Optimizing Synthetic Data Pipelines and Explainable AI for Industrial Applications

Internship Project report

ROB9

Aalborg Universitet
Det Tekniske Fakultet for IT og Design

AALBORG UNIVERSITY
STUDENT REPORT

The Technical Faculty for IT og Design
Niels Jernes Vej 10, 9220 Aalborg Øst
http://www.aau.dk

Title:
Optimizing Synthetic Data Pipelines and Explainable AI for Industrial Applications

Theme:
Technological Project Work

Project Period:
July 2024 - December 2024

Project Group:
None

Participant(s):
Adrián Sanchis Reig 20232285

Supervisor(s):
Dimitris Chrysostomou

Copies: 1

Page Numbers: 43

Date of Completion:
06. 01. 2025

Abstract:

This project report outlines the development of advanced quality inspection and synthetic data generation systems to support industrial manufacturing processes. Conducted during an internship at Mercedes-Benz Group AG, the project focused on leveraging explainable AI (xAI) techniques and automating synthetic data generation workflows. The report explores the integration of Shapley values and k-Nearest Neighbors (kNN) to enhance the interpretability of AI models used for defect detection.
Key aspects of the project include the design and implementation of an optimized rendering pipeline using Blender and BlenderProc, which automates the generation of photorealistic industrial environments and their corresponding annotations. Significant effort was dedicated to improving rendering efficiency, reducing processing times, and enhancing the scalability of the pipeline. The project also included the migration of machine learning training workflows to a cloud-based environment using Databricks, streamlining data handling and scaling capabilities.

Contents

Preface

Aalborg University, 24. May 2022

Many thanks to my supervisors Dimitris Chrysostomou, Sarvenaz Sardari and Jose Moises Araya Martinez for their amazing support and guidance during this internship.

Adrián Sanchis Reig
asanch23@student.aau.dk

1 Introduction

The introduction of Industry 4.0 is transforming manufacturing and business models through the integration of communication, information, and intelligence technologies. These advancements enable higher efficiency, improved quality, and better workplace conditions [1].Industry 4.0 represents the fourth industrial revolution, characterized by a new level of organization and control over the entire product life cycle. This revolution combines fields such as the Internet of Things (IoT), Industrial Internet, Smart Manufacturing, and Cloud-based Manufacturing [2]. Moreover, its progress is fueled by cutting-edge technologies such as artificial intelligence and robotics [1].

Among the numerous advancements driven by Industry 4.0, quality inspection plays a crucial role in ensuring product conformance to specific requirements. This process involves inspecting and measuring product quality characteristics using specialized equipment and procedures. By comparing results to defined standards, manufacturers can identify compliant products and discard defective or low-quality parts. This helps to improve production quality and address issues in the manufacturing process [3].

Synthetic data generation is another pivotal development in Industry 4.0. It offers new opportunities for improving the efficiency and adaptability of production environments. However, current systems often face challenges in anticipating and adapting to changes in production processes, which can deal with significant costs and require extensive resources [4]. The ability to generate flexible and scalable synthetic data addresses these challenges, enabling manufacturers to simulate and optimize production workflows effectively.

1.1 Collaboration with Mercedes-Benz Group AG

The project-oriented internship was conducted in collaboration with Mercedes-Benz Group AG in Stuttgart, Germany.

Mercedes-Benz Group AG is a world-leading premium and luxury car manufacturer. Founded in 1926 by Karl Benz and Gottlieb Daimler, the company made history with the invention of the automobile [5]. Today, the group operates in 17 countries across five continents, with its headquarters in Stuttgart, Germany. The company's focus remains on innovative and sustainable technologies, as well as on producing safe and superior vehicles that captivate and inspire [6].

The internship aimed to collaborate with their team at the ARENA2036 research campus, which focuses on applying cutting-edge technologies in production systems and quality assurance. These efforts are primarily centred on artificial intelligence and its integration into large-scale production processes.

1.1.1 Workspace ARENA2036

ARENA2036 (Active Research Environment for the Next Generation of Automobiles) is a leading interdisciplinary research campus located in Stuttgart, Germany. It serves as a hub for innovation in the fields of automotive engineering, production systems, and digitalization. Established in 2013, the campus fosters collaboration between industry leaders, academic institutions, and startups to develop groundbreaking solutions for the mobility of the future.

The workspace at ARENA2036 is designed to facilitate agile and collaborative working methodologies. It features modular infrastructure, state-of-the-art laboratories, and open spaces that encourage creativity and interaction. This flexible setup allows teams to prototype, test, and implement ideas efficiently, bridging the gap between research and application.

During the internship, the ARENA2036 environment provided a dynamic and inspiring setting to explore advanced technologies in artificial intelligence. The focus was on leveraging AI to enhance production quality and efficiency, aligning with Mercedes-Benz's vision of Industry 4.0. The campus's unique ecosystem enabled seamless collaboration with experts from diverse fields, contributing to innovative advancements in automotive manufacturing.

2 Problem Analysis

This chapter outlines the two main projects undertaken during the internship, providing an in-depth analysis of the tasks and challenges. The work primarily focused on leveraging artificial intelligence (AI), particularly deep learning, to develop innovative solutions for quality inspection and synthetic data generation. These efforts align with the goals of advancing manufacturing processes within the framework of Industry 4.0.

The two projects of this internship have been:

- **Quality Inspector:** This project aims to enhance the reliability and efficiency of quality inspection processes using AI-driven techniques.
- **Synthetic Data Generation:** This project aims to automate and optimise synthetic data workflows to overcome the challenges of data scarcity and annotation in industrial settings.

The following sections will describe in detail each project, providing insights into their problems, objectives and strategies employed to address key challenges.

2.1 Quality Inspector project

The Mercedes-Benz AG factory in Sindelfingen specializes in producing metallic components for vehicles. These metal sheets, which are welded together to form the vehicle chassis, go through a quality inspection process to ensure the absence of malformations or anomalies.

One critical step in the production involves welding nuts onto the metal sheets. These nuts play a critical role in vehicle safety, as they are positioned directly below the seat belts. Consequently, any defects or production issues must be detected and addressed promptly. An example of these metal sheets is shown in Figure 2.1.

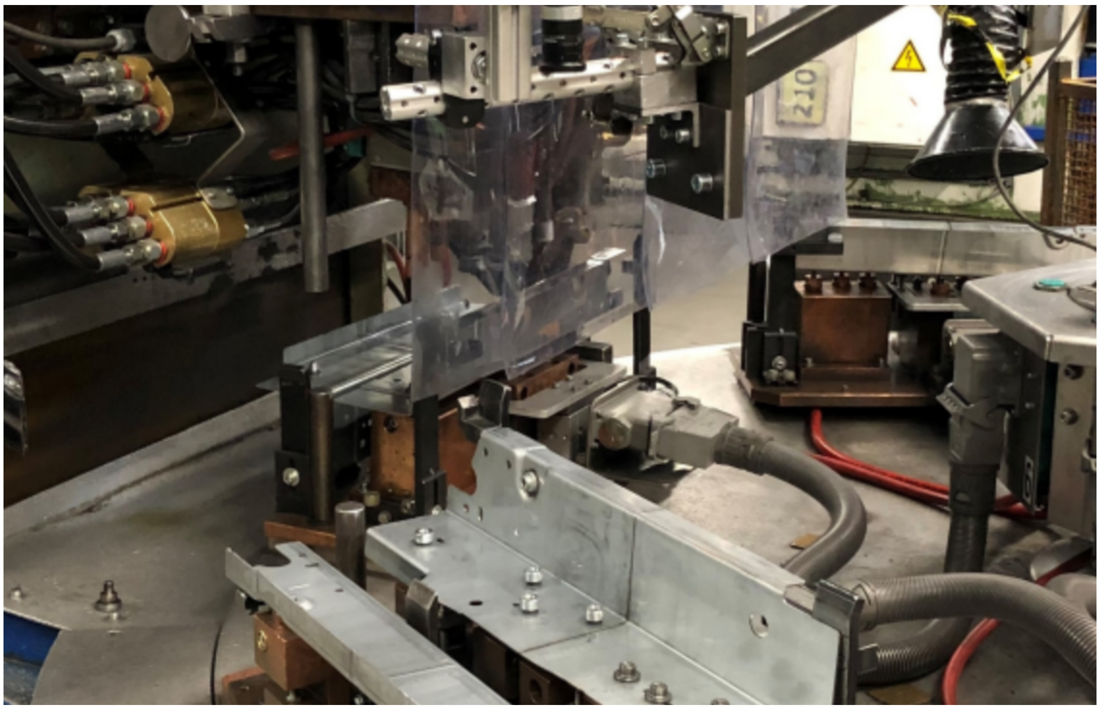


Figure 2.1: Example of metal sheets at factory

The quality inspection process is influenced by several factors, including the configuration, shape, and conditions of the objects being inspected. These variables need adaptable inspection solutions to account for the diversity of components. Relying only on solutions designed specifically for metal sheets risks limiting the scalability of the inspection system, as it may not generalize effectively to other types of parts.

To address the challenges in quality inspection, my supervisors at the arena Sarvi, Moises, et al. are developing a Quality Inspection project as part of their PhD in collaboration with Mercedes-Benz AG. This project features a Human Machine Interface (HMI) designed to streamline the inspection process. The HMI enables users to capture images of any metal sheet or object, label them, and subsequently train a machine-learning model on the created datasets. The trained model learns to distinguish between faulty and correct objects and its performance can be tested through real-time inference.

The inference functionality includes support for various xAI techniques. These tools provide insights into the model's decision-making process, helping users understand its performance and identify potential issues or errors in the dataset.

The HMI comprises two main components:

- **Frontend:** An intuitive interface for controlling and receiving real-time feedback on its state. An example of this frontend is shown in Figure 2.2.
- **Backend:** Responsible for executing the training of the machine learning model and performing inference tasks.

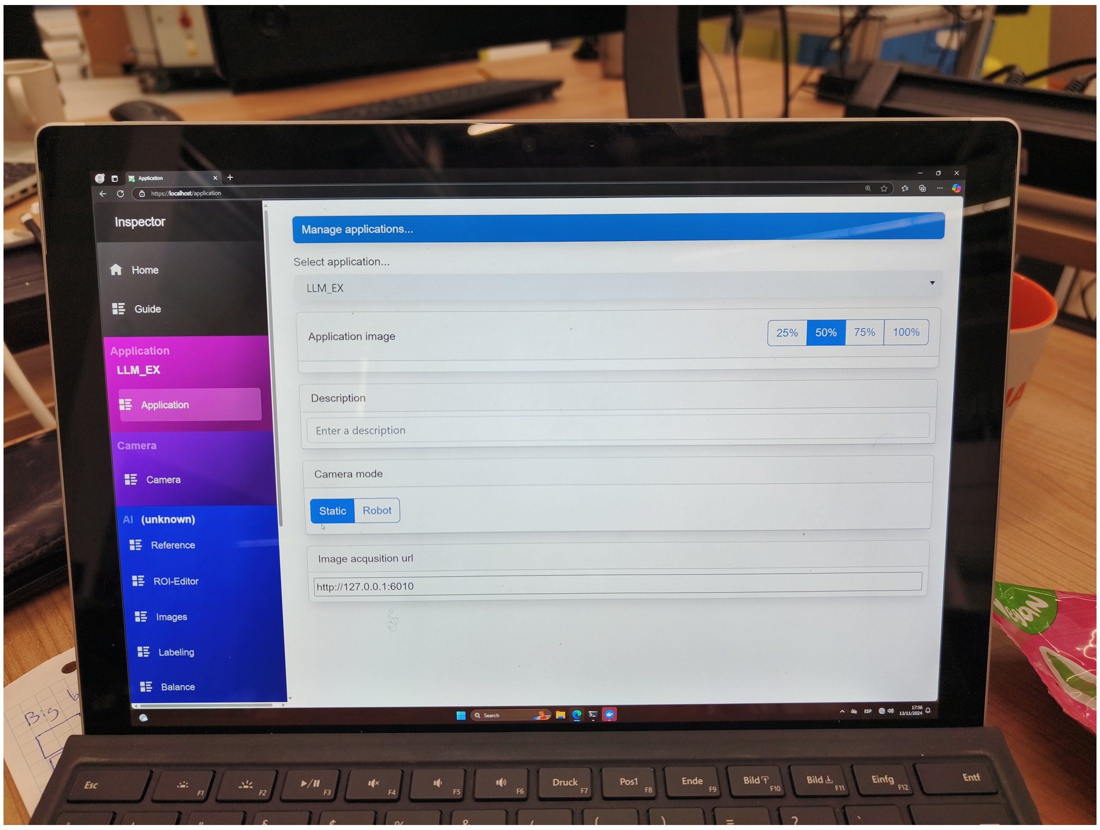


Figure 2.2: Frontend of the Scalable AI app

Initially, the frontend and the backend were running on different devices. Therefore, whenever the user wanted to train a new model, the frontend would send an HTTP POST/GET request to the backend running on the second device and start the training process. Once it has finished, the backend will save the model with the best accuracy and loss metrics, while notifying it to the frontend. Figure 2.3 depicts how the workflow between both machines was.

