cls_pw	Class BCELoss positive weight	1.0
obj	Object loss gain (scale with pixels)	0.7
obj_pw	Object BCELoss positive weight	1.0
iou_t	IoU training threshold	0.20
anchor_t	Anchor-multiple threshold	4.0
fl_gamma	Focal loss amma	0.0

Table 10.4.: YOLOv5m6 training parameters.

Parameter	Parameter Description	Parameter Value
hsv_h	Image HSV-Hue augmentation (fraction)	0.015
hsv_s	Image HSV-Saturation augmentation (fraction)	0.7
hsv_v	Image HSV-Value augmentation (fraction)	0.4
degrees	Image rotation	0.0
translate	Image translation	0.1
scale	Image scale	0.9
shear	Image shear	0.0
perspective	Image perspective	0.0
flipud	Image flip up-down (probability)	0.0
fliplr	Image flip left-right (probability)	0.5
mosaic	Image mosaic (probability)	1.0
mixup	Image mixup (probability)	0.1
copy_paste	Segment copy-paste (probability)	0.1

Table 10.5.: YOLOv5m6 augmentation parameters.

Experiment 1 Datasets

	Split						Total
	Train		Validation		Test		
Dataset	DNV	Gen.	DNV	Gen.	DNV	Gen.	
Image Count	6121	0	845	0	799	0	7765
Crack/No Crack	3075/3046	0	440/405	0	412/387	0	3927/3838

Table 10.6.: Experiment 1. DNV Dataset.

	Split						Total
	Train		Validation		Test		
Dataset	DNV	Gen.	DNV	Gen.	DNV	Gen.	
Image Count	6121	10000	845	0	799	0	17765
Crack/No Crack	3075/3046	5000/5000	440/405	0	412/387	0	8927/8838

Table 10.7.: Experiment 1. Mixed Dataset.

Experiment 2 Datasets

	Split						Total
	Train		Validation		Test		
Dataset	DNV	Gen.	DNV	Gen.	DNV	Gen.	
Image Count	6119	0	947	0	699	0	7765
Crack/No Crack	3124/2995	0	454/493	0	349/350	0	3927/3838

Table 10.8.: Experiment 2. DNV Dataset.

	Split						Total
	Train		Validation		Test		
Dataset	DNV	Gen.	DNV	Gen.	DNV	Gen.	
Image Count	6119	10000	947	0	699	0	17765
Crack/No Crack	3124/2995	5000/5000	454/493	0	349/350	0	8927/8838

Table 10.9.: Experiment 2. Mixed dataset.

Experiment 3 Datasets

	Split						Total
	Train		Validation		Test		
Dataset	DNV	Gen.	DNV	Gen.	DNV	Gen.	
Image Count	6109	0	898	0	758	0	7765
Crack/No Crack	3099/3010	0	436/462	0	392/366	0	3927/3838

Table 10.10.: Experiment 3. DNV Dataset.

	Split						Total
	Train		Validation		Test		
Dataset	DNV	Gen.	DNV	Gen.	DNV	Gen.	
Image Count	6109	10000	898	0	758	0	17765
Crack/No Crack	3099/3010	5000/5000	436/462	0	392/366	0	8927/8838

Table 10.11.: Experiment 3. Mixed dataset.

10.5. Experimental Results

This section presents a curated collection of findings derived from the undertaken experiments. Additional plots produced during these experiments are available in appendix B.

Experiment 1 Results
See appendix B.2 for plots.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.359	0.308	0.332	0.247	0.103
Mixed	0.355	0.311	0.332	0.25	0.105

Table 10.12.: Experiment 1. `Dataset` refers to the dataset the detector was trained on. `DNV` refers to the dataset in table 10.6, `Mixed` refers to dataset in table 10.7. Both validated on the identical `Validation` split in the aforementioned tables.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.464	0.316	0.376	0.277	0.113
Mixed	0.393	0.362	0.377	0.284	0.123

Table 10.13.: Experiment 1. `Dataset` column refers to the dataset the detector was trained on. `DNV` refers to the dataset in table 10.6, `Mixed` refers to dataset in table 10.7. Both validated on the identical `Test` split in the aforementioned tables.

Experiment 2 Results
See appendix B.3 for plots.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.371	0.291	0.326	0.233	0.0999
Mixed	0.310	0.335	0.322	0.237	0.0983

Table 10.14.: Experiment 2. `Dataset` column refers to the dataset the detector was trained on. `DNV` refers to the dataset in table 10.8, `Mixed` refers to dataset in table 10.9. Both validated on the identical `Validation` split in the aforementioned tables.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.411	0.332	0.367	0.297	0.128
Mixed	0.388	0.344	0.365	0.289	0.134

Table 10.15.: Experiment 2. `Dataset` column refers to the dataset the detector was trained on. `DNV` refers to the dataset in table 10.8, `Mixed` refers to dataset in table 10.9. Both validated on the identical `Test` split in the aforementioned tables.

Experiment 3 Results
See appendix B.4 for plots.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.438	0.331	0.377	0.260	0.107
Mixed	0.343	0.342	0.342	0.235	0.107

Table 10.16.: Experiment 3. `Dataset` column refers to the dataset the detector was trained on. `DNV` refers to the dataset in table 10.10, `Mixed` refers to dataset in table 10.11. Both validated on the identical `Validation` split in the aforementioned tables.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.443	0.270	0.336	0.253	0.0944
Mixed	0.420	0.278	0.335	0.253	0.104

Table 10.17.: Experiment 3. `Dataset` column refers to the dataset the detector was trained on. `DNV` refers to the dataset in table 10.10, `Mixed` refers to dataset in table 10.11. Both validated on the identical `Validation` split in the aforementioned tables.

Aggregated Experimental Outcomes

In this section, the average outcomes obtained from the three experiments conducted are presented. The process of averaging was performed distinctly for the results validated on two different splits, the `Validation` split and the `Test` split.

The results obtained from each individual experiment exhibit variability due to the randomness inherent in the splitting of datasets and the training process. Averaging the results across multiple runs helps account for this variability and provides a more stable estimate of performance.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.389	0.310	0.345	0.247	0.103
Mixed	0.336	0.329	0.332	0.241	0.103

Table 10.18.: Averaged results from experiments over the `Validation` splits.

Dataset	Precision	Recall	F1	mAP@0.5	mAP@0.5:0.95
DNV	0.439	0.306	0.360	0.276	0.112
Mixed	0.400	0.328	0.359	0.275	0.120

Table 10.19.: Averaged results from experiments over the `Test` splits.

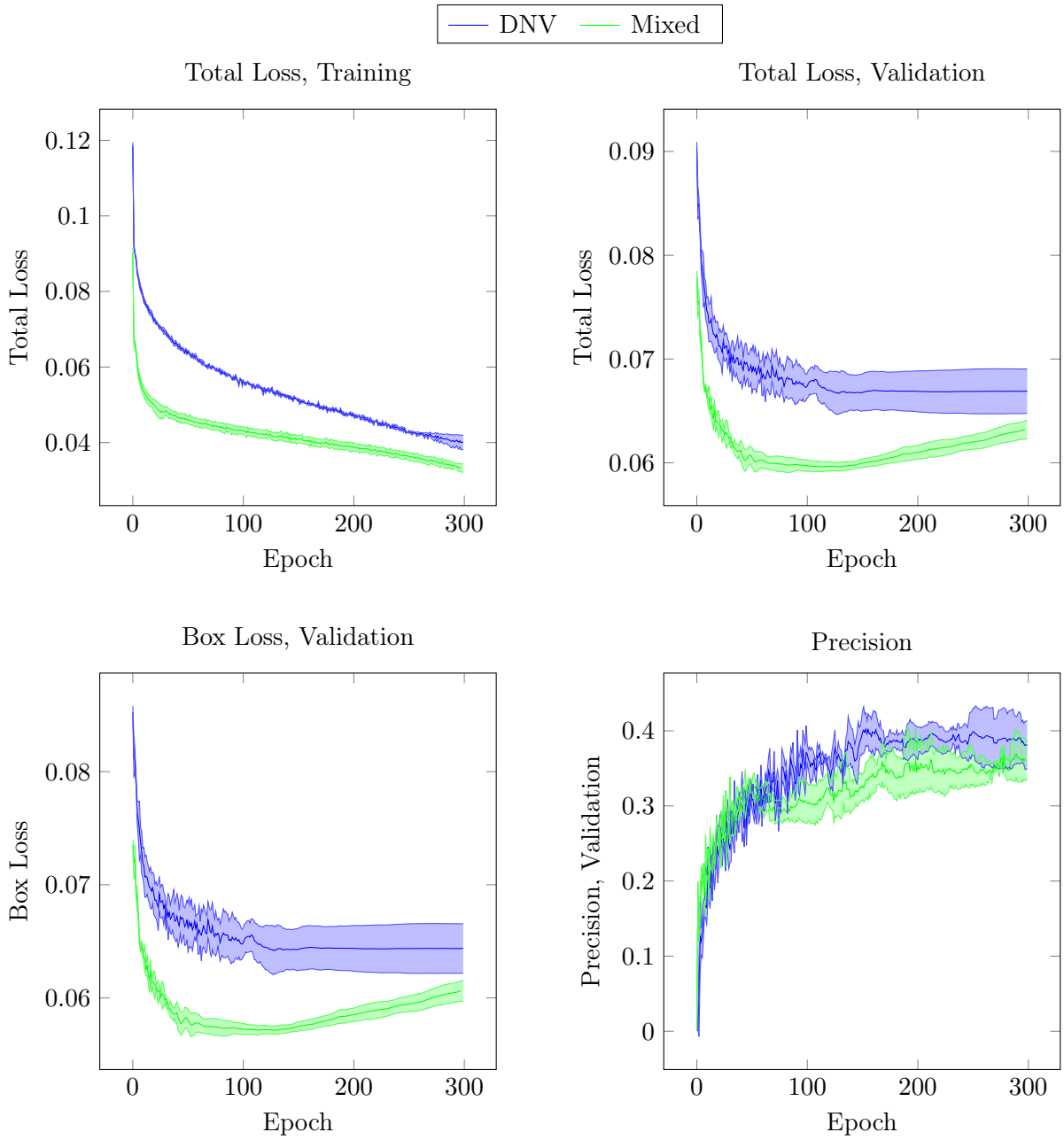


Figure 10.3.: Relevant averaged experiment plots. The upper and lower bands are each a single standard deviation from the mean. See all plots from the averaged and individual experiments in appendix B.

