Open-Source Pipeline for Synthetic Data Generation in Industrial Applications

Master's thesis report

ADRIÁN SANCHIS REIG

Aalborg Universitet
Det Tekniske Fakultet for IT og Design

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

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Open-Source Pipeline for Synthetic Data Generation in Industrial Applications

Theme:
Master's thesis project

Project Period:
Spring Semester 2025

Project Group:
None

Participant(s):
Adrián Sanchis Reig

Supervisor(s):
Dimitris Chrysostomou,
Jose Moises Araya Martinez

Page Numbers: 89

Date of Completion:
04.06. 2025

Abstract:

This thesis report outlines the development of a synthetic data generation pipeline to support industrial manufacturing processes. Conducted in collaboration with Mercedes-Benz Group AG, the thesis focuses on automating synthetic data generation workflows using the Blender render engine Cycles. The report explores the use of techniques such as Domain Randomisation and Guided Domain Randomisation to narrow the sim-to-real gap.
Key aspects of the thesis include the use of Blender and its Python API for designing and implementing an optimised rendering pipeline which automates the generation of photorealistic scenes in industrial environments and their corresponding materials. Data augmentation and filtering are implemented as well using Stable Diffusion XL and by comparing high and low image level features.

Contents

Preface

Many thanks to my supervisors Dimitris Chrysostomou and Jose Moises Araya Martinez for their support and guidance.

Moreover, many thanks to Matthias Reichenbach, Florian Toeper, Sarvenaz Sardari, Jan Niklas Ewertz and the rest of the team in Arena2036 for making this an amazing year and experience.

The code developed during the whole thesis can be found in the following private GitHub repository: https://github.com/Adriagent/Thesis_SyntheticPipeline.git

Adrián Sanchis Reig
asanch23@student.aau.dk

Chapter 1

Introduction

Vison-based Deep Learning models have propelled the automation of many industrial tasks, such as robotic-based material handling and industrial quality inspection. However, training and testing a deep neural network is a time-consuming and expensive task that typically involves collecting and manually annotating large amounts of data due to its supervised learning nature. Data availability also plays an important role in dataset creation, as it can quickly increase the time, effort, and expenses invested in ensuring a considerable amount of high-quality data. In addition, the low scalability of collected data can also be a problem if its scope is only valid for a specific use case or if some changes are needed that make the part of the data impossible to use. These problems become more evident when the needed data falls within industrial applications, as data can be scarce and difficult to obtain [1]. For instance, it is fair to assume that Computer-Aided Design (CAD) availability is granted in most industrial settings. But not to assume that it will be possible to collect and annotate hundreds of thousands of images in different configurations.

An increasingly popular approach to mitigate these limitations is the use of synthetically generated data to replace partially or entirely real-world data, which can help overcome these problems while improving time and cost expenses [2][3]. Synthetic data leverages advanced graphic simulators [4][5][6], allowing the creation of a virtual environment where training examples can be produced. Such simulations are scalable and easily customisable, avoiding the challenges of collecting and annotating data in the real world. Moreover, they allow fine-grained control over scene parameters, enabling the generation of diverse and challenging corner-case scenarios that might be difficult or impossible to replicate consistently in the real world. The rendering engines used in these simulations inherently possess comprehensive scene information, facilitating automatic annotation, and significantly reducing manual intervention. Additionally, virtual environments offer reusability and adaptability, allowing rapid dataset iteration in response to changing requirements.

However, synthetic data suffers from discrepancies when compared to real-world data, mainly in the photorealistic aspect [6]. Image renderings often fail to replicate the richness, complexity, and noise characteristics intrinsic to real-world images [6]. This mismatch, known as the reality gap, is a significant challenge, since models trained purely on synthetic data may struggle to generalise effectively when deployed in actual industrial environments.

To bridge this reality gap, Domain Randomisation (DR) has emerged as a strategy to improve model generalisation by increasing the diversity of synthetic data. DR works by randomising simulation

parameters, such as lighting conditions, object textures, and camera viewpoints to expose the model to a wide range of scenarios during training. By learning a more diverse dataset, models can generalise better, effectively transferring learned representations from simulated to real-world settings without the need to add real data [6]. The use of DR can not only enhance the robustness of the trained model but also reduce dependence on real-world data.

Another complementary approach to bridging the reality gap involves Guided Domain Randomisation (GDR) techniques. GDR focuses on leveraging limited amounts of real-world data to fine-tune trained models by aligning feature distributions between simulated and real-world domains [7]. By minimising domain discrepancies, DA techniques help to further improve the robustness of detection models.

In summary, leveraging synthetic data generation combined with domain randomisation and adaptation strategies provides a promising solution to address the challenge of data scarcity, cost, and scalability in vision-based deep learning models for industrial applications. This thesis proposes a pipeline that makes use of these techniques in order to automate the process of training detection models while bridging the reality gap.

1.1 Collaboration with Mercedes-Benz Group AG

This thesis is a continuation of the project-oriented internship conducted during the previous semester in collaboration with Mercedes-Benz Group AG in Stuttgart, Germany. The previous work consisted of optimising synthetic data pipelines and explainable AI for industrial applications [8].