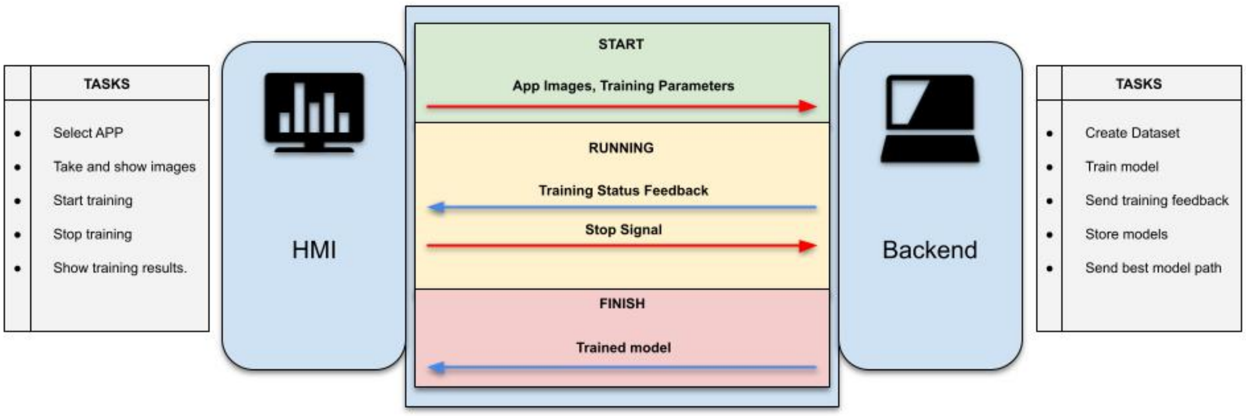


Figure 2.3: Old Workflow of the Quality Inspector App

Once the user starts the training, the backend sends a GET request to the frontend in order to retreive all the images from the dataset, which are taken and stored in the frontend device. The dataset comprises images classified into positive cases (good quality) and negative cases (defective). The core idea is to use these examples to understand how to classify an input image as good or bad.

The system employed a Siamese network to perform quality inspection by learning similarity metrics. This architecture is particularly effective for identifying faulty cases by comparing feature similarities. Referring to Figure 2.3, the training workflow is as follows:

- **Dataset Preparation:** The backend requests the dataset images and their associated labels to identify the regions of interest (ROI) within each image. Each image is cropped based on its ROI and categorized into two folders: IO (In Ordnung, positive examples) and NIO (Nicht In Ordnung, negative examples). This prepares the dataset for training.
- **Model Training:** The Siamese network is trained with specified parameters set via the frontend, such as the number of epochs and learning rate. During training, the network extracts features from input pairs and computes the Euclidean distance between their feature representations. This distance serves as a similarity metric, with smaller distances indicating higher similarity. By learning to minimize the distance for similar pairs (positive cases) and maximize it for dissimilar pairs (negative cases), the model becomes proficient in distinguishing between IO (In Ordnung) and NIO (Nicht in Ordnung) examples.
- **Frontend Updates:** During training, the backend provides status feedback to the frontend, allowing the user to monitor the training progress.
- **Performance Monitoring:** While the model trains, the system evaluates loss and accuracy metrics for each epoch to identify the best-performing model and determine if early stopping is necessary. The status is sent on each epoch as a POST request to the backend.
- **Training Completion:** After reaching the final epoch or receiving a stop signal from the frontend, the backend finalizes training. Then, it sends a summary update to the frontend, including details of the best epoch, relevant training data, and the model's saved location.

This workflow highlights the system's step-by-step approach to effectively prepare and train the Siamese network, ensuring accurate and reliable quality inspection results. A trained model can be easily tested since the frontend also has the option to run the inference in real-time, as can be seen in Figure 2.4.

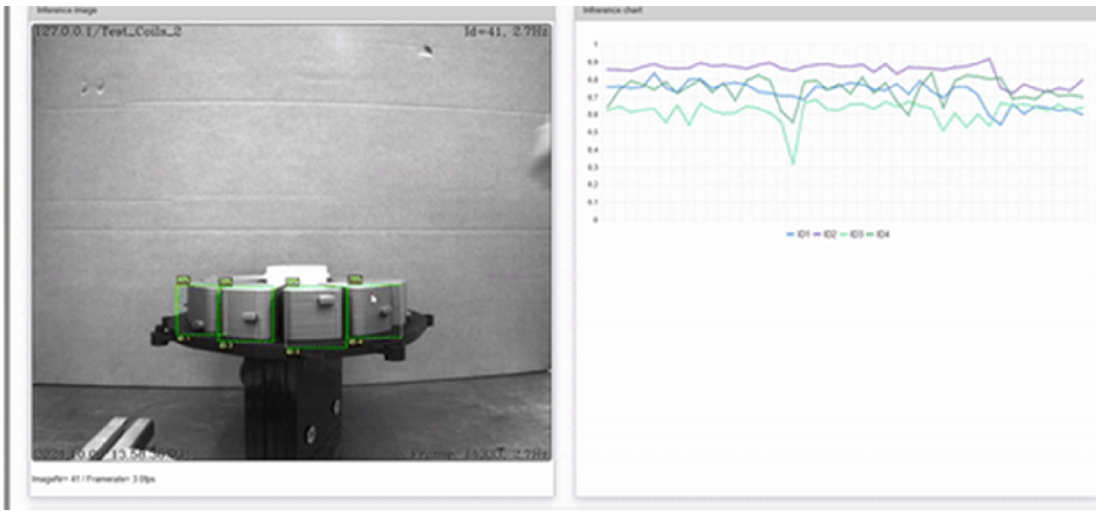


Figure 2.4: Example of inference measuring whether the coils of an electric motor are in the correct position or not.

The previous image depicts how the inference looks on the HMI, showing on the left side the input frame, which can be taken from the camera in real-time and on the right side each input frame being classified as IO (positive case) or NIO (negative case) based on how high is the predicted value (the closer to 1 the more sure the model is that it is an IO).

2.2 Synthetic Data Generation project

To train the aforementioned Siamese network, a dataset labeled with both IO (In Ordnung) and NIO (Nicht in Ordnung) cases is essential, as this architecture relies on positive and negative examples. While multiple publicly available datasets, such as COCO [7], have driven advancements in supervised object detection, data availability remains a significant barrier in industrial environments [8]. This limitation poses challenges for the widespread adoption of machine learning in these settings. Consequently, there is growing interest in the creation of synthetic data [9]. The prevalence of Computer-Aided Design (CAD) files in industrial contexts provides an opportunity to automate dataset annotation and generation [10].

The synthetic data generation approach employed by the team combines **Domain Randomization (DR)** and **Guided Domain Randomization (GDR)**. These methods generate synthetic datasets where relevant images are selected based on low-level and high-level features derived from real test images. These features serve as domain-specific information to bridge the gap between synthetic and real data.

- **Low-level features** include pixel-level attributes such as contrast, blur, lighting conditions, and colour variations.
- **High-level features** represent more abstract information, such as objects that define a factory setting, a meeting room, or an open space.

The team leverages these synthetically generated datasets to train industrial multi-class object detection models, such as YOLOv8 [11], addressing challenges like limited real-data availability and the high cost of manual annotation. By employing a GDR approach, the pipeline aims to minimize the sim-to-real gap and enhance the usability of synthetic data in real-world applications.

Figure 2.5 illustrates the conceptual framework of the previous synthetic data generation pipeline. One component involves the acquisition of real images from a context-related scene. The other component involves the generation of synthetic data via a rendering pipeline while for the data selection process, a simple image hashing [12] is performed for filtering synthetic images by their semantic content.

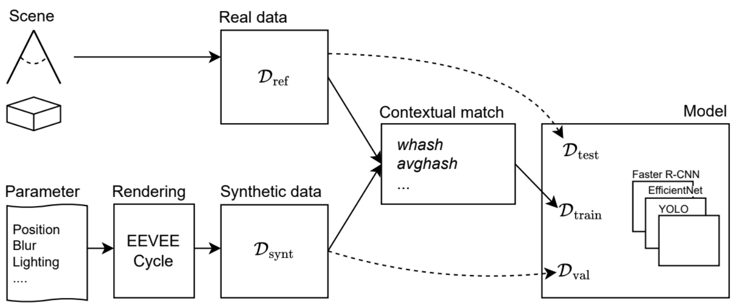


Figure 2.5: Old Synthetic data generation pipeline. It comprises a domain randomizer framework for photo-realistic image generation and a context-aware image selection based on high-level (semantic) and low-level (pixel information) features.

The key element of the pipeline, and which will be mainly described as it is a main section of this internship, is the rendering part. Implemented using Blender [13], a whole scene was created with multiple backgrounds and distractor elements common in an industrial environment so that there is a semantic context. The simulation can also control lighting conditions, camera parameters and the position of all the objects in the scene as can be seen in figure 2.6 which shows the starting point of the rendering.

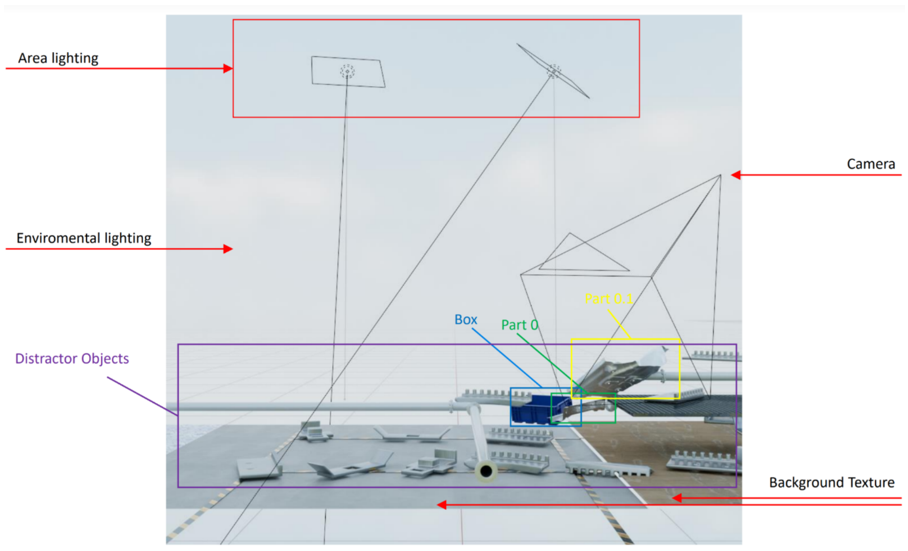


Figure 2.6: Elements present in a typical scene of the old synthetic data generation framework.

To create diverse backgrounds, multiple planes are set up, each with a unique background texture. The simulation generates 1000 frames in total, where the training objects (Box, Part 0, and Part 0.1) move around in front of the camera, orbiting and spinning at the same time.

In the earlier rendering setup, creating synthetic data with annotations involved two separate rendering steps. First, the Cycles render engine [14], known for its photorealistic ray-tracing capabilities, was used. During this phase, the camera followed a predefined path across the planes, which were lined up side by side. The target objects (Box, Part 0, Part 0.1) followed the camera's movement while being occasionally blocked by distracting objects like metal rods, plates, or pieces of furniture. Adding to the complexity, the lights in the scene moved along their paths too, changing brightness as they went to create dynamic lighting effects.

After the Cycles render was complete, it produced 1000 RGB images. To generate segmentation masks for these images, the simulation was run again, but with a few tweaks. This time, all the lights in the scene were turned off, making the background completely black. A glowing material was applied to the target objects so they were the only visible elements, each in a unique colour. Since these segmentation masks didn't need to be photorealistic, the faster EEVEE render engine was used instead of Cycles, cutting rendering time dramatically—up to 30 times faster.