11. Evaluation of the Achieved Results

11.1. Evaluation of Generation of Synthetic Image Dataset

The first goal of this thesis encompassed the construction of a *synthetic data generation pipeline for cracks on ship surfaces*. This objective was systematically deconstructed into specific prerequisites outlined in Table 1.1. Each of these requirements is further elucidated in the ensuing discussions.

R.1. The images produced by the process shall be photorealistic.
An examination of sample images generated by the pipeline, detailed in Section 8, reveals instances where the output demonstrates a high degree of realism, notably in figures 8.3 and 8.36. Nevertheless, there are instances where an element of artificiality permeates the images, as shown in figures 8.27 and 8.53. The discrepancy could be attributed to the scene's texture, material, and context.

The synthetic crack typically possesses a photorealistic appearance, notwithstanding some issues with colouration, as observed in figure 8.41. A significant limitation of the scene, and the pipeline in general, is its inability to convey the scale of the rendered objects. The only scale reference available to an observer is the plane dimensions, set at $10m \times 10m$, and the crack width. Beyond these markers, accurately assessing the scale of the rendered scene presents considerable challenges.

R.2. The geometric models used in the scene shall accurately represent the geometry of the objects they represent.
This requirement has been partially met. The welding seams are well-rendered and considered high quality, as illustrated in figure 3.3. Thus, they would be low on the priority list for pipeline enhancements. What falls short in satisfying this criterion, however, is the overarching structure of the scene, the cube. This oversimplification fails to encapsulate the geometric complexity of a ship tank sufficiently. This limitation was recognised during the development phase of the pipeline, as discussed in section 9.1.

R.3 The textures and materials used on the geometric models shall be high resolution and accurately represent the objects' surface details.
As for the PBR texture and material reviewed in sections 4.2 and 8.1, they are considered to be of sufficient resolution, having been gathered at 4096×4096 pixels. Determining their accuracy in depicting the surface details of the corresponding objects is a more challenging matter. On analysing images from the DNV dataset, it appears that in some cases, they serve as accurate representations, while in many instances, they fall short. The natural complexity of ship tank surfaces often exhibits patterns, such as the pattern formed when a liquid remains at a consistent level within a tank for some time, that is not replicated in the synthetic images. Consequently, the surfaces often present as overly pristine and homogeneous.

Regarding the stained metal textures highlighted in sections 4.3 and 8.2, their resolution is not a component of this evaluation as they are procedurally generated. Their accuracy in replicating surface details bears similarities to the PBR textures. While some examples accurately mirror reality, others do not. Nevertheless, they tend to represent the most photorealistic ship surfaces due to the inclusion of realistic stains that align with those found in the DNV dataset.

The painted metal texture discussed in sections 4.4 and 8.3 is assessed as being of lower quality than the stained metal texture. It is challenging to pinpoint the exact factors contributing to its lower quality. However, one identifiable element is the sudden colour transitions within the texture, as seen in figure 8.47. Altough, there are instances where the colours integrate more seamlessly, as depicted in figure 8.56.

R.4. The geometric models of the cracks shall have realistic dimensions. This includes the width, depth and overall shape of the crack.
The general geometry of the synthetic cracks is assessed as being of high quality. However, one notable limitation is that the crack length cannot be controlled, as it extends across the entirety of the plane. Noteworthy examples of well-rendered cracks are figures 8.35, 8.37, 8.38, where both the geometry and colours work together to create an impression of photorealistic cracks. Nevertheless, there are scenarios such as in figures 8.53 and 8.54 where, despite good geometric representation, the colour bleed emanating from the cracks is unrealistic.

R.5. The rendering engine used shall be capable of accurately simulating the behaviour of light, shadows, and reflections in the scene.
The assessment concludes that Cycles' rendering engine can accurately simulate these characteristics. However, it is essential to note that Cycles cannot calculate physical light properties from a radiometric or photometric standpoint. As detailed in section 6.2, light settings eventually had to be configured from an artistic perspective rather than a purely scientific one.

R.6. The lighting used in the scene shall simulate the lighting on the REDHUS drone.
Determining if the lighting employed in the pipeline accurately emulates the lighting associated with the REDHUS drone presents a challenge due to the limited footage available from the drone. As previously indicated, the lighting parameters ultimately had to be determined through an artistic lens, further complicating a direct comparison.

R.7. The camera settings should accurately simulate the REDHUS camera.
The camera parameters used in the REDHUS were replicated, as detailed in section 5.4. This accurate representation of settings ensures that the virtual camera used in the pipeline closely mirrors the physical REDHUS camera in terms of optical and mechanical properties such as focal length, sensor size, and resolution. As a result, the pipeline can simulate the unique visual perspective and inherent limitations of the REDHUS camera, effectively fulfilling this requirement.

R.8. The annotations shall be available in common formats, i.e. COCO.
The annotations generated in the pipeline have been made available in various universally recognised formats to ensure their broad usability and accessibility. Specifically, these include a segmentation image format, which offers a pixel-wise categorisation of the objects present in the image. Additionally, the annotations are available in the COCO (Common Objects in Context) format. It provides not only bounding box annotations but also segmentation annotations. By offering annotations in these formats, the pipeline fulfils this requirement.

Summary
The pipeline was evaluated against eight essential requirements, which addressed the photorealism of the images, the accuracy of the geometric models and textures, the realistic simulation of lighting, and the availability of annotations in standard formats.

The pipeline showed mixed results across the requirements. It produced both highly realistic and slightly artificial-looking images. While accurately depicting the welding seams, it simplified the overall scene geometry. It used high-resolution textures but struggled to depict complex patterns found on real ship tanks accurately.

The lighting simulation was accurate but relied on artistic configuration rather than scientific cal-

culation. Despite having limited drone footage, the pipeline could simulate the REDHUS drone's lighting conditions and accurately replicate the camera settings.

Finally, it successfully provided annotations in various universally recognised formats, including a segmentation image format and the COCO format. Overall, while there were areas of success, the evaluation identified several areas for improvement within the pipeline.

The requirements and whether they have been satisfied are listed in 11.1.

Requirement	Description	Satisfied
R.1	The images produced by the process shall be photorealistic.	Partially
R.2	The geometric models used in the scene shall accurately represent the geometry of the objects they represent.	No
R.3	The textures and materials used on the geometric models shall be high resolution and accurately represent the objects' surface details.	Partially
R.4	The geometric models of the cracks shall have realistic dimensions. This includes the width, depth and overall shape of the crack.	Yes
R.5	The rendering engine used shall be capable of accurately simulating the behaviour of light, shadows, and reflections in the scene.	Yes
R.6	The lighting used in the scene shall simulate the lighting on the REDHUS drone.	Yes
R.7	The camera settings should accurately simulate the REDHUS camera.	Yes
R.8	The annotations shall be available in common formats, i.e. COCO.	Yes

Table 11.1.: Requirements related to the synthetic data generation pipeline as in goal 1 and whether they are evaluated as satisfied.

11.2. Evaluation of Experiments with Detectors on Generated Dataset

The experimental results from section 10.5 first lead us through a quantitative evaluation and then a qualitative investigation.

Quantitative Evaluation
During the testing of the detectors, the detectors were only tested on DNV data.

Upon examining the results of the `Test` split found in table 10.19, it is apparent that a nuanced dynamic emerges. A trade-off between precision and recall is noticed. An increase in the detector's recall is counterbalanced by a decrease in precision. This observation is interesting as it reveals how the detector adjusts its behaviour when tackling diverse data. The `F1` score remains relatively steady despite this flux in precision and recall. The F1 score, being a harmonic mean of precision and recall, offers us a balanced measure of the test's accuracy, insinuating a consistent level of performance throughout the evaluation process.

When looking into the mean average precision values, it is observed that `mAP@0.5` remains almost constant, while `mAP@0.5:0.95` experiences an increase of over 7%. The change in mAP is coupled with a decrease in box loss, as shown in figure 10.3. This decrease points to the detector's enhanced

capability to accurately draw bounding boxes around a specific type of crack, emphasising its effectiveness on the problem at hand.

Furthermore, a noteworthy revelation comes from the `Total Loss, Validation`. This metric is dictated by the box loss as seen in `Box Loss, Validation`. This relationship between the two loss metrics underscores the box loss's critical role in the total validation loss and, by extension, the overall performance of the detector.