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 [9]. Today, the group operates in 17 countries on 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 [10].

The project aimed to collaborate with their team at the ARENA2036 e.V. research campus, which focuses on applying cutting-edge technologies in production systems and quality assurance. These efforts are primarily focused 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 digitalisation. 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 in 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 thesis, 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.

Chapter 2

Problem Analysis

This chapter provides a comprehensive analysis of the project undertaken in this thesis. The main objectives and tasks are defined, followed by an examination of key challenges and a survey of related work. The central focus of the thesis lies in utilising render-based techniques to automate and optimize synthetic data generation workflows for industrial applications.

A pipeline for producing synthetic training datasets is developed and its effectiveness on object detection tasks is evaluated. To assess data quality, a state-of-the-art object detection model is trained and tested on both synthetic and real-world images. Performance metrics are then compared to quantify the utility of the generated data.

2.1 Challenges in industrial object detection

Object detection in industrial environments involves the identification and localisation of specific components, defects, or products within complex scenes. Common applications include:

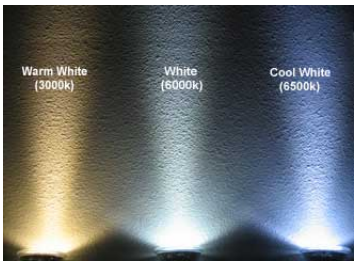
- **Defect inspection:** Detecting surface imperfections, cracks, or anomalies in manufactured items.
- **Pick-and-place automation:** Locating parts on conveyor belts for robotic handling.
- **Inventory management:** Recognising and counting items in storage or on shelves.

On the other hand, object detection presents unique challenges, the most common including clutter, occlusion, perception noise, and illumination variations.

- **Clutter:** In industrial scenes it is common that numerous elements and overlapping parts are present in the image. Clutter refers to non-target objects with similar shapes, sizes, textures or colours that confuse the detector. Figure 2.1 depicts an example of a cluttered image in which it is difficult to perform an object detection task.
- **Occlusion:** Targets may be partially hidden by other objects, machinery or elements in the scene. This makes it difficult for the detection to perceive and interpret the target and can lead to perception noise. Even slight occlusions can obscure crucial features such as edges or markings, resulting in missed detections or inaccurate bounding boxes.
- **Illumination variation:** Lighting variations in factories can vary in intensity and colour among others, affecting shadows, reflections and object appearance.

(a) Example of clutter

(b) Example of occlusion [11]



(c) Example of light variation

Figure 2.1: Examples of most common challenges (clutter, occlusion and illumination variation)

Robust detection in industrial conditions can be critical, as small localisation errors can lead to downstream failures in automation or quality control. However, acquiring and labelling real-world industrial images is expensive and time-consuming.

State-of-the-art deep learning methods for object detection include two-stage detectors such as Faster R-CNN and single-stage detectors like YOLO and SSD. These models have demonstrated high accuracy on standard benchmarks (e.g., MS COCO), but their performance often degrades when applied to industrial domains due to domain shifts between training and target data.

Leveraging synthetic data can mitigate annotation bottlenecks and reduce domain gaps if generated samples capture the variability and complexity of real scenes. In subsequent sections, the design of a synthetic data generation pipeline tailored to industrial object detection will be detailed, followed by evaluation results comparing model performance on synthetic versus real datasets.

2.2 Related work

This section will introduce relevant works and background for this project to better understand its key aspects, focusing on object detection, domain randomization/adaptation and data augmentation. The first subsection analyses the possible object detection methods and evaluation metrics. The second subsection focuses on the use of domain randomisation and adaptation techniques for closing the gap between real and synthetic domains. The last subsection will introduce data augmentation techniques using diffusion models.

2.2.1 Object Detection

The object detection task is classified into two main approaches: traditional CV-based methods and Deep Learning (DL)-based methods. Although traditional methods can result in high accuracy, they are not robust enough to appearance variations and require a significant amount of effort to design and optimise feature extractors, object proposals, and their classes [12]. In recent years, DL-based approaches using Convolutional Neural Networks (CNN) have shown state-of-the-art performance in general object detection tasks, removing the need for hand-designed feature extractors and object proposals. These methods, however, typically demand large volumes of fully annotated training data, which is often costly and time-consuming [13]. Synthetic data offers a practical solution by supplying abundant, automatically labelled examples.

Due to the aforementioned reasons, the scope of this thesis will only focus on DL-based methods. DL-based detectors can broadly grouped into two paradigms: CNN-based models and transformed-based (DETR) models. The first group can also be divided into one or two-stage detectors:

- **One-stage detectors** consider object detection as a regression problem, hence using a unified framework for learning the probabilities of the classes and the coordinates of the bounding boxes.
- **Two-stage detectors** use region proposal networks to produce regions of interest (ROI) in which the targets may be found, and apply deep neural networks to classify each proposal into class categories.

It is important to mention that the approach used by one-stage detectors makes them faster than their counterparts two-stage detectors.