The real images were split into two groups: a reference dataset, D_{ref}, and a test dataset, D_{test}. Finally, the YOLOv8 model was trained on the training dataset, D_{train}, which included RGB images and instance segmentation masks created using both Cycles and EEVEE. Figure 2.7 shows how the target objects Box, Part 0, and Part 0.1 appear when rendered with Cycles, alongside examples of real images and a synthetic scene.

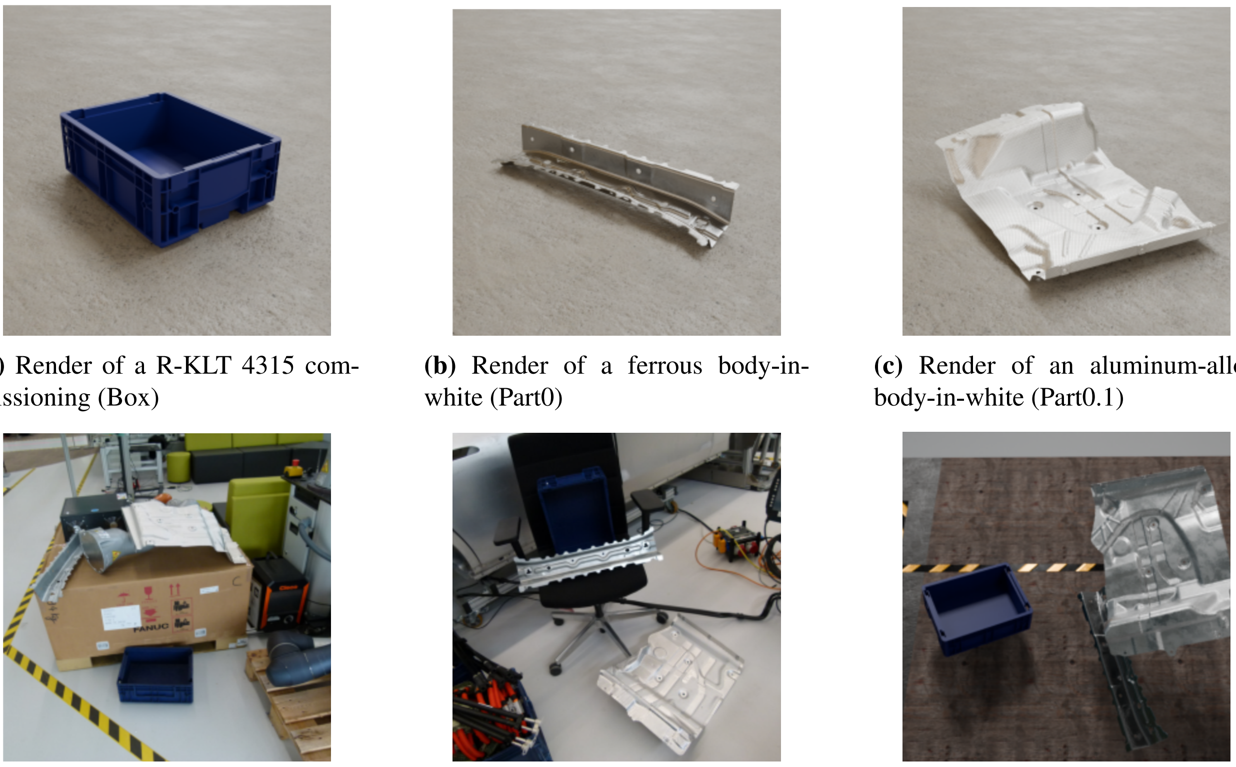

(a) Render of a R-KLT 4315 commissioning (Box)
(b) Render of a ferrous body-in-white (Part0)
(c) Render of an aluminum-alloy body-in-white (Part0.1)
(d) Real test scene $\in D_{train}$
(e) Context scene
(f) Synthetic scene

Figure 2.7: Examples of rendered images and real images:
Top row: renderings of all three target objects present in the industrial dataset.
Bottom row: 2.7d real photo belonging to the test set, 2.7e unlabeled reference frame for contextualization and 2.7f a rendered frame $\in D_{train}$ with elements from 2.7a, 2.7b and 2.7c.

This pipeline rendered 1000 frames using 250 samples for Cycles render engine. The total time it takes is 5.9 hours, or 21.24 s/frame. This time will change depending on the amount of distractors in the scene and the complexity of it. For instance, adding more background textures, more complex meshes or complex materials can also increase the rendering time.

2.3 State of the art

This section will introduce relevant background for this project in order to better understand the key aspects behind the two main sections of the internship.

2.3.1 Siamese Networks

Introduced by Bromely and LeCun in the early 1990s, Siamese Neural Networks (SNN) is a type of architecture designed to learn and measure the similarities between two or more inputs [15]. Although they were initially developed to address the signature verification problem, they have been widely adopted across various domains. Some of these domains include similarity comparison, forgery detection, facial recognition, etc.

Siamese networks consist of twin subnetworks that share identical parameters, i.e weights. This parameter sharing is what ensures that the same function is applied to all the inputs, preserving consistency in the extraction and comparison of features. These subnetworks, often Convolutional Neural Networks (CNNs), encode the inputs into latent feature space, where a similarity score can be computed using a distance metric or an energy function. Figure 2.8 provides a visual representation of a Siamese network.

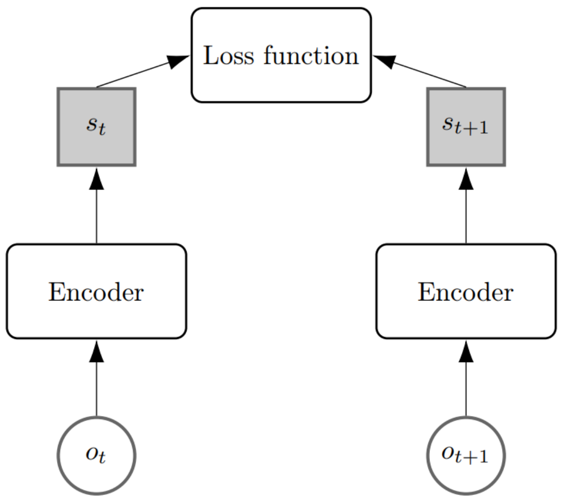


Figure 2.8: Representation of siamese encoders: networks with tied (shared) parameters. (Extracted from [16])

Bromley and LeCun's original implementation used a loss function that reduced the distance between similar pairs and increased it for dissimilar pairs [15]. Later advancements introduced weighted L1 distance combined with a sigmoid activation function to predict similarity scores between 0 and 1 [17], making the architecture adaptable to binary classification tasks.

Loss functions are critical in guiding the training of Siamese networks. Two prominent choices are contrastive loss and triplet loss. Contrastive loss aims to bring similar pairs closer in the feature space while pushing dissimilar pairs apart [17]. On the other hand, triplet loss incorporates an anchor, a positive sample, and a negative sample to refine the latent space further, ensuring more robust separation [18]. This approach is particularly effective for applications like fine-grained classification and facial recognition.

Siamese networks have been applied across diverse domains. In signature verification and facial recognition [15], [17], they excel at learning a similarity metric. In time-contrastive networks, they model temporal dependencies, as demonstrated by Sermanet et al. [19]. Lesort et al. [16] and Goroshin et al. [20] extended Siamese networks to state representation learning (SRL), where they leverage temporal priors to predict sequential states, enhancing continuity in dynamic systems.

Another notable application is forgery detection, where Siamese networks outperform conventional distance metrics such as Euclidean or cosine distances [17], [18]. For instance, Goroshin et al. [20] utilized three Siamese networks to encode sequential states, feeding these representations into a predictive model to anticipate future states. This highlights the adaptability of Siamese architectures in handling complex temporal dynamics.

In summary, Siamese neural networks provide a versatile and robust framework for similarity-based learning tasks. From their inception by Bromley and LeCun [15] to subsequent refinements by Chopra et al. [17] and others, these architectures have significantly influenced modern machine learning. Their shared-parameter design, coupled with effective loss functions, has enabled breakthroughs in metric learning, temporal modeling, and beyond, establishing their role as a cornerstone in the field.

2.3.2 Explainable AI

Explainable Artificial Intelligence (XAI) has become an essential field for AI research, addressing the growing need for transparency in machine learning systems. XAI aims to make complex models more understandable by giving the user the necessary tools for better understanding how they make decisions. This is particularly important in fields such as healthcare, finance and autonomous systems, wherein understanding the logic behind the output of a model can play a big role in the real-world consequences.

Deep learning models often operate as black boxes, producing outputs without providing any insight into their decision-making process. This lack of interpretability can lead to trust issues and complicate debugging [21]. To address this, XAI methods offer tools and frameworks to interpret and validate AI predictions. These methods can be intrinsic—where the model itself is designed to be interpretable, or post-hoc, which provide explanations for already trained models. Doshi-Velez and Kim [22] emphasize the importance of actionable insights and propose ways to rigorously evaluate interpretability in machine learning systems.

In object detection, where models identify and classify multiple objects in complex scenes, XAI techniques like k-Nearest Neighbors (kNN) [23], Shapley values [24], and Gradient-weighted Class Activation Mapping (Grad-CAM) [25] have proven particularly valuable. These methods not only improve our understanding of predictions but also build trust in the models.

2.3.2.1 k-Nearest Neighbors (kNN) for Explainability

k-Nearest Neighbors (kNN) is a simple yet effective technique for enhancing interpretability. By comparing a data point to its closest neighbors in a feature space, kNN provides a tangible way to explain model predictions. This approach is especially useful in object detection, where embeddings from intermediate layers of a neural network can help identify similar instances.

For example, when a model detects a car in an image, kNN can show other car examples from the training data that have similar features. This comparison helps validate the prediction and makes the model's decision-making process more transparent. Additionally, by analyzing how neighbors are distributed across different classes, kNN can uncover biases in the training data, guiding efforts to improve fairness and model performance. These capabilities make kNN a practical and intuitive tool for debugging and validating AI systems [23].

2.3.2.2 Shapley Values

Shapley values is a concept from cooperative game theory which provides a way to fairly measure the contribution of individual features to the predictions of a model [24]. The core idea is to assess how each feature impacts the outcome by evaluating all possible combinations of features. This approach ensures that every feature gets credit proportional to its actual influence, making Shapley values one of the most robust tools for interpretability.

In object detection, Shapley values can help uncover the regions of an image that matter most for identifying an object. For instance, in detecting a dog, Shapley values might show that the model relies heavily on the ears and eyes, offering a clear explanation of why the prediction was made. This kind of insight does not just help validate the reasoning of the model, it also makes it easier to spot and fix issues, like when the model focuses on irrelevant parts of an image.

One of the main issues with Shapley values is that calculating them can involve a high computational cost, as it needs to evaluate every possible combination of features. However, methods like SHAP (Shapley Additive exPlanations) can make this process more manageable by approximating the results without sacrificing much accuracy [24]. This has opened the door to using Shapley values in real-world applications, where interpretability often has to balance with efficiency.

2.3.2.3 Gradient-weighted Class Activation Mapping (Grad-CAM)

Grad-CAM is a practical and widely used technique for understanding how convolutional neural networks (CNNs) make decisions [25]. By generating heatmaps that highlight the areas of an image most relevant to a model's prediction, it provides a clear visual explanation of what the model "sees" when making a decision. This makes Grad-CAM especially useful in tasks like object detection, where understanding individual predictions is crucial.

The process begins with a forward pass through the CNN to calculate the target class score, such as the likelihood that an image contains a particular object. During the backward pass, Grad-CAM computes gradients of this score with respect to the feature maps of a specific convolutional layer. These gradients indicate how much each feature map contributes to the target class score.

Next, the gradients are averaged spatially to produce a weight for each feature map channel. These weights are then used to combine the feature maps, highlighting the regions that were most influential in the model's decision. A ReLU activation function is applied to the resulting heatmap to filter out negative values, focusing only on the positive contributions. Finally, the heatmap is resized to match the input image dimensions and overlaid for easy interpretation.

For example, if a model detects a dog in an image, Grad-CAM might highlight regions like the dog's face, fur, or ears. This helps confirm whether the model is focusing on meaningful parts of the image or being misled by irrelevant details, such as the background. Grad-CAM can also uncover issues, like reliance on spurious correlations, which can be addressed during model refinement.

One of Grad-CAM's strengths is its applicability to multi-object detection tasks. It can generate separate heatmaps for each detected object in a single image, providing detailed explanations for complex scenarios. Additionally, Grad-CAM requires no changes to the network architecture, making it straightforward to apply to existing models. By offering insights into model behavior, Grad-CAM helps practitioners debug errors, address biases, and improve the overall reliability of AI systems [25].