Proceeding to compare `Total Loss, Training` and `Total Loss, Validation` depicted in figure 10.3, there are clear indications of overfitting beginning roughly around `Epoch` 125. This trend of overfitting is evident from the continuous decrease in training loss in contrast with an increasing validation loss, implying that the model is starting to memorise the training data rather than learning to generalise from it. This phenomenon could be attributed to a surplus of synthetic data, creating an imbalance in the representation of real and synthetic images, thus affecting the model's ability to generalise and possibly leading to its poorer performance on unseen data. This insight calls for careful consideration of the ratio between synthetic and real data in the training set to optimise the detector's performance.

Qualitative Evaluation

The qualitative ramifications of the generated data on the detector's performance were studied using two distinct models, one trained solely on the DNV dataset and another on a mixed dataset of DNV and synthetic data. These models were tested on a shared `Test` split.

The comparative study illustrated through figures 11.1, 11.2, 11.3, and 11.4, shows that the detector trained on the mixed dataset exhibits greater confidence in its detections on thin cracks that occupy a more significant distance on the image. An interesting observation in figure 11.2 is the proximity of the mixed dataset detector's bounding box's right vertical line to the crack's edge, which is markedly closer than its counterpart from the DNV dataset, as shown in figure 11.1. This supports the lower `Box Loss, Validation` in figure 10.3.

Figure 11.1.: Detector trained with DNV dataset. Image from DNV dataset.

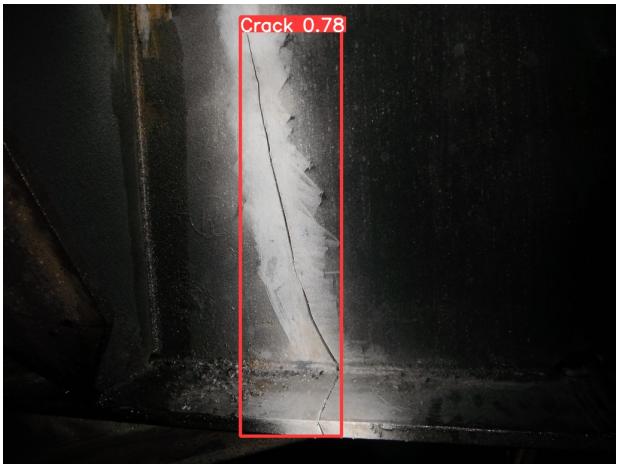


Figure 11.2.: Detector trained with Mixed Dataset. Image from DNV dataset.

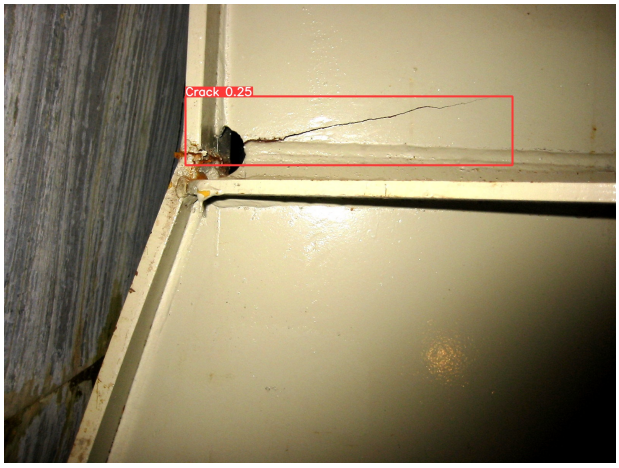


Figure 11.3.: Detector trained with DNV dataset. Image from DNV dataset.

Figure 11.4.: Detector trained with Mixed Dataset. Image from DNV dataset.

In contrast between figures 11.5 and 11.6, it is apparent that the mixed dataset detector successfully identifies the crack, a detail missed by the DNV dataset detector, suggesting an increased recall ability of the former.

Figure 11.5.: Detector trained with DNV dataset. Image from DNV dataset.

Figure 11.6.: Detector trained with Mixed Dataset. Image from DNV dataset.

However, it is crucial to acknowledge the instances where the detector lacks efficacy, which could indicate an imbalance in the datasets, particularly an overrepresentation of long, thin cracks. This pattern is discernible in figures 11.7 and 11.8, wherein the mixed detector displays reduced confidence when tasked with detecting wider, irregular cracks that do not extend over a large area of the image.

Figure 11.7.: Detector trained with DNV dataset. Image from DNV dataset.

Figure 11.8.: Detector trained with Mixed Dataset. Image from DNV dataset.

This issue becomes more noticeable in figures 11.9 and 11.10, where the mixed detector misclassifies a string as a crack, suggesting a need for refinement in detection.

Figure 11.9.: Detector trained with DNV dataset. Image from DNV dataset.

Figure 11.10.: Detector trained with Mixed Dataset. Image from DNV dataset.

Summary

Quantitative and qualitative analyses were conducted to evaluate experiments with the detector on the generated dataset. Quantitatively, a trade-off between precision and recall was observed, with an increase in recall but a slight decrease in precision. The `mAP@0.5:0.95` showed improved performance for the detector trained on the mixed dataset, indicating better bounding box regression accuracy. Overfitting was also identified, suggesting the need for a balanced dataset. Qualitatively, the mixed dataset-trained detector demonstrated higher confidence and more accurate detections for thin cracks. However, challenges were observed in detecting broader and more irregular cracks. Overall, including synthetic data improved the detector's performance, highlighting the importance of dataset balance and ongoing optimisation.

11.3. Perspectives for Future Work

The results presented in this thesis have shown that adding synthetic data to the training set, improved the results when detecting defects in actual images from ship tanks. Additional research is suggested to further realize the potential of adding synthetic data to the training set.

Development of photorealistic Materials from Ship Tank Images
Striving for a high level of realism in any synthetic dataset demands the faithful recreation of real-world parameters. A fundamental aspect of this process lies in creating accurate PBR (Physically-Based Rendering) materials, especially in scenarios where the correct depiction of surface properties can make a significant difference, like in the simulation of ship tanks. Therefore, future work could include developing PBR materials based on actual photographs of various ship tanks or similar surfaces. This could be achieved by employing methods such as bitmap approximations, multi-angle lighting, photogrammetry, and 3D photogrammetry [38, 39]. Creating these tailor-made materials would significantly improve the degree of realism and, hence, the quality of the synthetic data generated.

Production of Authentic 3D Models of Ship Tanks and Associated Objects
The use of simplified geometric shapes, such as the cube, in the current pipeline as primitive stand-ins for ship tanks inevitably leads to the loss of intricate geometric characteristics that real ship tanks possess. These lost features likely affect the detector's learning capacity, as they carry crucial information. Future studies could focus on creating, sourcing, or obtaining more authentic 3D models of ship tanks, possibly from DNV or other similar entities, to rectify this issue. These models could then be integrated into the existing pipeline, and the previously developed PBR materials could be applied to these models through UV mapping for added realism. Furthermore, to make the scene more representative of real-world scenarios, one should incorporate additional objects commonly found in the same environment, such as various industrial equipment.

Procedural Generation of Photorealistic 3D Scenes
As an alternative to the labour-intensive process of manually constructing 3D scenes, a plausible strategy could be to repurpose the methodologies described in Raistrick et al. [40]. This generator could be tailored to suit the unique requirements of environments such as ship tanks or other industrial settings. In the paper, the authors employed randomised mathematical algorithms to automatically generate all the components of the scene. This approach not only ensures efficiency but also introduces an element of randomness that more closely mimics real-world variability, potentially leading to a more realistic and comprehensive representation of a desired domain.

Restricting Crack Extent
The cracks in the simulation extend across the entire plane they occupy. A beneficial modification could involve imposing restrictions on the extent of these cracks. Implementing this could be challenging with the current approach, but initial investigations suggest a few promising methods. For instance, a displacement modifier could be applied on a selected vertex group or an additional, crack-free render of the image could be produced, with the cracks added in later during post-processing.

Improvement of Crack Geometry
Currently, cracks are displaced strictly along the surface normal of the plane, which might not accurately reflect their real-world behaviour. A potential improvement could be to explore methods that allow for a more realistic displacement of cracks. Blender's Vector Displacement Nodes could help achieve this.

Simulation of Cracks at Natural Locations
In the current pipeline, the positions of cracks are determined randomly, which likely do not accu-

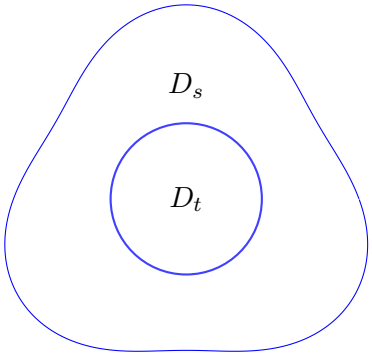


Figure 11.11.: Instead of attempting to align the source domain D_s with the target domain D_t as depicted in figure 1.3, a different approach is deployed. A source domain D_s is generated through randomisation, which is expansive enough to encompass the target domain D_t.

rately represent real-world scenarios. A logical next step would be to simulate the appearance of cracks at locations where they are likely to occur naturally. This could be done by defining regions in the scene where cracks could appear based on insights gathered from real-world data or experienced surveyors. Alternatively, one could perform stress calculations such as Von Mises stress using Finite Element Analysis to predict the locations of potential cracks.

Domain Randomisation

Synthetic data generation often involves a conscientious effort to mimic reality as closely as possible, as this thesis reflects. This endeavour necessitates meticulous attention to detail, stretching from the creation of realistic scenes and materials to the precise application of these materials and the authentic representation of cracks. The process resembles the intricate work of a 3D artist, meticulously sculpting every element to perfect the final rendering.