2.3.3 Synthetic Data Generation

Synthetic data generation refers to the process of creating artificial data in order to replicate the statistical properties and complexities of real-world datasets [4]. This approach is getting more attention in the field of machine learning since the process of collection and annotating data is both time-consuming and expensive [26]. Since machine learning is heavily dependent on it, some of the challenges it can solve are:

- **Data quality** is one of the most important aspects of a dataset. When data has not good quality, models can generate incorrect or imprecise predictions due to misinterpretation [27].
- **Data scarcity** is another relevant challenge. Data is not always easily available or the number of accessible datasets is insufficient [28].
- **Data privacy** is a challenge which is solved as soon as data is generated. Many datasets cannot be publicly released due to privacy and fair issues, making synthetic options an attractive alternative.

Synthetic data not only can be cost-effective but also highly customisable, allowing the creation of datasets focused on specific tasks such as object detection, segmentation and classification.

The generation of synthetic data typically involves simulation engines, 3D modeling tools and/or procedural pipelines. These systems allow the creation of virtual environments, the application of realistic textures, the simulation of lighting conditions and the generation of detailed annotations. This makes it not only scalable and reproducible but also diverse enough to improve the generalization capabilities of machine learning models [29].

One of the key stenghts of synthetic data lies in its ability to simulate corner cases or rare scenarios that are difficult to capture with real-world data. For instance, in autonomous driving systems wherein simulate extreme weather conditions or complex traffic scenarios can be easier than finding them in reality.

2.3.3.1 Domain Randomization

Domain randomization (DR) is a simple yet powerful technique for generating training data for machine-learning algorithms. Instead of trying to create perfect copies of real-world scenarios, DR focuses on introducing random variations in the generated data by modifying non-essential features for the learning task [30]. This strategy allows models to reduce the ”reality gap” by training on synthetic images that incorporate a big range of randomised parameters.

The key idea behind DR is that sufficient variability in simulation allows models to generalize successfully to real-world conditions. For instance, variations in lighting, textures and/or object positions ensure that models learn robust, invariant features rather than relying on specific visual cues. Exposing models to this variability during training enhances their ability to adapt to unseen environments [31].

To achieve meaningful results, the design of the simulation environment plays a crucial role. It is important to identify and modify parameters that do not contribute directly to the core task but significantly expand the variability of the training data. For instance, in object detection tasks, randomizing the background textures, camera angles and lighting conditions can help mitigate overfitting and improve the performance with real-world data.

Several research groups have already performed ablation studies to investigate the effects of randomizing various parameters. These include the aforementioned (camera angles, lighting, textures) and also flying distractors, and random noise [30].

2.3.3.2 Guided Domain Randomization

Guided Domain Randomization (GDR) is an improvement of DR on which reference features can be essential in creating a training set with high information [32]. In this context, GDR introduces constrains and real-world data distributions to guide the randomization process. By following this approach, the generated synthetic data remains not only diverse but also relevant to the specific task and domains being targeted. Moreover, GDR narrows the gap between synthetic and real-world data [32].

One of the defining features of GDR is the use of task-specific metrics or feedback loops to refine the randomization process. For instance, in autonomous driving simulations, real-world traffic statistics and lighting conditions can be used to prioritize the generation of more accurate scenarios.

Moreover, adaptive feedback mechanisms are often integrated into guided randomization frameworks to dynamically optimize training data generation. These mechanisms evaluate the performance of the model on validation or test data and iteratively adjust the randomization parameters to enhance training efficiency. Heindl et al. [33] showcased this approach in BlendTorch, where adaptive randomization was used to improve the sim-to-real transfer for robotic perception tasks.

Another interesting library for creating DR or GDR projects in BlenderProc [34]. Blenderproc is a modular procedural pipeline, which helps in generating real-looking images for the training of CNNs. These images can be used in a variety of use cases, including segmentation, depth, normal and pose estimation.

BlenderProc uses the python API of Blender [35] in order to add new functionalities useful for synthetic data generation. Some of the functionalities added are random sample of positions, randomization of textures and of trajectories among others. It is developed by the German Aerospace Center (DLR) as a tool for simplifying the generation of real-looking images of scenes that can be fully annotated. While the python API for Blender does not add these functionalities by default, BlenderProc also adds functions like rendering segmentation masks for specific objects, depth images and normal maps depending on the needs of the user.

The biggest benefit of this pipeline, is that it allows to obtain image annotations at the same time that it renders the RGB images of the scene, removing the need of rendering a simulation more than one time if multiple annotations are needed. The resulting data is stored compressed as a hdf5 file [36], making it easy to access the stored infornation through their corresponding keys (colors, depth, etc).

3 Requirements

The following requirements are considered essential for the improvement of both projects. They are based on the needs of the team in order to experiment and keep researching in both projects.

3.1 Problem statement

Based on the previously introduced projects, it can be concluded that there are multiple areas of improvement to accommodate the requirements of the team 3.2 and 3.3. Thus, the problem statement is the following:

How can their quality inspector/rendering pipeline be improved while maintaining its functionality and fulfilling the expected requirements?

3.2 Requirements for the Quality Inspector

As the HMI is right now, it is able to create datasets for specific usecases and train a Siamese model with it. However, the current workflow has some problems since frontend and backend are running on different devices instead of having one as a final product would require. Moreover, there are still some explainability options missing as well as an interest on the team for running the training on the cloud so that there is no need to have a very powerful computer running the HMI.

1. The training of the HMI should be able to run on the cloud.
2. The HMI should include Shapley Values as an explainability option.
3. The HMI should include KNN as an explainability option.
4. The whole project should be able to run on a single device rather than separated into two.

3.3 Requirements for Synthetic Data Generation

As the synthetic data generation is right now, a lot of work is needed in order to be used for another project, since the whole scene, trajectory and parameters have been done manually. Instead, the rendering pipeline should be redesigned to be more scalable and compatible with different scenes without needing an expert.

1. Create a new render pipeline fully automated.
2. Include BlenderProc in the new pipeline for automated annotations.
3. Object-to-camera translation $f(t_x)$ $[0m, 1.6m]$
4. Object-to-camera translation $f(t_y)$ $[0m, 1.2m]$
5. Object-to-camera translation $f(t_z)$ $[0.75m, 4.15m]$
6. Object-to-camera rotation $f(\alpha, \beta, \gamma)$ $\{(\alpha, \beta, \gamma) \mid [-180°, 180°] \in \mathbb{R}^3\}$
7. In frame object percentage $[0\%, 100\%]$

8.	Amount of distractors	$[0,10] \in \mathbb{N}$ obstacles per frame
9.	Environment lighting	$[0,300]W$
10.	Background textures	$[0,10] \in \mathbb{N}$ moving planes
11.	Blur / depth of field	F-Stop variation $[1.0,10.0]$
12.	Speed-up rendering time by at least two	$\sim$ 2.95 h / 1000 frames.

4 Implementation

In this chapter, the focus is on the work realised during the internship for fulfilling the requirements and goals. Additionally, it explores implementations for alternative usecases like the RoX project or the electric motor prototype.

4.1 Quality Inspection

4.1.1 Training migration to the cloud

In order to improve the Human-Machine Interface (HMI) for quality inspection developed by the team at Mercedes-Benz AG in the ARENA2036, all necessary modifications were implemented in the backend. This approach was required as the frontend was being developed by an external company, leaving the team without direct access to it. Despite this limitation, all required changes were successfully implemented without modifying the frontend.

One of the first challenges identified in the development of the HMI was the lack of adequate resources for training new models. Although the backend was running on a powerful computer (Acer laptop: i7-13700H, RTX 4070, and 32GB RAM), using this device for model training or testing new backend implementations meant monopolizing a critical team resource. As the team at ARENA2036 continued to grow, this became increasingly unsustainable. To address this, training processes were migrated to the cloud, specifically using Azure Databricks.

Azure Databricks [37] is a platform designed for building, deploying, sharing, and maintaining enterprise-grade data, analytics, and AI solutions at scale. It also integrates cloud storage, which is essential for saving trained models and datasets. Migrating to this platform involved several technical steps to ensure the compatibility and functionality of the backend code.

The migration process began by getting familiar with Azure Databricks and adapting the training code for the Siamese model to work within this environment. Projects can be created on Databricks, organized with necessary files and folders, creating a collaborative workspace that eliminates the need for local storage of the code. The project repository was cloned to the cloud, and an example dataset was manually uploaded to Azure Blob Storage. A machine-learning-compatible cluster was then created to run the training code. Databricks clusters [38] are computational resources that execute code, functioning similarly to Jupyter Notebooks. For this development, a 14.3 LTS cluster was selected, as it was compatible with Python 3.10 and already included the many machine-learning libraries required for the project. The cluster used T4 compute with 8 cores at a cost of 1.5 DBU/h[1].

After resolving library compatibility issues, the training code ran successfully on the platform. Accessing the example dataset stored in Azure Blob Storage required specifying the correct path. By mounting the Azure Blob Storage container to the Databricks File System (DBFS), files could be accessed through paths like *"dbfs:/mnt/<mount_name>/"*. This integration streamlined the process of managing datasets.

To enable backend communication with the cloud environment, the existing workflow was modified.

[1]DBU stands for Databricks Unit (DBU), a normalized unit of processing power on the Databricks Lakehouse Platform used for measurement and pricing purposes.

Previously, the backend would receive an HTTP request from the frontend to initiate training. The backend would then process the images first and then start the training. Figure 2.3 illustrates this original workflow. With the new workflow, after creating the dataset from frontend-provided images, the backend uploads it to Azure Blob Storage. From there, the training job is executed on Databricks.

The Azure Storage SDK, specifically the Python module *azure-storage-blob*, was used to upload and download files to and from Azure Blob Storage. To optimize the upload process, the dataset was compressed into a ZIP file before being transferred. Once uploaded, the Databricks SDK for Python, *databricks-sdk* [39], facilitated starting training jobs. This SDK supports setting GitHub credentials to automatically configure repositories. The workflow begins by starting a Databricks job with the path to the dataset as an argument. During job execution, the dataset is decompressed, the GitHub repository containing the training code is cloned, and training is initiated. An example of how this can be done can be seen on the snippet 4.1.1. A significant advantage of this approach is that updates to the training code on GitHub are automatically reflected during execution, as the repository is freshly cloned for each job. After training, the backend downloads the trained model using *azure-storage-blob*.

```python
import time
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.jobs import RunLifeCycleState

EXIT_STATES = [RunLifeCycleState.TERMINATED, RunLifeCycleState.SKIPPED,
 ↪  RunLifeCycleState.INTERNAL_ERROR]

# Initialize the Databricks workspace client
workspace = WorkspaceClient()

# Submit the job for execution
job_id = "<your-job-id>"
response = workspace.jobs.run_now(job_id=job_id)

# Get the Run ID
run_id = response.run_id
print(f"Job submitted successfully with Run ID: {run_id}")

# Poll the run status until it completes
while True:
    # Get the current status of the run
    status = workspace.jobs.get_run(run_id=run_id)
    life_cycle_state = status.state.life_cycle_state
    result_state = status.state.result_state

    print(f"Run status: {life_cycle_state}")

    # Check if the job has completed or failed
    if life_cycle_state in EXIT_STATES:
        if result_state:
            print(f"Run completed with result: {result_state}")
        else:
            print("Run ended, but no result state is available.")
        break

    # Wait before polling again
    time.sleep(10)
```

Snippet 4.1.1: Example code of how to start a job using the databricks SDK for Python and receive feedback from it.

While this setup efficiently handles dataset transfer, model training, and retrieval, it initially lacked a mechanism for providing status updates to the frontend. Previously, such updates were sent using REST APIs over the local network. However, with the training now executed in the cloud, direct communication between Databricks and the frontend was unavailable due to network isolation. This issue was resolved using Ngrok [40], a tool that creates secure tunnels to local servers. By deploying Ngrok, an HTTP server endpoint was established to map Databricks' cloud environment to the backend's local address. This allowed Databricks to send HTTP requests to the backend, which forwarded status updates to the frontend, restoring the communication pipeline.

A representation of the new workflow can be seen in Figure 4.1, it still shows on the left the dual device setup for the backend and the frontend since they are still running on different machines. However, now there is the option to delegate the training of the Siamese model to the cloud instead of perform the training locally. Moreover, the endpoint created with Ngrok also appears at the bottom.