However, there is an alternate approach known as domain randomisation. This methodology acknowledges the inherent complexity of accurately replicating the real world and veers away from curated scenarios. As showcased in Tobin et al. [11], this strategy involves a more random selection of textures. Tobin et al. [11] use "a random RGB value", "a gradient between two random RGB values", and "a checker pattern between two random RGB values".

Adopting this perspective, one could amass a diverse array of materials, irrespective of their specific attributes, such as being metallic or not. The same indiscriminate approach is applied to the collection of objects and scenes, disregarding their fit within the conventional notion of an industrial look. These elements are then combined in myriad ways. Amidst these vast, random combinations, representations of the real world begin to emerge.

When the objective is to achieve photorealism, as depicted in figure 1.3, the ultimate goal becomes the perfect alignment of the source domain with the real world. Rather than striving to make the source domain mimic the target domain, the aim is to create a source domain so extensive and diverse that it encapsulates the target domain. The target domain is represented within this expansive and randomised source domain. This concept is visualised in figure 11.11.

12. Conclusion

The primary objective of this thesis was to explore the potential of synthetic data, specifically of cracks on ship surfaces, to enhance the performance of supervised learning detectors. The investigation entailed the development of a synthetic data generation pipeline and the subsequent training and evaluation of detectors based on the data produced.

In the pipeline, a geometrical scene was conceived, illuminated, and adorned with photorealistic textures and materials. Additionally, a method was implemented to produce photorealistic crack images from tanks and their corresponding labels. The effect of adding the photorealistic synthetic data to the training set was measured by evaluating the performance of the trained algorithm on actual crack images from ship tanks.

The results in this thesis show that synthetic data indeed contribute positively to the performance of detectors. However, it is essential to note that these improvements were most noticeable in detecting thin and uniform cracks. This observation underscores a critical insight that the capability of a detector is determined mainly by the variety and relevance of the data it is trained on. In other words, a detector can only improve at recognizing patterns that are well represented in its training data.

Therefore, it becomes clear that increasing the breadth and diversity of synthetic data is critical for achieving more comprehensive and adaptable detector performance. Future work should centre around diversifying synthetic data, which could involve generating various crack types based on their physical attributes and the conditions leading to their formation. Additional environmental variables such as lighting conditions, material types, and viewing angles could also be studied to build a more extensive and robust training dataset. A shift could also be considered, moving beyond a solely photorealistic methodology towards applying domain randomization.

Bibliography

[1] E. Stensrud, "Autonomous Drone Based Survey of Ships in Operation." [Online]. Available: https://prosjektbanken.forskningsradet.no/en/project/FORISS/282287

[2] G. Hamre, "REmote Drone-based ship HUll Survey." [Online]. Available: https://prosjektbanken.forskningsradet.no/en/project/FORISS/317773

[3] "Scout 137 Drone System." [Online]. Available: https://www.scoutdi.com/scout-137-drone-system/

[4] E. Stensrud, T. Skramstad, C. Cabos, G. Hamre, K. Klausen, B. Raeissi, J. Xie, and A. Ødegårdstuen, "Automating Inspections of Cargo and Ballast Tanks using Drones," in *COMPIT '19: 18th International Conference on Computer and IT Applications in the Maritime Industries*. Tullamore: Technische Universität Hamburg-Harburg, Mar. 2019, pp. 391–404.

[5] I. Abdel-Qader, O. Abudayyeh, and M. E. Kelly, "Analysis of Edge-Detection Techniques for Crack Identification in Bridges," *Journal of Computing in Civil Engineering*, vol. 17, no. 4, pp. 255–263, Oct. 2003. [Online]. Available: https://ascelibrary.org/doi/10.1061/%28ASCE%290887-3801%282003%2917%3A4%28255%29

[6] Y.-A. Hsieh and Y. J. Tsai, "Machine Learning for Crack Detection: Review and Model Performance Comparison," *Journal of Computing in Civil Engineering*, vol. 34, no. 5, p. 04020038, Sep. 2020. [Online]. Available: https://ascelibrary.org/doi/10.1061/%28ASCE%29CP.1943-5487.0000918

[7] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting Unreasonable Effectiveness of Data in Deep Learning Era," Aug. 2017. [Online]. Available: http://arxiv.org/abs/1707.02968

[8] R. Nevatia and T. O. Binford, "Description and recognition of curved objects," *Artificial Intelligence*, vol. 8, no. 1, pp. 77–98, 1977. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370277900066

[9] B. Sun and K. Saenko, "From Virtual to Reality: Fast Adaptation of Virtual Object Detectors to Real Domains," in *Proceedings of the British Machine Vision Conference 2014*. Nottingham: British Machine Vision Association, 2014, pp. 82.1–82.12. [Online]. Available: http://www.bmva.org/bmvc/2014/papers/paper062/index.html

[10] Y. Movshovitz-Attias, T. Kanade, and Y. Sheikh, "How useful is photo-realistic rendering for visual learning?" Sep. 2016. [Online]. Available: http://arxiv.org/abs/1603.08152

[11] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World," Mar. 2017. [Online]. Available: http://arxiv.org/abs/1703.06907

[12] Blender Online Community, *Blender - a 3D Modelling and Rendering Package, Version 3.3.0*, Blender Foundation, Blender Institute, Amsterdam, 2022. [Online]. Available: http://www.blender.org

[13] M. Denninger, M. Sundermeyer, D. Winkelbauer, D. Olefir, T. Hodan, Y. Zidan, M. Elbadrawy, M. Knauer, H. T. Katam, and A. Lodhi, "BlenderProc: Reducing the Reality Gap with Photorealistic Rendering," Jul. 2020.

[14] M. Denninger, D. Winkelbauer, M. Sundermeyer, W. Boerdijk, M. Knauer, K. H. Strobl, M. Humt, and R. Triebel, "BlenderProc2: A Procedural Pipeline for Photorealistic Rendering," *Journal of Open Source Software*, vol. 8, no. 82, p. 4901, Feb. 2023. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.04901

[15] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation: Third Edition*, third edition ed. Elsevier Science and Technology Books, Inc, 2016.

[16] E. Catmull and J. Clark, "Recursively generated B-spline surfaces on arbitrary topological meshes," *Computer-Aided Design*, vol. 10, no. 6, pp. 350–355, Nov. 1978. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0010448578901100

[17] R. L. Cook, "Shade trees," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 223–231, Jan. 1984. [Online]. Available: https://dl.acm.org/doi/10.1145/964965.808602

[18] G. , "How to create proper displacement for weld joints?" Sep. 2014. [Online]. Available: https://blender.stackexchange.com/a/15629

[19] Blender Made Easy, "Blender Tutorial - Procedural Cracked Earth Effect," Jun. 2021. [Online]. Available: https://www.youtube.com/watch?v=oYEaJxw4pSo

[20] J. T. Kajiya, "The rendering equation," *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 143–150, Aug. 1986. [Online]. Available: https://dl.acm.org/doi/10.1145/15886.15902

[21] K. E. Torrance and E. M. Sparrow, "Theory for off-specular reflection from roughened surfaces," *Journal of The Optical Society of America*, vol. 57, no. 9, pp. 1105–1114, Sep. 1967. [Online]. Available: https://opg.optica.org/abstract.cfm?URI=josa-57-9-1105

[22] R. L. Cook and K. E. Torrance, "A Reflectance Model for Computer Graphics," in *International Conference on Computer Graphics and Interactive Techniques*, 1981.

[23] "Cycles Open Source Production Rendering," 2022. [Online]. Available: https://www.cycles-renderer.org/features/

[24] B. Burley, "Physically Based Shading at Disney," 2012. [Online]. Available: https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf

[25] L. Demes, "ambientCG," 2023. [Online]. Available: https://ambientcg.com/

[26] H. Nguyen and N. Corporation, Eds., *GPU Gems 3*. Upper Saddle River, NJ: Addison-Wesley, 2008.

[27] J. Lampel, "Scattershot - Pbr Texture Bombing For Blender." [Online]. Available: https://blendermarket.com/products/scattershot---procedural-image-texture-scattering--tiling-with-voronoi

[28] R. King, "Procedural Dirty Metal (Blender Tutorial)." [Online]. Available: https://www.youtube.com/watch?v=uqfcV56SHMc

[29] ——, "Procedural Rusty Painted Metal (Blender Tutorial)." [Online]. Available: https://www.youtube.com/watch?v=waCeHg7yHMw

[30] M. Charity, "Blackbody color datafile (bbr_color.txt)," Jun. 2001. [Online]. Available: http://www.vendian.org/mncharity/dir3/blackbody/UnstableURLs/bbr_color.html

[31] CIE TC 2-93, "ISO/CIE 23539:2023 Photometry — The CIE system of physical photometry," International Commission on Illumination (CIE), Tech. Rep., Mar. 2023. [Online]. Available: https://cie.co.at/publications/photometry-cie-system-physical-photometry-3

[32] J. Lampel and B. Van Lommel, "Cycles, Unit of Light Energy," 30.3.20. [Online]. Available: https://devtalk.blender.org/t/cycles-unit-of-light-energy-attn-brecht/12456/5

[33] R. Szeliski, *Computer Vision Algorithms and Applications*, 2nd ed., ser. Texts in Computer Science. Springer Cham, 2022.

[34] B. Peterson, *Understanding Exposure: How to Shoot Great Photographs with Any Camera*, 3rd ed. New York: Amphoto Books, 2010.

[35] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda, "Photographic tone reproduction for digital images," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 267–276, Jul. 2002. [Online]. Available: https://dl.acm.org/doi/10.1145/566654.566575

[36] "Technical Introduction to OpenEXR." [Online]. Available: https://openexr.com/en/latest/TechnicalIntroduction.html

[37] G. Jocher, "YOLOv5 by Ultralytics," May 2020. [Online]. Available: https://github.com/ultralytics/yolov5

[38] L. Demes, "Creating PBR Materials," Jun. 2022. [Online]. Available: https://docs.ambientcg.com/books/creating-pbr-materials/chapter/general-workflows

[39] W. McDermott, "The PBR Guide," Feb. 2018. [Online]. Available: https://substance3d.adobe.com/tutorials/courses/the-pbr-guide-part-1

[40] A. Raistrick, L. Lipson, Z. Ma, L. Mei, M. Wang, Y. Zuo, K. Kayan, H. Wen, B. Han, Y. Wang, A. Newell, H. Law, A. Goyal, K. Yang, and J. Deng, "Infinite Photorealistic Worlds using Procedural Generation," Jun. 2023. [Online]. Available: http://arxiv.org/abs/2306.09310

A. Image Post Processing

A.1. Code

```
import cv2 as cv
import numpy as np

def post_process(img_path, gamma=2.2):
    # Read the image with .exr extension, and convert it to RGB.
    img = cv.imread(img_path, cv.IMREAD_COLOR | cv.IMREAD_ANYDEPTH)
    img = cv.cvtColor(img, cv.COLOR_BGR2RGB)

    # Create a tone mapper, then tone map the image.
    tonemapper = cv.createTonemapReinhard(gamma=gamma)
    img = tonemapper.process(img)

    # Convert the image from float32 that lie in the range 0-1, to 0-65535 to save
     as 16-bit PNG. Then convert the arrays data-type to uint16.
    img=img*65535
    img[img>65535]=65535
    img=np.uint16(img)

    # Save the image, with .png extension.
    cv.imwrite(save_path, img)
```

Listing A.1: Conversion of an image from linear color space to a color space more suitable for a monitor. Including conversion from OpenEXR (.exr) to Portable Network Graphics (.png).

B. Results

B.1. Averaged Results

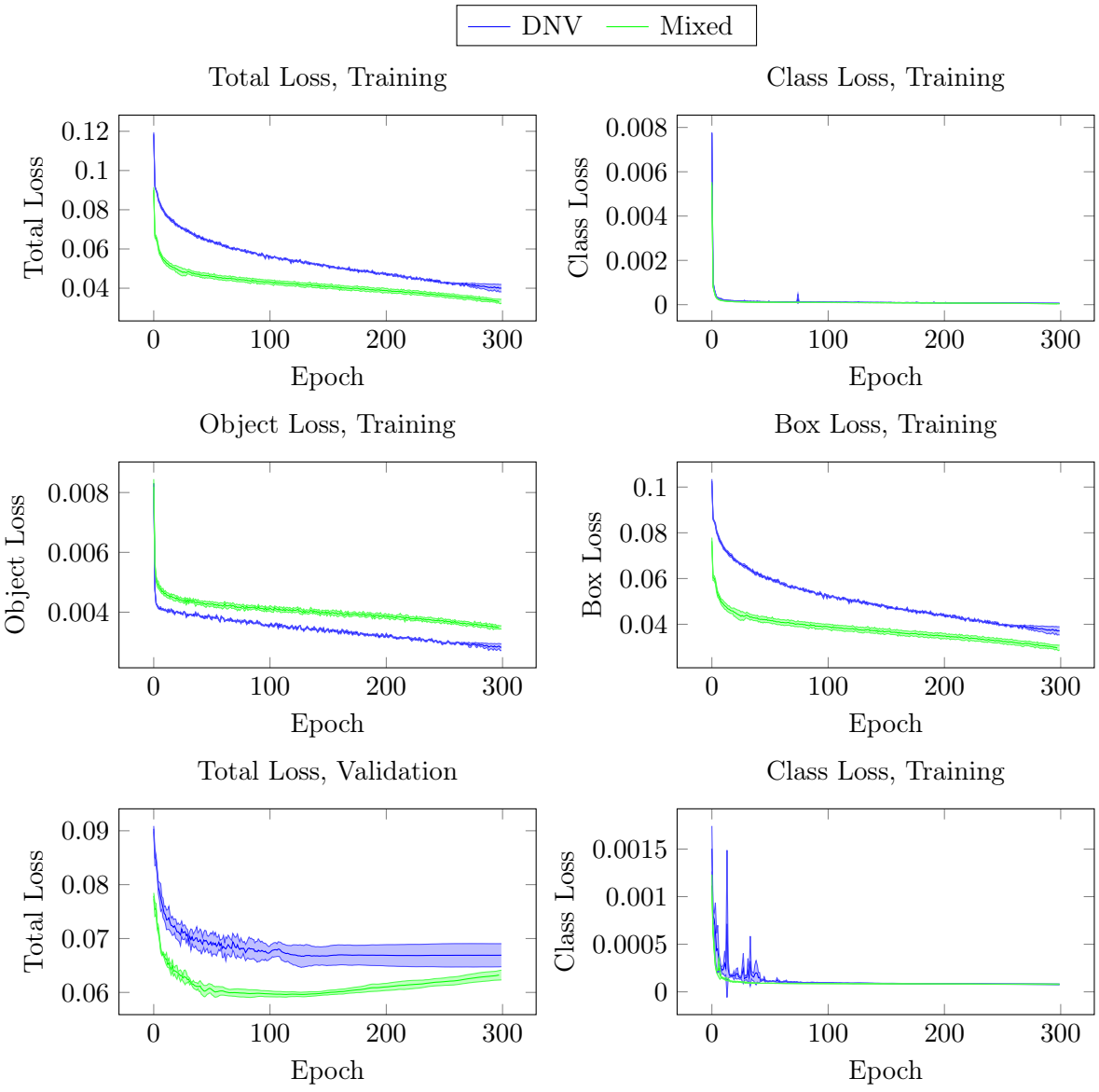


Figure B.1.: Plots of averaged training and validation results from experiments 1, 2, and 3. The upper and lower bands are each a single standard deviation from the mean.

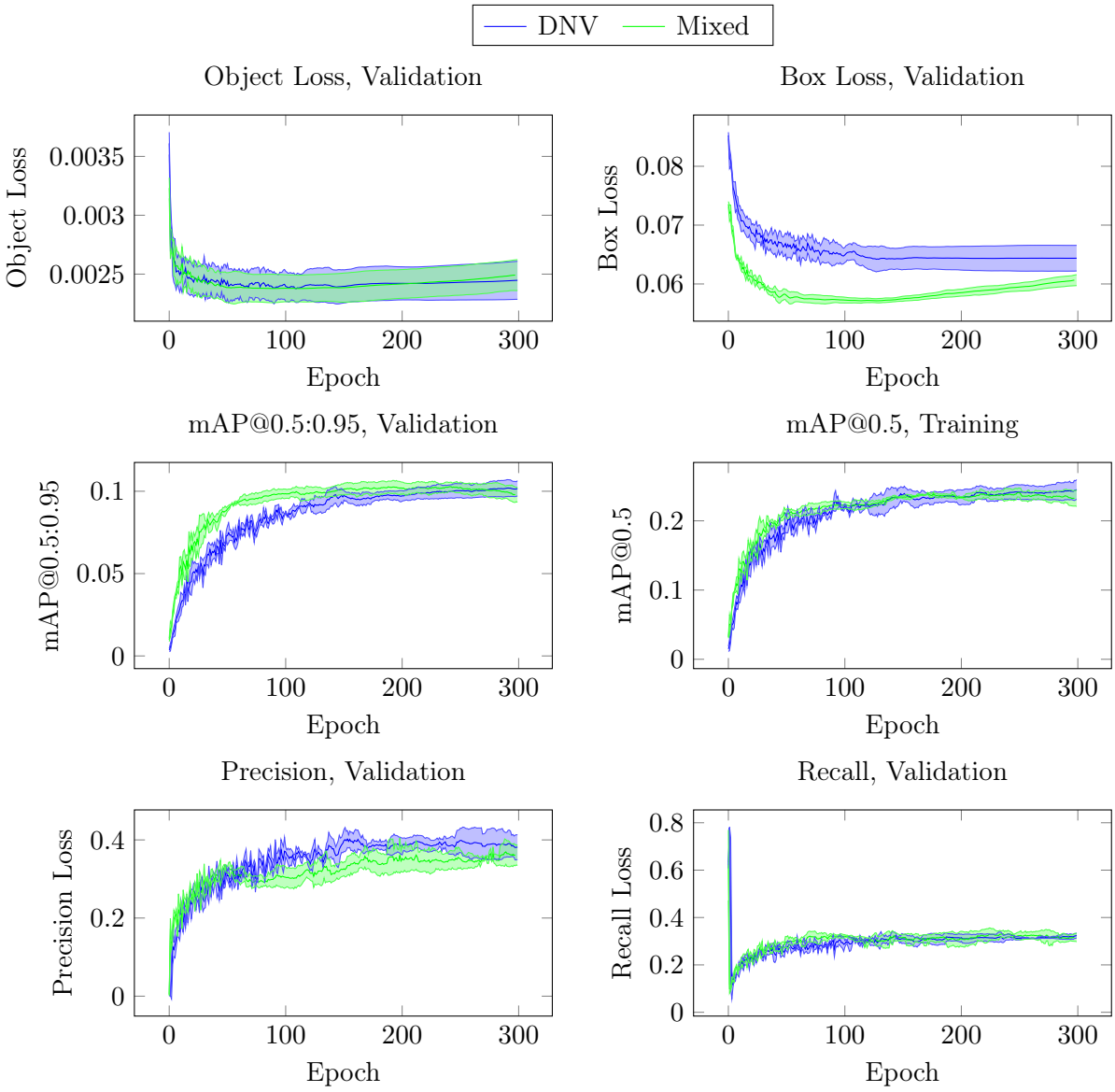


Figure B.2.: Plots of averaged training and validation results from experiments 1, 2, and 3. The upper and lower bands are each a single standard deviation from the mean.