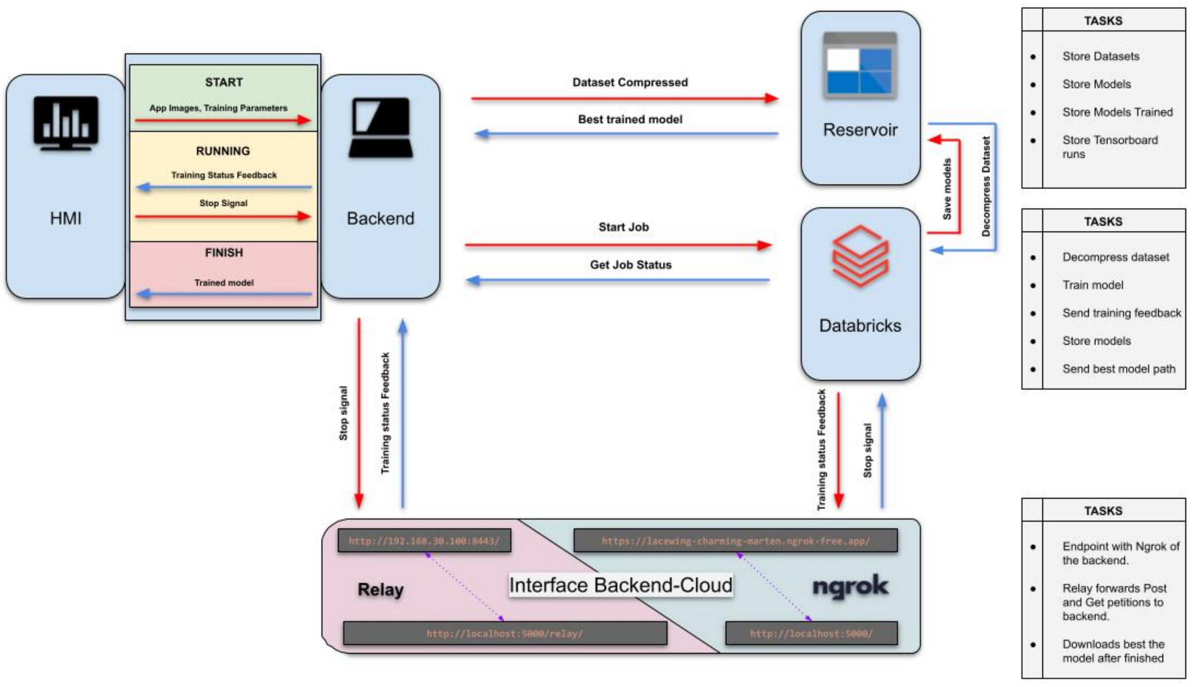


Figure 4.1: New workflow for cloud training

This cloud migration solution not only resolved the issue of running out of computers on the working area but also improved the scalability of the project, making it now able to train efficiently without monopolizing local hardware.

4.1.2 Adding Explainable AI

As discussed previously on 2.3.2, adding to the HMI explainable AI tools can help the user understand the reason why a trained model performs on a specific way. It can also be useful in order to detect cornercases that may have not been properly represented while creating the datset, and includes the possibility of improving any detected issue.

The explainability modes are inside of the inference section in the frontend. The HMI already had implemented the GRAD-CAM option, so the ones missing were Shapley values and KNN. These options can be selected in the inference section. From it, users can specify the threshold upon which the prediction will be considered as an IO case, whether it run will run in real-time or by taking snapshots from a camera each time a button is pressed and finally, whether it will use any of the implemented explainability options.

Shapley values can be obtained using the *shap* python module [41]. The module takes random images from the train set, which will be referred as background. These background images are feeded to the SHAP module through the *shap.DeepExplainer(model, background)* method, which takes an instance of the model (the siamese model) and the background images. Since this step can be computationally expensive, it is done before starting the inference so that the user does not perceive a considerable delay.

Once the model and background has been processed, shape values can be obtained for any image of the test set or from images obtained from the camera in real-time. The snippet 4.1.2 shows an example of how these values can be obtained and visualized.

```python
# ...include code from
↪  https://github.com/keras-team/keras/blob/master/examples/demo_mnist_convnet.py

import shap
import numpy as np

# select a set of background examples to take an expectation over
background = x_train[np.random.choice(x_train.shape[0], 100, replace=False)]

# explain predictions of the model on four images
e = shap.DeepExplainer(model, background)
# ...or pass tensors directly
# e = shap.DeepExplainer((model.layers[0].input, model.layers[-1].output), background)
shap_values = e.shap_values(x_test[1:5])

# plot the feature attributions
shap.image_plot(shap_values, -x_test[1:5])
```

Snippet 4.1.2: Example code of how to obtain SHAP values given a model and a random train images [41].

On the other hand, kNN is obtained using the *sklearn* python module. Before starting the inference, and in order to reduce the perceived delay by the user, the features of all the images in the train set are extracted using the ResNet-50 convolutional layer architecture with preload weights. This is the same extractor that the Siamese model uses for extracting the features of the input and reference images.

Once the features are extracted, the *NearestNeighbors* object from the *sklearn* module can be used to query the k-nearest neighbours of any input image based on the feature vectors. The object is initialized with the desired number of neighbors *k* and a distance metric (euclidean for our case).

To perform the query, the features of the input image are also extracted. These features are then passed to the *NearestNeighbors* object, which returns the indices of the k most similar images. For the case of the HMI, the three closest images from the train set are obtained. The following snippet 4.1.3 shows an example of how the 3 nearest neighbors were obtained for a given input on the HMI.

```python
from sklearn.neighbors import NearestNeighbors
import numpy as np

# Extracting features from train and input data
train_features = [features] # Features extracted for each image of the train set.
input_feature = features    # Features extracted for the input image.

# Fit the k-NN model
k = 3  # Number of neighbors
knn_model = NearestNeighbors(n_neighbors=k, metric='euclidean')
knn_model.fit(train_features)

# Query the k nearest neighbors for the input image
distances, indices = knn_model.kneighbors(input_feature)

# Output the results
print("Indices to k nearest neighbors:", indices[0][:k])
```

Snippet 4.1.3: Example of the code used to obtain the k nearest train images.

This xAI option is very useful to detect failures in the dataset since it allows one to check what the model considers as close images, allowing one to spot possible outliers and/or lack of data.

4.1.3 Migrating project to a single device

The last task done for the quality inspector HMI was the migration of the whole project or a single device. The HMI is part of a project which aims to create a product that will be used in a factory for quality inspection. Therefore, it needed to be migrated to the final device on which it will run for any demonstration or once the project has finished. Figure 4.2 shows the final device at the workshop working.

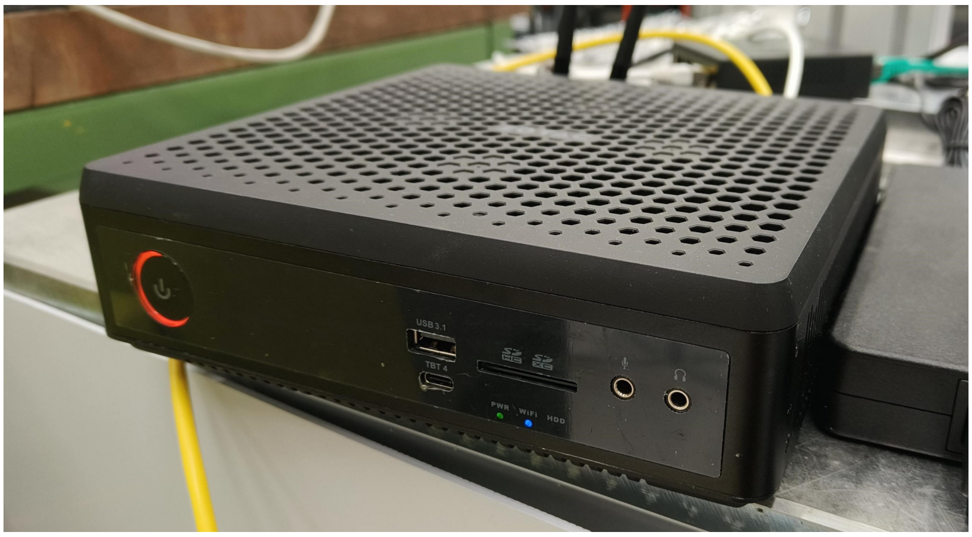


Figure 4.2: ZBOX Magnus on at the workshop

The final device is a ZOTAC ZBOX Magnus EN474070C, which is equipped with: i7-14th Gen, a RTX 4070 and 16GB of RAM. This small computer is powerful enough to run both the frontend and the backend at the same time while training new models if needed.

The operative system of the ZBOX is Windows 11, which is the same so as the one where the frontend is running. Because of that, the migration of the frontend to it was straightforward, and without needing to change anything. However, since the backend runs on Ubuntu 20.04 it is not possible to directly migrate the project making many changes. In order to solve this, the backend was migrated inside of a docker container with an image of Ubuntu 20.04. By doing so, the amount of changes needed in order to make the backend work on the same device as the frontend are minimal.

In order to allow the Docker to receive and reply back to the messages of the frontend, the ports 8443 and 5000 within the container are mapped to the ones on the host machine. This allows not only the communication between frontend and backend but also the communication between the backend and any client which connects through the Ngrok domain. Figure 4.3 shows what the final workflow looks like.

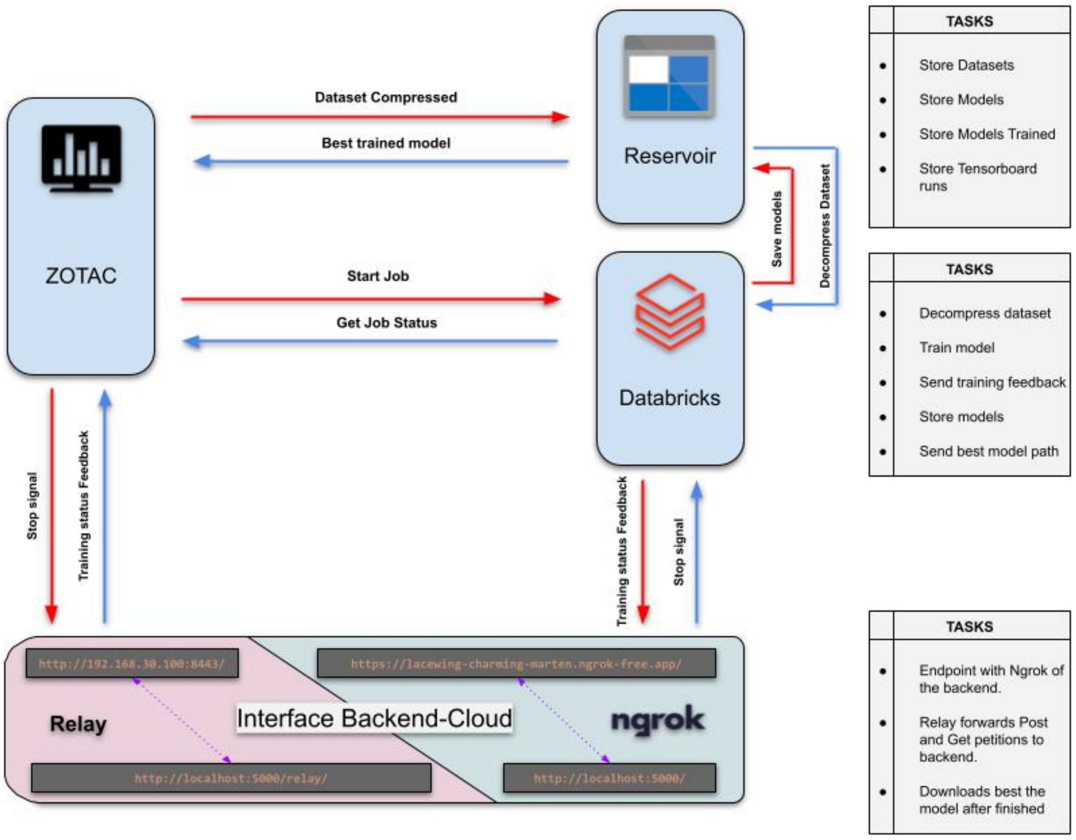


Figure 4.3: Final workflow of the HMI after moving the whole project to a single device

These changes allow reducing the use of hardware even more since now the project is able to run on its dedicated device and without the need to use any extra computers.

4.2 Synthetic Data Generation

4.2.1 New render pipeline

The two biggest problems with the old render pipeline were its lack of scalability and its inability to directly annotate rendered data.

The Blender scene consisted of a world with one plane for each background texture. On top of each plane, distracting objects were duplicated multiple times to populate each plane. This setup resulted in a very slow simulation that consumed significant memory and lacked flexibility. For instance, adding a new distractor or moving the camera closer to the objects required creating an entirely new scene with new trajectories. Figure 4.4 illustrates the Blender scene of the old pipeline.

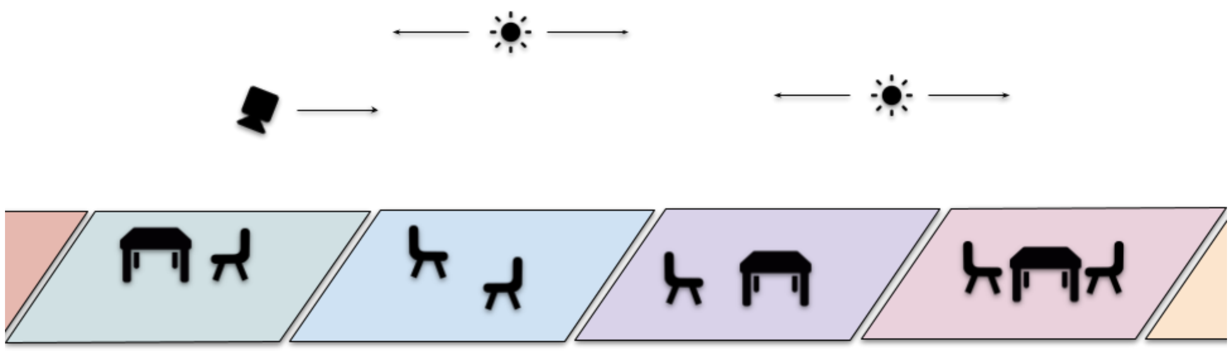


Figure 4.4: Concept of old pipeline with a plane for each setup.

Each plane was placed adjacent to the previous one, with its own texture and distractors. Each setup was manually designed by copying and positioning objects to maximize variation between setups. Camera and light trajectories also had to be defined manually, moving them across each plane along with the target objects.

The new render pipeline uses some BlenderProc functionalities to simplify the process while maintaining variation between random setups. As shown in Figure 4.5, the new pipeline uses a single plane at the world origin. Positions and orientations are randomly generated within the boundaries of the plane, and all necessary elements are placed on top of it.

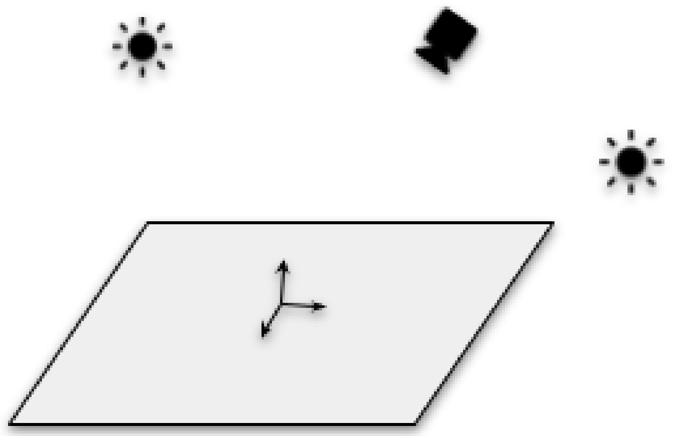


Figure 4.5: Concept of new pipeline with just a plane.

Blender allows creating animations and trajectories that are automatically reproduced while rendering. This is done using Blender keyframes [42], which store a property value for a specific frame. For instance, camera trajectory points in the old pipeline were position keyframes. Using this feature, it is also possible to modify the object visibility in the scene for each frame, allowing the selective rendering of distractors rather than rendering all of them simultaneously, as done previously.

The new Blender scene contains only a textureless plane. Other models are stored in their respective folders (e.g., `assets/models/distractors` and `assets/models/train`). Background textures are also saved in a specific folder, creating a structured file organization where each asset is stored in its respective location.

However, due to Blender's design, it is not possible to keyframe an object's material. This limitation means that a single plane cannot have different background textures for each frame. To address this, linked copies of the original plane are created, with each having assigned a unique background texture. These planes share the same mesh data as the original, consuming minimal additional memory apart from the loaded texture. All planes are positioned at the same location as the original plane, with their visibility disabled to avoid rendering overlap.

The new pipeline is designed to simplify future development. A `config.json` file specifies the desired configuration to be loaded when running the pipeline. This file includes paths for asset folders and parameters to control simulation elements such as light energy, camera resolution, and f-stop values. Additionally, it contains a whitelist and blacklist for assets, allowing users to specify which models and textures to include or exclude. By default, if the whitelist is empty, all models and textures in their respective folders are loaded. An example of how this config file looks like can be found in the appendix A.0.1

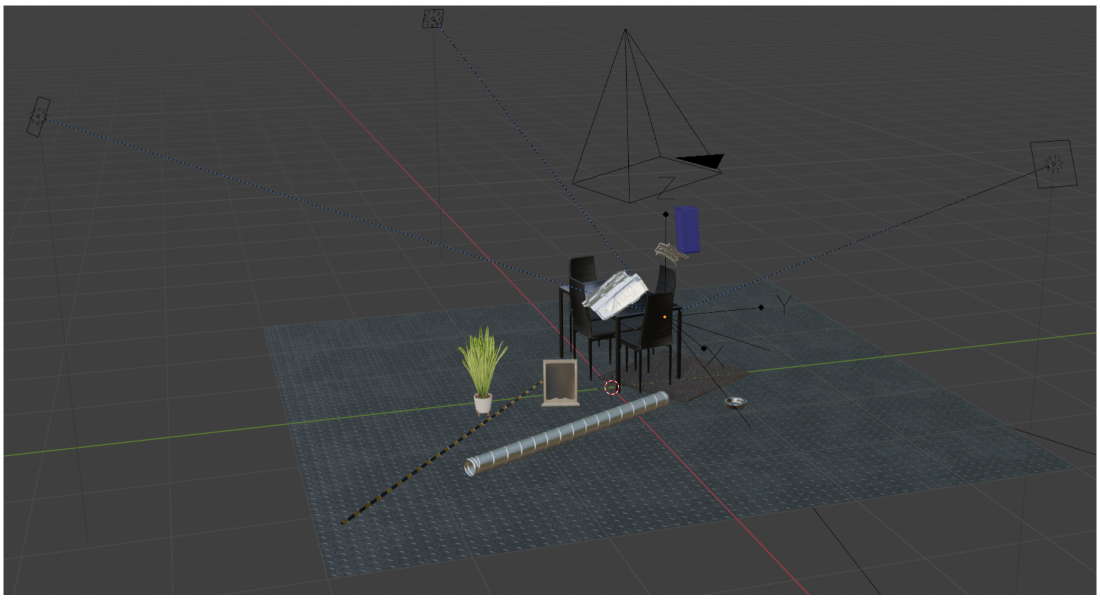
The developed Python script works the following way:

- **Load** `config.json`**:** Loads the config.json file to get the desired configuration for the pipeline.
- **Call *load_scene*:** Loads all the models and background textures included in the whitelist, while ignoring the ones in the blacklist. Disables the visibility for all the loaded models and places

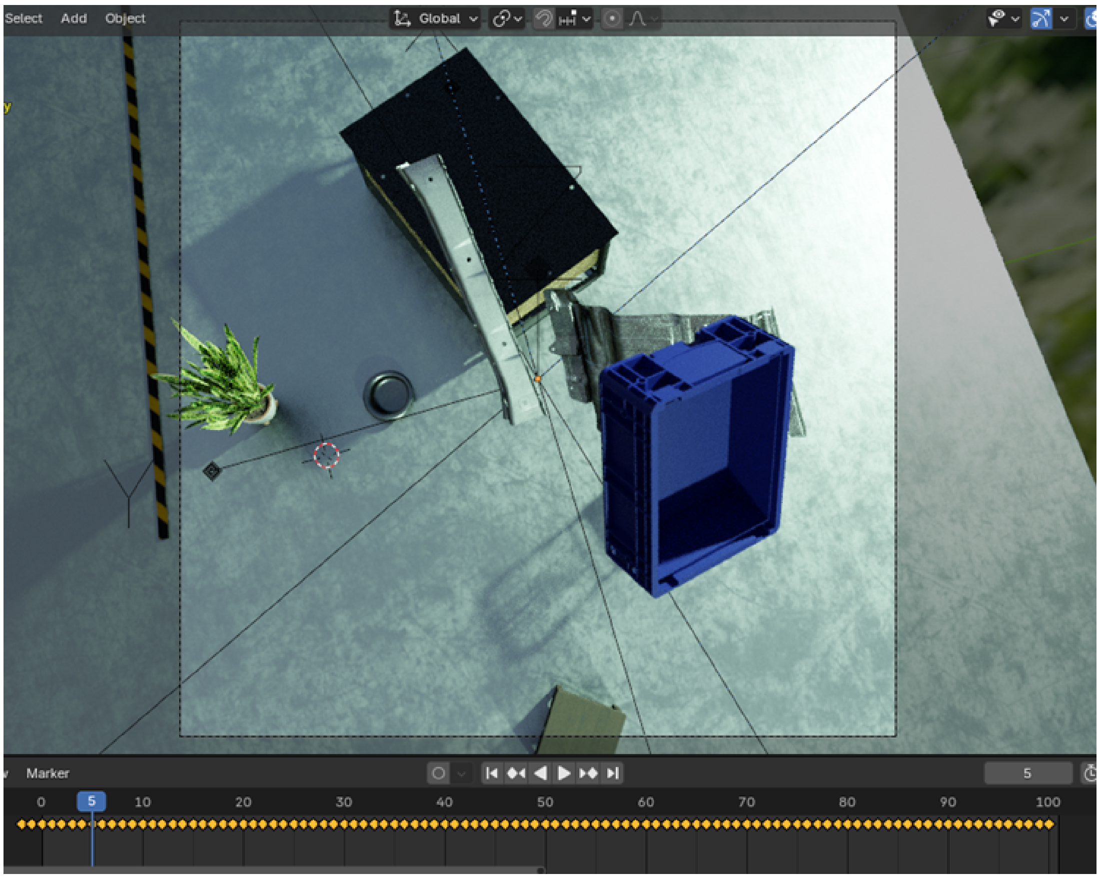
them near the original plane based on their classification as distractors or targets. Finally, it adds a custom property, *category_id*, to target models, with values between [1, `n_targets`].

- **Call *setup_scene*:** Creates linked copies of the original plane, assigning a background texture to each copy and disabling their visibility. It also sets the world background texture based on the `config.json`. Finally, it adds a three-point lighting setup and places an empty object at the world centre.
- **Call *setup_keyframes*:** Generate keyframes for each frame to be rendered. The initial setup (frame `-1`) stores all models at their default positions with visibility disabled, allowing easy scene resets. For subsequent frames (up to `max_frames`), the scene resets to frame `-1`, followed by:
  - Selecting one plane randomly and moving the empty object to a random position on it.
  - Randomly selecting and positioning distractors on the plane, ensuring a minimum spacing to avoid collisions.
  - Enabling visibility for the selected models.
  - Randomizing the power and position of area lights and moving the camera to a random position oriented towards the empty object.
  - Placing target objects orbiting around the empty object, ensuring they remain in the camera's field of view.
  - Keyframing all modified properties, including position, orientation, and visibility, for the current frame.
- **Call *render_keyframes*:** Render all keyframes. BlenderProc's custom rendering function generates RGB images, depth images, normals, and instance segmentation masks. The segmentation masks can be filtered to include specific models using the *category_id* property, producing category masks where each pixel value corresponds to the ID of the model in that pixel. Unlabeled models default to ID zero (background).

  Rendered images and annotations are stored in HDF5 files, with one file per frame. Figure 4.6 shows examples of the randomized layouts generated by the new pipeline.



(a) Far away view of a generated scene with the new pipeline.



(b) Camera viewpoint of a generated scene with the new pipeline at frame five.

Figure 4.6: Two examples of the randomized layout generated by the new pipeline. 4.6a shows the scene from far away while 4.6b shows a scene from the viewpoint of the camera as well as the keyframes at the bottom.

Although all data is stored in HDF5 format to minimize disk usage, it can also be extracted into separate folders to simplify its use with machine learning models. BlenderProc supports exporting data as a COCO-formatted dataset, utilizing RGB images and instance segmentation masks to create annotations.