B.2. Experiment 1 Results

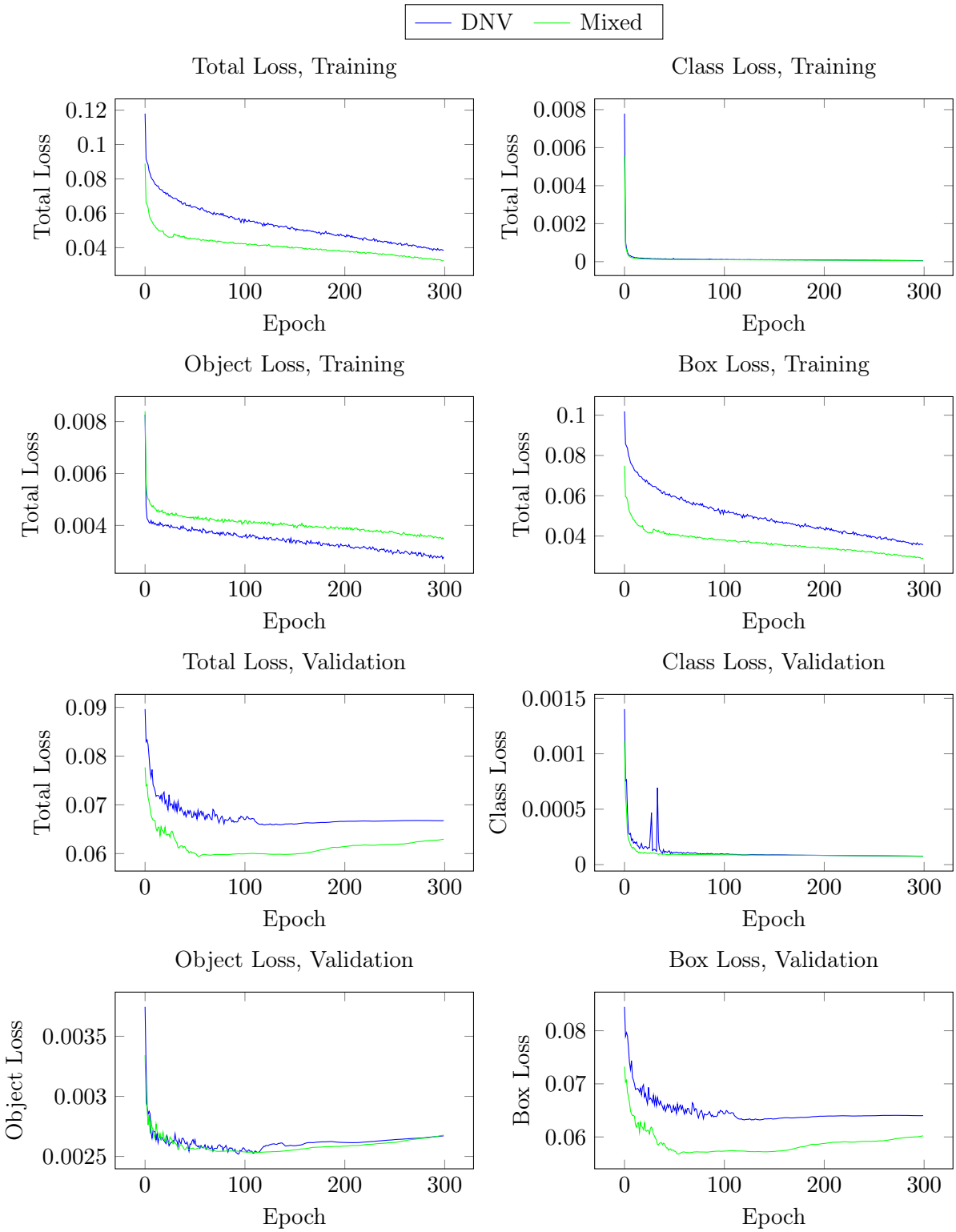


Figure B.3.: Plots of training and validation results from experiment 1.

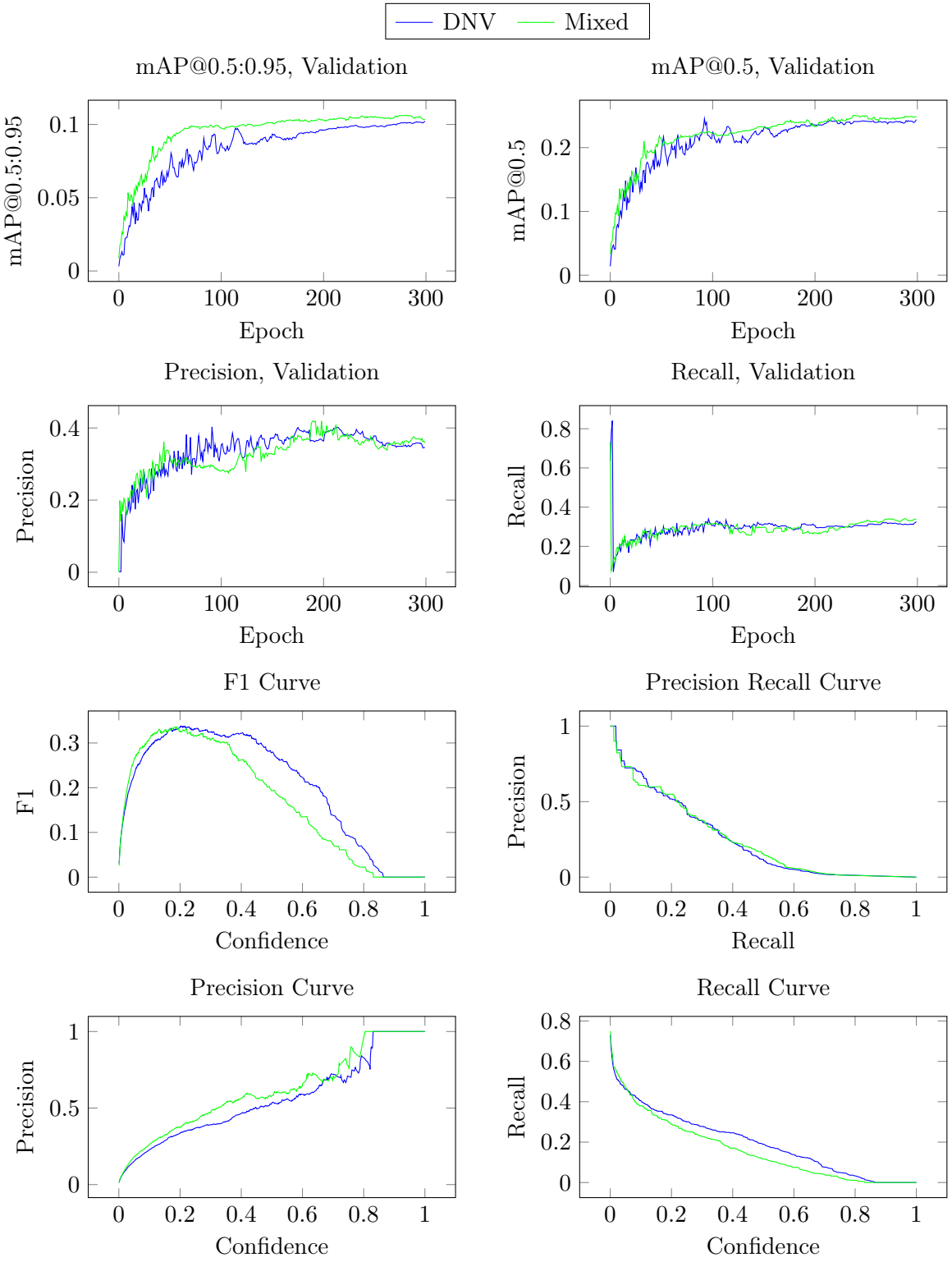


Figure B.4.: Plots of training and validation results from experiment 1.