4.2.2 Modifications for the boxes use case

At ARENA2036, another team is working on training a machine learning model to detect the edges of industrial boxes stacked on top of each other on a pallet. Since there was interest in creating a larger dataset, this new pipeline for synthetic data generation was utilized. However, the approach differs from the render pipeline previously described because the team is using a digital twin approach for the pallet and boxes, resulting in some key differences:

- There are no distractors in this scene, but the boxes may contain objects inside.
- The target objects are not floating on a plane but are stacked on a pallet instead.
- There is no need for planes which change the background texture.

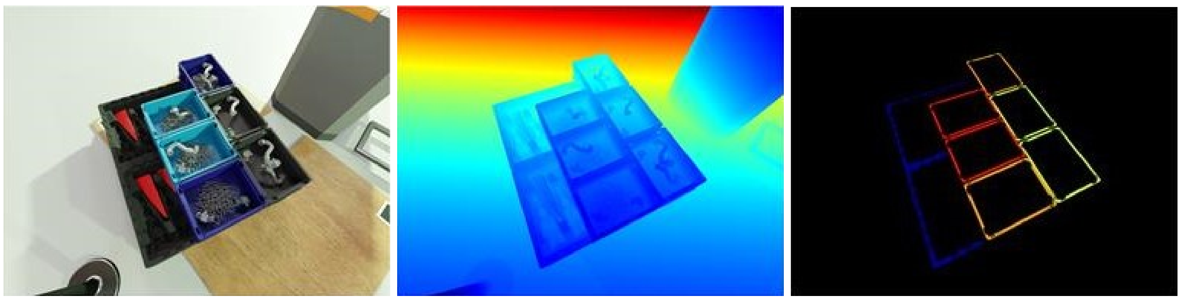
Most of the code and pipeline developed could still be reused. The only necessary changes were in *setup_scene* and *setup_keyframes*, as the boxes required additional preprocessing, and their placement in the scene now follows a specific pattern.

- **Adapted *setup_scene*:** A model for each type of box is loaded, and these models are duplicated to create enough boxes of each type to form a stack. These are linked copies, similar to the planes in the previous use case, with the colour of each box being randomized. The possible colours for each type of box are specified in the `config.json` file, which now includes an option for each type of box to be added. An example of how this config file looks like can be found in the appendix B.0.1

  Since the focus is on detecting the upper border of the boxes, the CAD models have been modified in Blender. Each box consists of the CAD model of the box itself and a second mesh corresponding to the upper border. This setup allows assigning the custom property *category_id* to this mesh so that the segmentation mask will only highlight the edge of the box instead of the entire box. An example of this can be seen in figure 4.7.

  Additionally, the boxes may contain objects inside. The original loaded box includes various objects, and for each linked copy created, some of these objects are randomly deleted. This ensures that the contents of each box are unique without requiring individual keyframing of every object inside each box for every frame.

- **Adapted *setup_keyframes*:** The randomization of the camera and lights remains the same as before. The main difference lies in the placement of the boxes on the pallet. Each level of the stack can only consist of a single specific type of box. For each level, one type of box from the loaded types is selected. Then, copies of that type of box are randomly selected and arranged to fill the pallet space. This process is repeated for each level of the stack until the final level, where not all boxes are always placed, ensuring variability. The number of levels is also randomized within the range $[1, \texttt{max\_floors}]$.



(a) RGB image of stack **(b)** Depth image of stack. **(c)** Segmentation mask of edges.

Figure 4.7: Rendered images for a stack of boxes

4.2.3 Optimizations

To enhance the efficiency and reproducibility of the synthetic data generation pipeline, several optimizations were implemented. These improvements address reproducibility, selective frame rendering, keyframe management, and parallelization, ensuring a scalable and efficient rendering process.

4.2.3.1 Reproducibility through Random Seeds

To ensure consistent results across multiple runs, random seeds were introduced for both the numpy and random modules. By setting a fixed seed value, the pipeline produces identical outputs for a given configuration, making it easier to debug, reproduce results, and fine-tune specific aspects of the simulation. This optimization is particularly useful when iterating on the same setup or troubleshooting errors in the rendering process. The seed is specified in the *config.json* file. If set to -1, the seed will also be randomised.

4.2.3.2 Interval-Based Rendering

Rendering the entire simulation repeatedly can be time-intensive, especially when only a subset of frames needs to be corrected or updated. To address this, the pipeline now supports rendering specific intervals of frames rather than the entire simulation. For instance, if an error occurs while rendering, it is now possible to re-render the last 300 frames independently, ensuring consistency with the original sequence.

This interval-based rendering capability is made possible by retaining the deterministic behaviour introduced by setting the random seeds. By recalculating the random transformations for all frames—regardless of whether they are keyframed—the pipeline ensures that the final output matches what it would have been if the entire simulation had been rendered in a single pass.

4.2.3.3 Keyframe Management

Blender's performance decreases as the number of keyframes increases, this is due to the growing complexity of the animation and Blender's internal logic for managing it. To address this, the keyframing process was optimized by limiting the addition of keyframes to only those within the specified rendering interval. This significantly reduces the computational overhead associated with managing large numbers of keyframes.

Although keyframes are added selectively, the pipeline still calculates random transformations and values for all frames in the simulation. This ensures that the deterministic behaviour of the pipeline remains intact, even if only a subset of frames is rendered. Without this step, skipping the calculation of intermediate frames would result in inconsistencies due to the influence of random functions.

4.2.3.4 Parallelised Rendering

Due to the improvement in the pipeline to make it capable of rendering specific intervals independently, it was possible to parallelise the whole pipeline to further improve performance. The parallelisation process involves dividing the total number of frames into smaller intervals and assigning each interval to a separate instance of the Python script for rendering.

To achieve GPU-specific parallelization, each script instance is assigned to a specific GPU This ensures that only a single GPU is utilized per instance, avoiding resource contention. By leveraging multiple GPUs, the pipeline can render several frame intervals simultaneously, significantly reducing the overall rendering time.

5 Testing

In this section, whether the initial requirements outlined in sections 3.2 and 3.3 have been fulfilled or not will be evaluated. While, most of the requirements are focused on making sure that specific features or functionalities are added, rather than having measurable performance metrics, they still play a critical role in achieving the objectives of the project. Because of that, most of the requirements will be addressed and discussed in detail in the discussion chapter 6, to provide insights into their implementation and impact on the overall system.

The one exception is the requirement to improve the rendering speed in the synthetic data generation pipeline as this requirement involves a measurable outcome. The following subsection details the test performed to evaluate the rendering speed of the new rendering pipeline implemented.

5.1 Rendering speed

To assess whether the new rendering pipeline fulfills Requirement *12.*, its performance was tested and compared to the previous version. The goal was to confirm that the rendering time was reduced by at least a factor of two.

The test involved rendering the same models used by the original pipeline, including distractors and background textures. The results were as follows: the original pipeline took 5.9 hours to render 1000 frames, while the new pipeline completed the same task in just 0.72 hours.

These tests were conducted before implementing the optimizations described in Section 4.2.3. The computer used for testing is equipped with four NVIDIA GTX 1080 Ti GPUs, all of which were initially utilized simultaneously. However, additional optimizations like the parallelization, enabled the assignment of specific frame intervals to each GPU, significantly improving performance.

After running again the test with the optimisations, dividing the 1000 frames into equal intervals (250 frames per GPU), each GPU rendered its assigned interval independently. This approach resulted in a total rendering time of just 0.5 hours, a substantial improvement over the original performance.

6 Discussion

6.1 Rendering speed test discussion

The results of the rendering speed test outlined in Section 5.1 demonstrate a significant improvement in performance with the new rendering pipeline. While the original pipeline required 5.9 hours to render 1000 frames (21.24 seconds per frame), the new pipeline accomplished the same task in just 0.72 hours (2.59 seconds per frame). This is eight times faster than the original version.

This huge improvement can be attributed to several key optimizations. BlenderProc allows the simultaneous generation of RGB images and annotations, eliminating the need for multiple rendering passes as required in the original pipeline. Also, the new version is much more memory efficient, as it avoids the creation and storage of hundreds of layouts in memory, a limitation of the previous approach.

Enabling and disabling the visibility of the models in the scene played a significant role in enhancing efficiency. By ensuring the render engine only processed visible objects, unnecessary computational overhead was avoided.

Additionally, after running the test with the optimisations described in 4.2.3, the rendering speed was lowered even more. Achieving a final rendering time of 0.5 hours for every 1000 frames (1.8 seconds per frame). This is **eleven times faster than the old rendering pipeline**, far exceeding the requirement of halving the rendering time of Requirement *12.*.

6.2 Remaining requirements discussion

While the remaining requirements outlined in Sections 3.2 and 3.3 do not lend themselves to direct quantitative testing they were still relevant and crucial for the development of the project.

6.2.1 Quality Inspector

The Requirement *1.* to migrate training to the cloud was successfully achieved, as detailed in Section 4.1.1. By leveraging Databricks and creating an endpoint using Ngrok, the training component was effectively moved to a cloud environment. This reduces the need to use local high-performance hardware and enhances scalability for larger datasets.

Both Requirements *2.* and *3.* were also successfully implemented as described in Section 4.1.2. The incorporation of Shapley values and kNN provides users with valuable tools for understanding and interpreting the behaviour of newly trained machine learning models.

The whole project was migrated into a single device, meeting Requirement *4.*. By using a Docker container, it became possible to run both the frontend and backend on the same device, streamlining deployment and simplifying the overall system architecture, as described in Section 4.1.3.

6.2.2 Synthetic Data Generation

The development of a fully automated rendering pipeline using Blender was successfully completed, as described in Section 4.2.1, fulfilling Requirements *1.* and *2.*. Moreover, an implementation of a real use case was also implemented in Section 4.2.2 demonstrating the potential of the pipeline.

Requirements *3.*, *4.*, *5.* and *6.* were also implemented when creating the new pipeline. A configuration file now allows users to specify how should the models be placed in the scene. This also applies to Requirements *7.*, *8.*, *9.*, *10.*, *11.*, which are more general parameters that can also be configured. Making the pipeline more flexible and scalable.

7 Conclusion

This internship at Mercedes-Benz Group AG has been a valuable experience that taught me a lot, both professionally and personally. The opportunity to work on new technologies in Industry 4.0 helped me better understand artificial intelligence, especially explainable AI and synthetic data generation. What seemed like abstract concepts before now make sense to me in practical ways.

Through this internship, I improved my technical knowledge, especially in machine learning and data engineering. In collaboration with my team, I developed and implemented solutions that integrate xAI techniques like Shapley values and k-NN, making AI models more interpretable. I also became proficient in synthetic data generation, leveraging Blender and BlenderProc to automate complex rendering tasks. This work honed my skills in Python scripting and the practical application of domain randomization techniques, both of which are invaluable in AI-driven industrial applications.

Migrating training workflows to the cloud using Databricks was another new experience for me. It exposed me to the benefits of cloud-based machine learning, including efficient data handling, cluster management, and the integration of Azure services. These experiences not only broadened my technical toolkit but also made me better appreciate scalable and sustainable AI development practices.

Personally, this internship was an amazing learning experience. The hands-on experience with Blender was particularly rewarding, as it showed me how tools like this can connect simulations to real-world applications. This involved experimenting with lighting setups, materials, and camera configurations to achieve realistic outputs. I also developed scripts to automate key aspects of the rendering process, making it more scalable and adaptable.

Future work could involve exploring other platforms for synthetic data generation, such as NVIDIA Omniverse, which offers advanced multi-GPU rendering capabilities to further enhance pipeline efficiency. Generative AI techniques could also be leveraged to improve the quality and diversity of synthetic datasets, such as using diffusion models for generating more realistic backgrounds or objects. Additionally, extending the pipeline to other industrial domains, such as aerospace for defect detection in aircraft components, electronics manufacturing for inspecting circuit boards and solder joints, or logistics for optimizing warehouse operations using synthetic environments, could open new opportunities for broader applications of these technologies.

Overall, this internship was a great learning experience that boosted my skills and helped me to shape my career goals.

Bibliography

[1] C. Bai, P. Dallasega, G. Orzes, and J. Sarkis, Industry 4.0 technologies assessment: A sustainability perspective, *International Journal of Production Economics*, vol. 229, p. 107 776, 2020, ISSN: 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2020.107776.
URL: https://www.sciencedirect.com/science/article/pii/S0925527320301559.

[2] S. Vaidya, P. Ambad, and S. Bhosle, Industry 4.0 – a glimpse, *Procedia Manufacturing*, vol. 20, pp. 233–238, 2018, 2nd International Conference on Materials, Manufacturing and Design Engineering (iCMMD2017), 11-12 December 2017, MIT Aurangabad, Maharashtra, INDIA, ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.02.034.
URL: https://www.sciencedirect.com/science/article/pii/S2351978918300672.

[3] N. Kurniati, R.-H. Yeh, and J.-J. Lin, Quality inspection and maintenance: The framework of interaction, *Procedia Manufacturing*, vol. 4, pp. 244–251, 2015, Industrial Engineering and Service Science 2015, IESS 2015, ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2015.11.038.
URL: https://www.sciencedirect.com/science/article/pii/S2351978915011543.

[4] R. Damgrave and E. Lutters, Synthetic prototype environment for industry 4.0 testbeds, *Procedia CIRP*, vol. 91, pp. 516–521, 2020, Enhancing design through the 4th Industrial Revolution Thinking, ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2020.02.208.
URL: https://www.sciencedirect.com/science/article/pii/S2212827120308593.

[5] Mercedes-benz group. about us.
URL: https://www.mbusa.com/en/about-us, (Retrieved: 19-12-2024).

[6] Mercedes-benz group. company at glance.
URL: https://group.mercedes-benz.com/company/at-a-glance.html, (Retrieved: 19-12-2024).

[7] T.-Y. Lin, M. Maire, S. Belongie, et al., Microsoft coco: Common objects in context, in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., Cham: Springer International Publishing, 2014, pp. 740–755.

[8] Q. Demlehner and S. Laumer, Shall we use it or not? explaining the adoption of artificial intelligence for car manufacturing purposes, in *European Conference on Information Systems*, 2020.
URL: https://api.semanticscholar.org/CorpusID:219416061.

[9] P. De Roovere, S. Moonen, N. Michiels, and F. wyffels, Sim-to-real dataset of industrial metal objects, *Machines*, vol. 12, no. 2, 2024, ISSN: 2075-1702. DOI: 10.3390/machines12020099.
URL: https://www.mdpi.com/2075-1702/12/2/99.

[10] P. D. Roovere, S. Moonen, N. Michiels, and F. Wyffels, Dataset of industrial metal objects, 2022. arXiv: 2208.04052 [cs.CV].
URL: https://arxiv.org/abs/2208.04052.

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, You only look once: Unified, real-time object detection, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[12] M. K. Mihçak and R. Venkatesan, New iterative geometric methods for robust perceptual image hashing, in *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, ser. DRM '01, Berlin, Heidelberg: Springer-Verlag, 2001, pp. 13–21, ISBN: 3540436774.

[13] Foundation, t.b.: Blender 4.0, 2023.
URL: https://group.mercedes-benz.com/company/at-a-glance.html, (Retrieved: 03-01-2025).

[14] Foundation, t.b.: The cycles render engine, 2023.
URL: https://projects.blender.org/blender/cycles, (Retrieved: 03-01-2025).

[15] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, Signature verification using a "siamese" time delay neural network, in *Proceedings of the 7th International Conference on Neural Information Processing Systems*, ser. NIPS'93, Denver, Colorado: Morgan Kaufmann Publishers Inc., 1993, pp. 737–744.

[16] T. Lesort, N. Díaz-Rodríguez, J.-F. Goudou, and D. Filliat, State representation learning for control: An overview, *Neural Networks*, vol. 108, pp. 379–392, Dec. 2018, ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.07.006.
URL: http://dx.doi.org/10.1016/j.neunet.2018.07.006.

[17] S. Chopra, R. Hadsell, and Y. LeCun, Learning a similarity metric discriminatively, with application to face verification, in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, 539–546 vol. 1. DOI: 10.1109/CVPR.2005.202.

[18] E. Hoffer and N. Ailon, Deep metric learning using triplet network, 2018. arXiv: 1412.6622 [cs.LG].
URL: https://arxiv.org/abs/1412.6622.

[19] P. Sermanet, C. Lynch, Y. Chebotar, et al., Time-contrastive networks: Self-supervised learning from video, 2018. arXiv: 1704.06888 [cs.CV].
URL: https://arxiv.org/abs/1704.06888.

[20] R. Goroshin, M. Mathieu, and Y. LeCun, Learning to linearize under uncertainty, 2015. arXiv: 1506.03011 [cs.CV].
URL: https://arxiv.org/abs/1506.03011.

[21] T. Miller, Explanation in artificial intelligence: Insights from the social sciences, *Artificial Intelligence*, vol. 267, pp. 1–38, 2019, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2018.07.007.
URL: https://www.sciencedirect.com/science/article/pii/S0004370218305988.

[22] F. Doshi-Velez and B. Kim, Towards a rigorous science of interpretable machine learning, 2017. arXiv: 1702.08608 [stat.ML].
URL: https://arxiv.org/abs/1702.08608.

[23] P. Thisovithan, H. Aththanayake, D. Meddage, I. Ekanayake, and U. Rathnayake, A novel explainable ai-based approach to estimate the natural period of vibration of masonry infill reinforced concrete frame structures using different machine learning techniques, *Results in Engineering*, vol. 19, p. 101 388, 2023, ISSN: 2590-1230. DOI: https://doi.org/10.1016/j.rineng.2023.101388.
URL: https://www.sciencedirect.com/science/article/pii/S2590123023005157.

[24] M. Louhichi, R. Nesmaoui, M. Mbarek, and M. Lazaar, Shapley values for explaining the black box nature of machine learning model clustering, *Procedia Computer Science*, vol. 220, pp. 806–811, 2023, The 14th International Conference on Ambient Systems, Networks and Technologies Networks (ANT) and The 6th International Conference on Emerging Data and Industry 4.0 (EDI40), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2023.03.107.
URL: https://www.sciencedirect.com/science/article/pii/S1877050923006427.

[25] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, Grad-cam: Visual explanations from deep networks via gradient-based localization, *International Journal of Computer Vision*, vol. 128, no. 2, pp. 336–359, Oct. 2019, ISSN: 1573-1405. DOI: 10.1007/

s11263-019-01228-7.
URL: http://dx.doi.org/10.1007/s11263-019-01228-7.

[26] M. I. Jordan and T. M. Mitchell, Machine learning: Trends, perspectives, and prospects, *Science*, vol. 349, no. 6245, pp. 255–260, 2015. DOI: 10.1126/science.aaa8415. eprint: https://www.science.org/doi/pdf/10.1126/science.aaa8415.
URL: https://www.science.org/doi/abs/10.1126/science.aaa8415.

[27] L. L. Pipino, Y. W. Lee, and R. Y. Wang, Data quality assessment, *Commun. ACM*, vol. 45, no. 4, pp. 211–218, Apr. 2002, ISSN: 0001-0782. DOI: 10.1145/505248.506010.
URL: https://doi.org/10.1145/505248.506010.

[28] R. Babbar and B. Schölkopf, Data scarcity, robustness and extreme multi-label classification, *Mach. Learn.*, vol. 108, no. 8–9, pp. 1329–1351, Sep. 2019, ISSN: 0885-6125. DOI: 10.1007/s10994-019-05791-5.
URL: https://doi.org/10.1007/s10994-019-05791-5.

[29] Y. Lu, M. Shen, H. Wang, et al., Machine learning for synthetic data generation: A review, 2024. arXiv: 2302.04062 [cs.LG].
URL: https://arxiv.org/abs/2302.04062.

[30] J. Borrego, A. Dehban, R. Figueiredo, P. Moreno, A. Bernardino, and J. Santos-Victor, Applying domain randomization to synthetic data for object category detection, 2018. arXiv: 1807.09834 [cs.CV].
URL: https://arxiv.org/abs/1807.09834.

[31] D. Horváth, G. Erdős, Z. Istenes, T. Horváth, and S. Földi, Object detection using sim2real domain randomization for robotic applications, *IEEE Transactions on Robotics*, vol. 39, no. 2, pp. 1225–1243, Apr. 2023, ISSN: 1941-0468. DOI: 10.1109/tro.2022.3207619.
URL: http://dx.doi.org/10.1109/TRO.2022.3207619.

[32] C. Mayershofer, T. Ge, and J. Fottner, Towards fully-synthetic training for industrial applications, in Springer, 2021, pp. 765–782.
URL: https://EconPapers.repec.org/RePEc:spr:sprchp:978-981-33-4359-7_53.

[33] C. Heindl, L. Brunner, S. Zambal, and J. Scharinger, Blendtorch: A real-time, adaptive domain randomization library, in *Pattern Recognition. ICPR International Workshops and Challenges: Virtual Event, January 10–15, 2021, Proceedings, Part IV*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 538–551, ISBN: 978-3-030-68798-4. DOI: 10.1007/978-3-030-68799-1_39.
URL: https://doi.org/10.1007/978-3-030-68799-1_39.

[34] M. Denninger, M. Sundermeyer, D. Winkelbauer, et al., Blenderproc, 2019. arXiv: 1911.01911.
URL: https://arxiv.org/abs/1911.01911.

[35] Foundation, t.b.: The cycles render engine.
URL: https://docs.blender.org/api/current/index.html, (Retrieved: 03-01-2025).

[36] T. H. Group, Hierarchical data format.
URL: https://www.hdfgroup.org/solutions/hdf5/, (Retrieved: 03-01-2025).

[37] Microsoft, What is azure databricks.
URL: https://learn.microsoft.com/en-us/azure/databricks/introduction/, (Retrieved: 03-01-2025).

[38] Microsoft, Databricks clusters.
URL: https://learn.microsoft.com/en-us/azure/databricks/compute/clusters-manage, (Retrieved: 03-01-2025).

[39] Databricks sdk for python (beta).
URL: https://github.com/databricks/databricks-sdk-py/tree/main, (Retrieved: 04-01-2025).

[40] Ngrok docs.
URL: https://ngrok.com/docs, (Retrieved: 03-01-2025).

[41] Shap github.
URL: https://github.com/shap/shap, (Retrieved: 04-01-2025).

[42] About blender keyframes.
URL: https://docs.blender.org/manual/en/latest/animation/keyframes/introduction.html, (Retrieved: 04-01-2025).

Appendices

A Example config.json file

```
"blender_world_path": "/media/.../boxes_scene.blend",
"target_models_dir": "/media/.../assets/train/",
"distraction_models_dir": "/media/.../assets/train/",
"backgrounds_dir": "/media/.../assets/backgrounds/",

"random_seed": 0,
"environmental_light": [0.1, 1],
"whitelist_targets": [],
"blacklist_targets": [],
"whitelist_distractors": [],
"blacklist_distractors": [],
"whitelist_backgrounds": [],
"blacklist_backgrounds": [],

"camera":{
    "resolution": [512,512],
    "fstop": [4,16],
    "distance_empty_object": [0.5,1.5]
},
"lights":{
    "nlights": 3,
    "power": [0, 500],
    "randomize_color": 0
},
"planes":{
    "origin": [0,0,0],
    "surface": [6,6]
},
"distractors":{
    "amount": [0, 10],
    "max_dist_center": [3,3,0],
    "min_rotation": [0,0,0],
    "max_rotation": [0,0,360]
},
"targets":{
    "dist_empty_object": [0.1, 0.5],
    "min_rotation": [0,0,0],
    "max_rotation": [360,360,360]
}
```

Snippet A.0.1: Example of config.json file for configuring the rendering.

B Example boxes config.json file

```
"blender_world_path": "/media/.../boxes_scene.blend",
"target_models_dir": "/media/.../assets/train/",
"distraction_models_dir": "/media/.../assets/train/",
"max_levels": 6,
"environmental_light": [0.1, 1],
"whitelist_targets": [],
"blacklist_targets": [],

"camera":{
    "resolution": [512,512],
    "fstop": [4,16],
    "distance_stack": [0.5,1.5],
},
"lights":{
    "power": [0, 500],
    "randomize_color": 0
},
"boxes":{
    "small_box":{
        "path": "/media/.../assets/train/small_box.blend",
        "colors":{
            "blue": [0.0024, 0.0, 0.13568, 1.0],
            "light_blue": [0.0, 0.1926, 0.4070, 1.0],
            "black": [0.0204, 0.0204, 0.0204, 1.0]
        }
    },
    "big_box":{
        "path": "/media/.../assets/train/big_box.blend",
        "colors":{
            "blue": [0.0024, 0.0, 0.13568, 1.0],
            "light_blue": [0.0, 0.1926, 0.4070, 1.0],
            "black": [0.0204, 0.0204, 0.0204, 1.0]
        }
    },
    "lights_box":{
        "path": "/media/.../assets/train/lights_box.blend",
        "colors":{
            "black": [0.0204, 0.0204, 0.0204, 1.0]
        }
    }
}
```

Snippet B.0.1: Example of the config.json file for configuring the rendering used in the boxes use case.