B.3. Experiment 2 Results

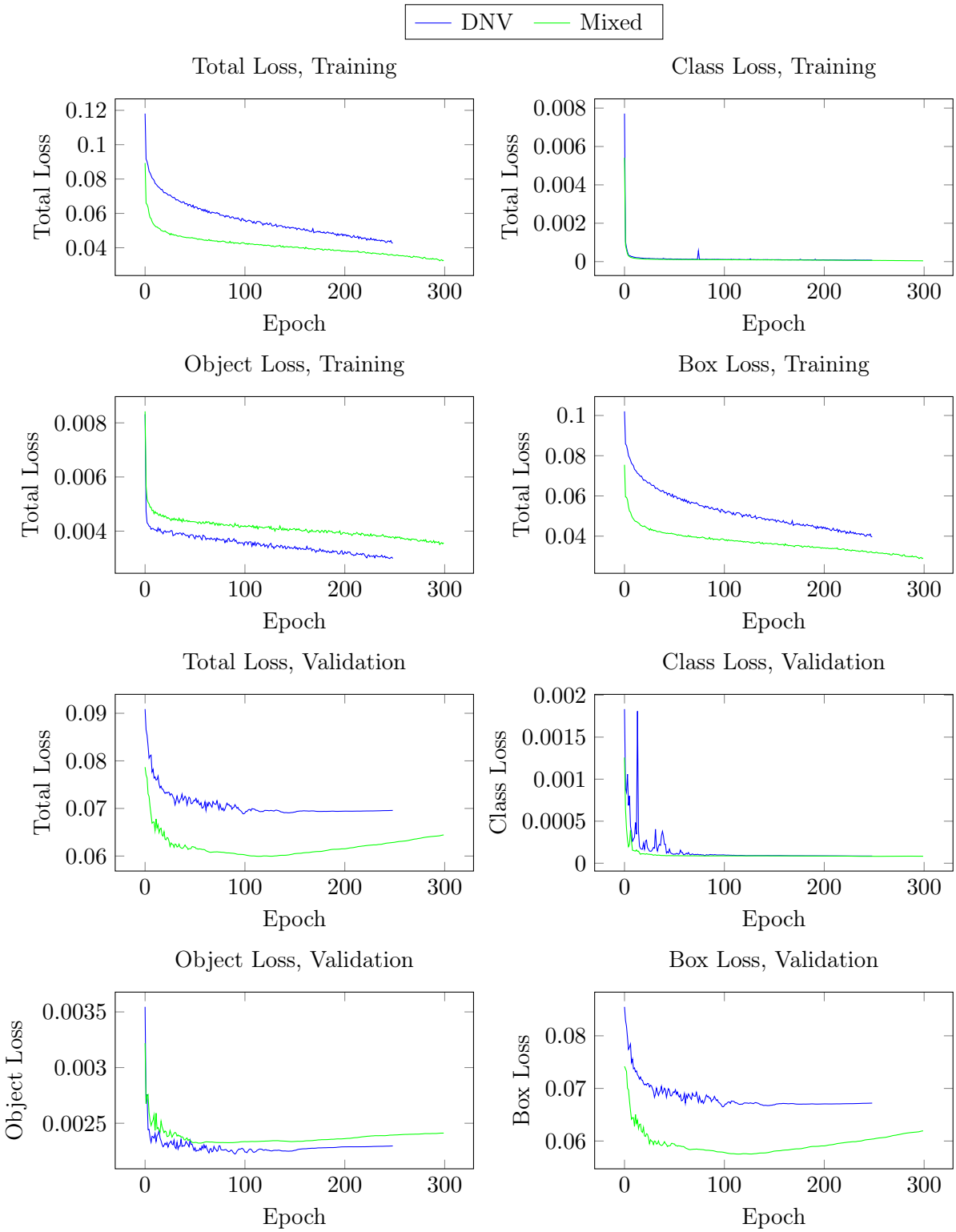


Figure B.5.: Plots of training and validation results from experiment 2.

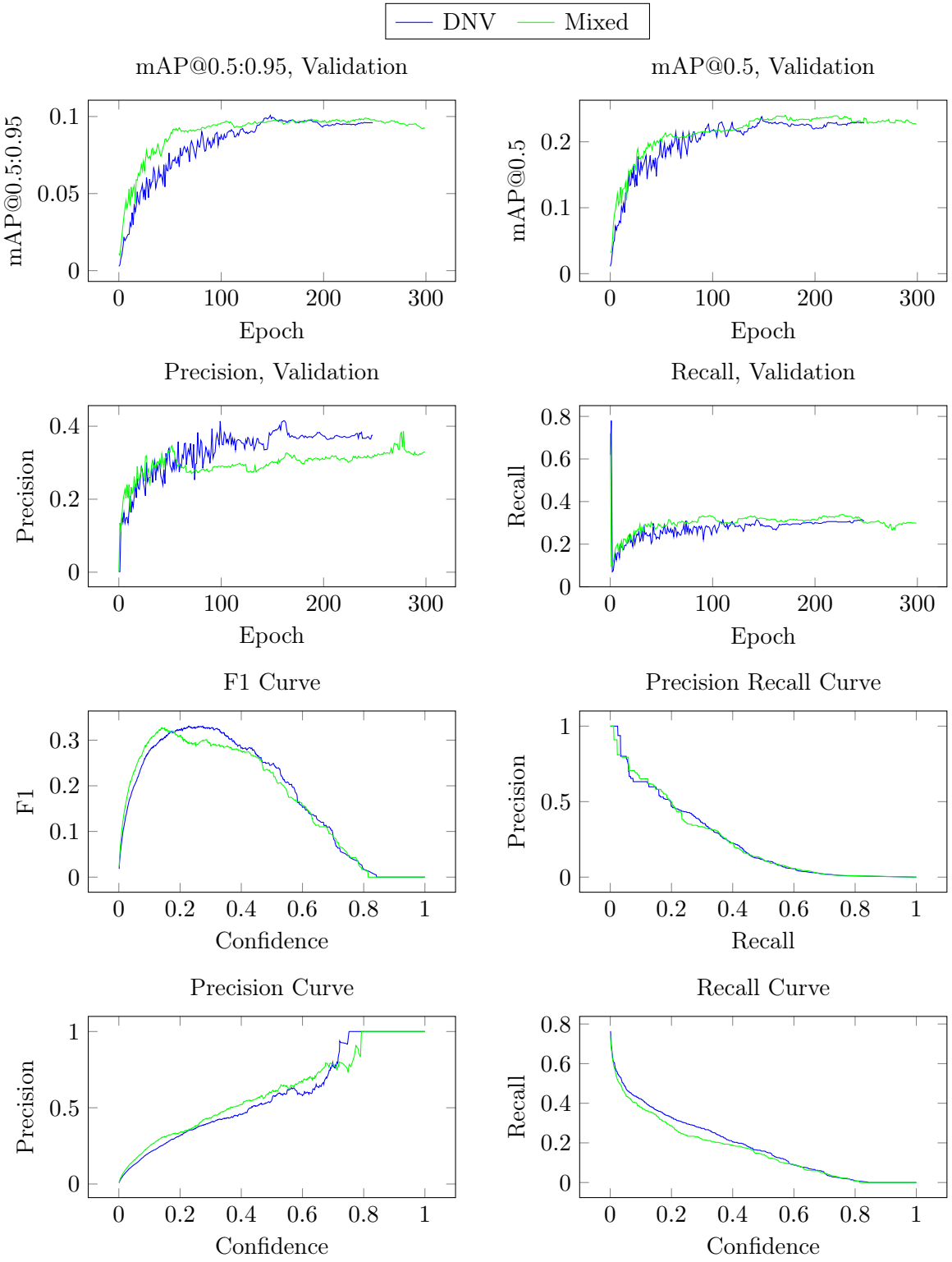


Figure B.6.: Plots of training and validation results from experiment 2.

B.4. Experiment 3 Results

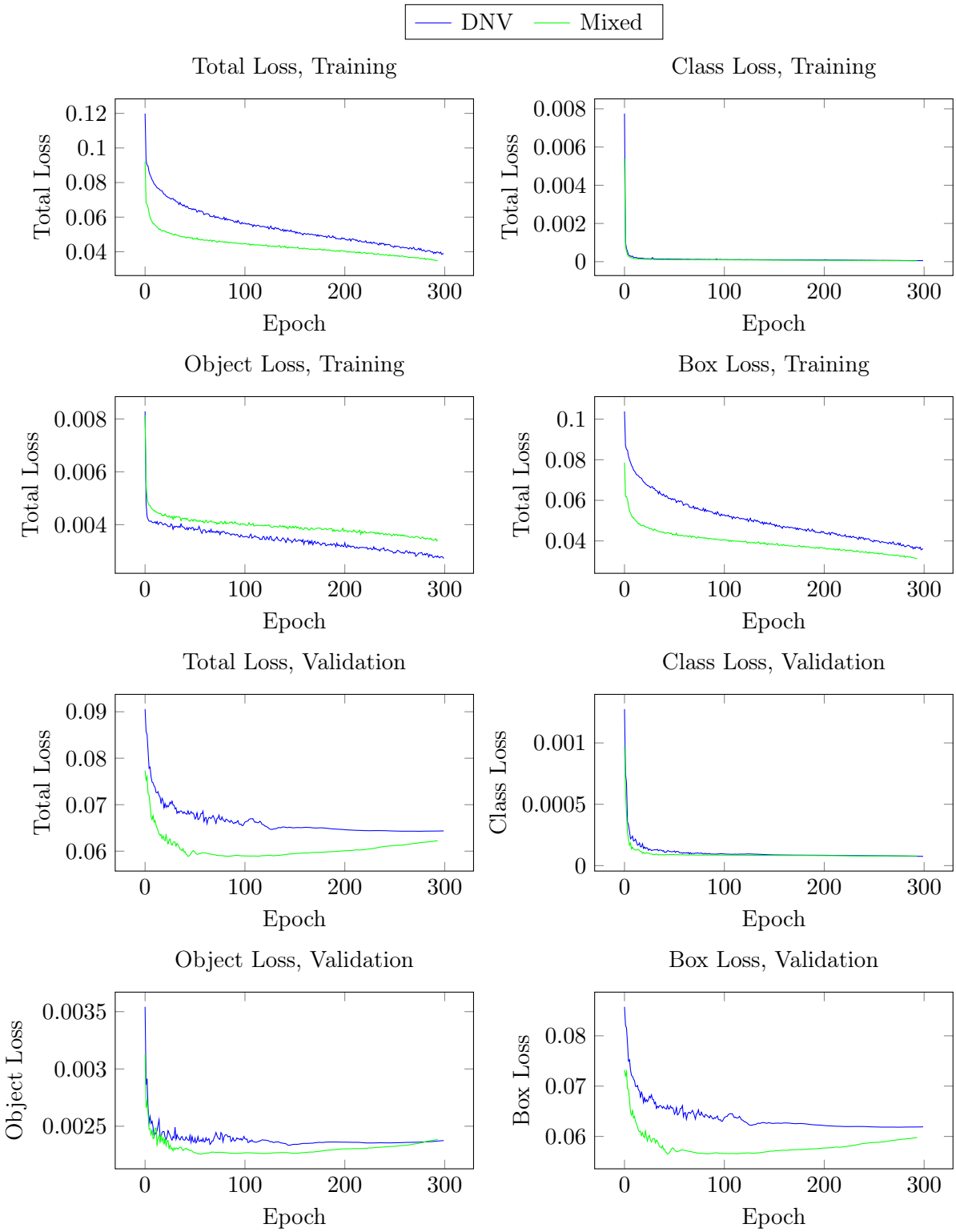


Figure B.7.: Plots of training and validation results from experiment 3.

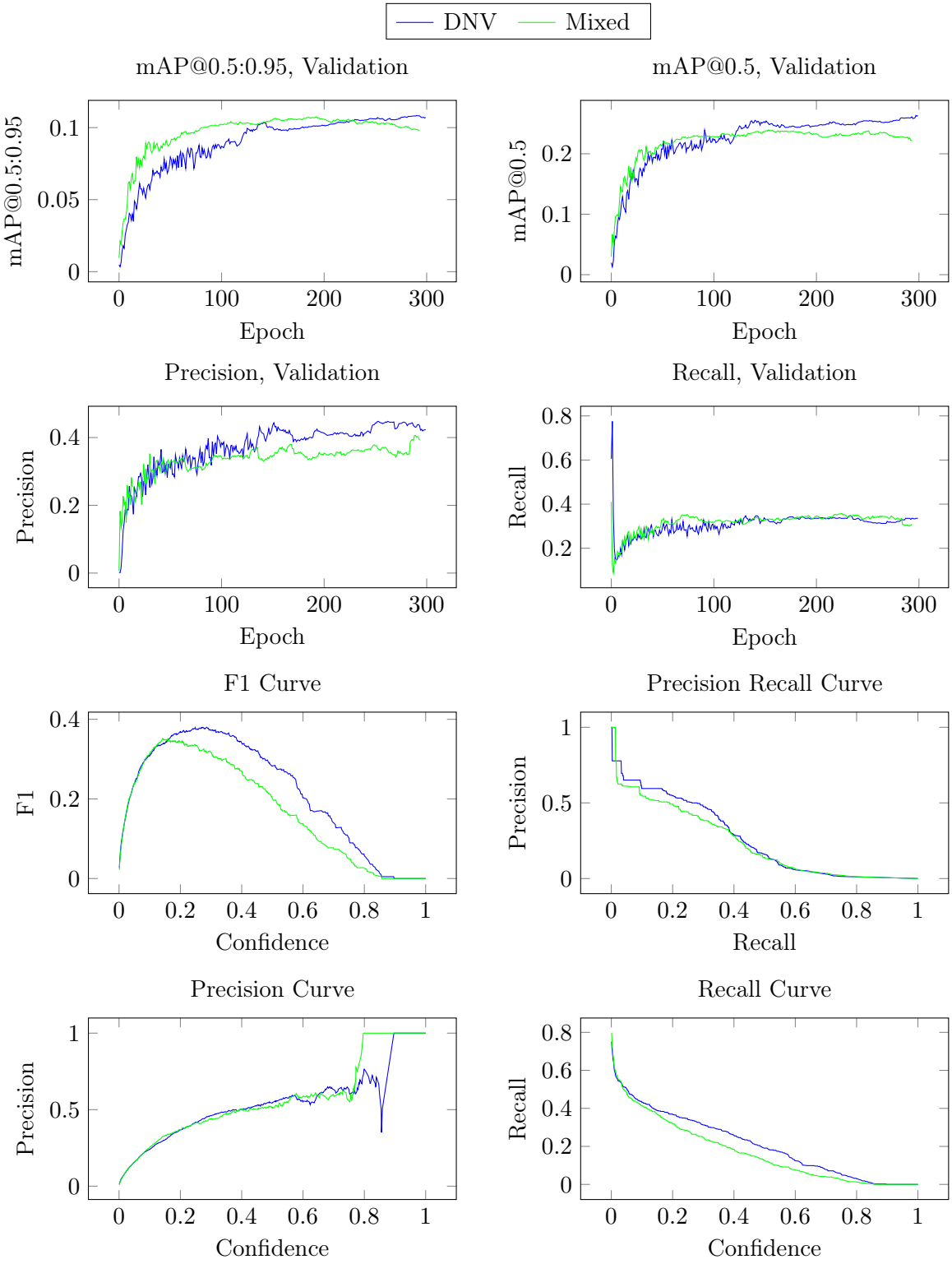
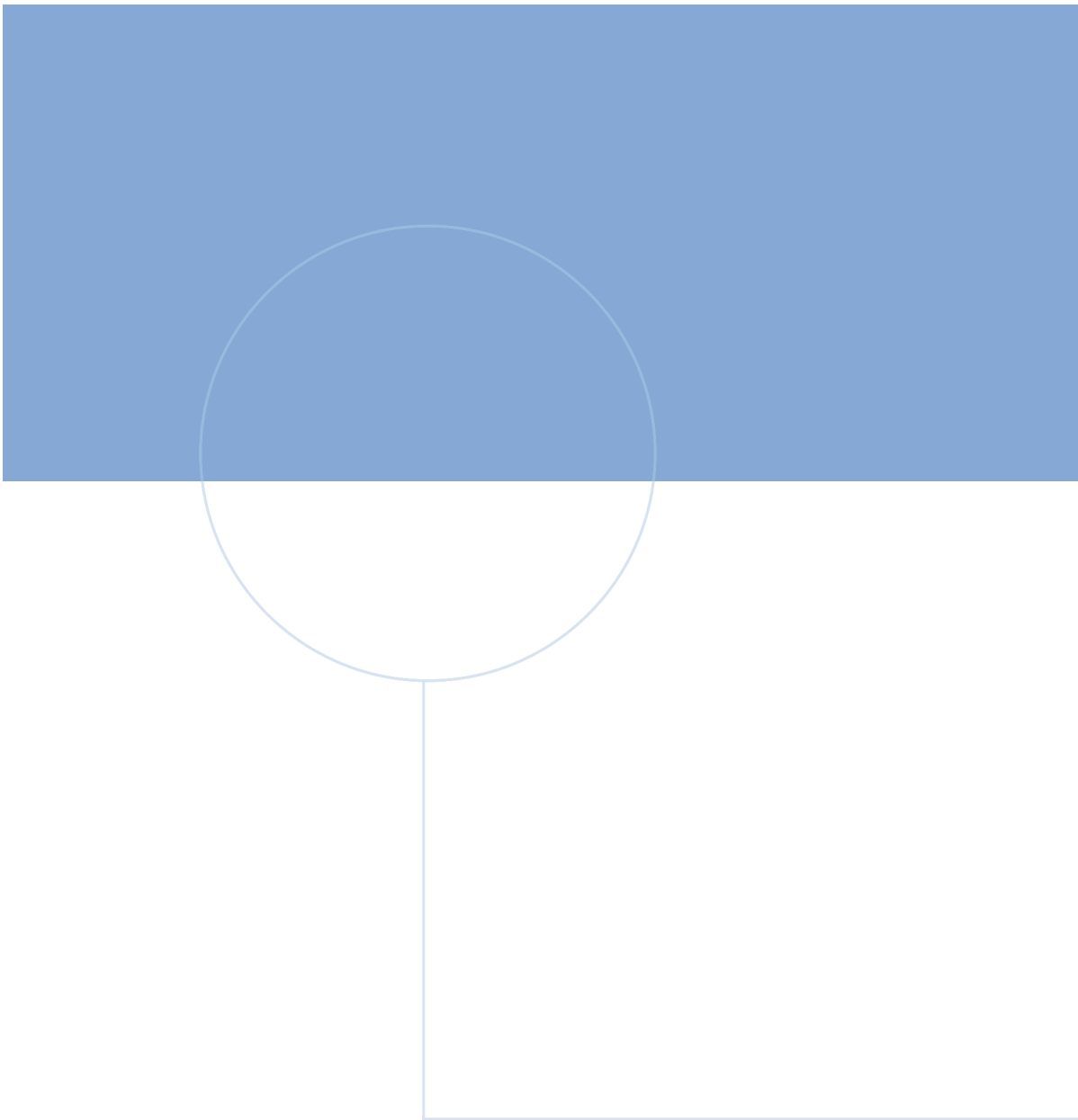


Figure B.8.: Plots of training and validation results from experiment 3.



NTNU
Norwegian University of
Science and Technology

DNV