2.2.1.1 Faster R-CNN

Faster Region-based Convolutional Neural Network (Faster R-CNN) [14] is an extension of Fast R-CNN [15] for object detection. As a two-stage detector, it is composed of two modules: a deep fully convolutional network, that proposes regions, known as a Region Proposal Network (RPN); and the Fast Region-based Convolutional Neural Network (R-CNN) [15] that uses these proposed regions. The entire system is a single, unified network for object detection as depicted in Figure 2.2.

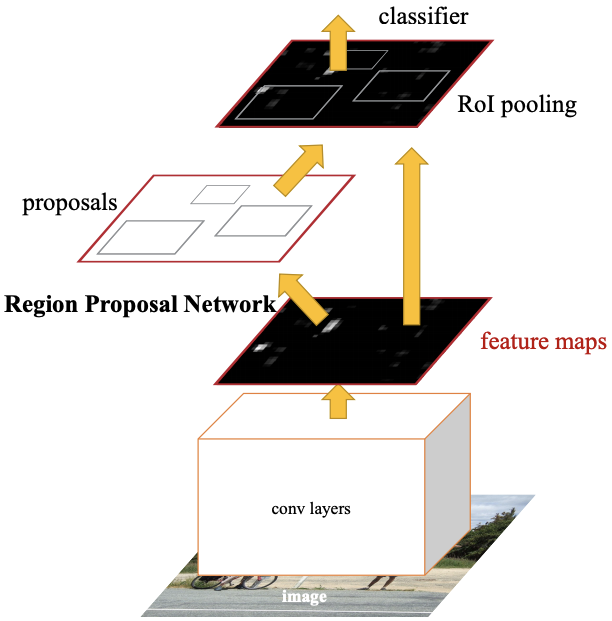


Figure 2.2: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the 'attention' of this unified network [14].

The RPN generates potential object regions (proposals) using anchors, which are predefined bounding boxes of various scales and aspect rations, and then passed to the Fast R-CNN module. This last one classifies the objects and adjusts the bounding box coordinates. By using feature maps from the convolutional layers, Faster R-CNN efficiently detects objects with high accuracy, but its two-stage process introduces latency compared to single-stage models, making it challenging to achieve real-time performance due to its complexity.

2.2.1.2 Single Shot Multibox Detector (SSD)

SSD [16] was proposed as a fast and accurate alternative for object detection tasks to Faster R-CNN, which due to its two-stage approach of calculating region proposals was too slow for real-time infer-

ence (7 FPS on the VOC2007 test [16]). Instead, SSD presented a network that does not resample pixels or features of bounding box hypotheses. It utilises a single-shot prediction approach for both classification and location, eliminating the need for time-consuming region proposals.

This model uses feature pyramids to predict objects at different scales from multiple layers of the network (see Figure 2.3). SSD uses fixed-sized bounding boxes, also called default boxes, which are predefined to capture objects of various sizes and aspect ratios. This multi-scale detection method makes SSD much faster than many two-stage models while maintaining its competitive accuracy.

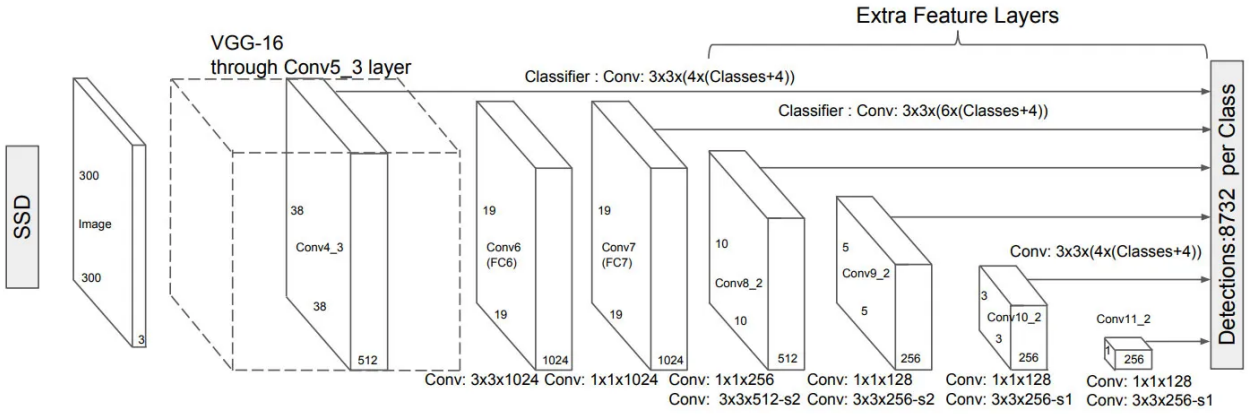


Figure 2.3: SSD model adds several feature layers to the end of a base network, which predicts the offsets to default boxes of different scales and aspect ratios and their associated confidences [16].

2.2.1.3 You Only Look Once (YOLO)

YOLO is a single-stage detection model introduced by Joseph Redmon et al. [17] in 2016. It is widely used due to its ease of use, speed and precision. Since the original model was released in 2016, it has constantly been developed by many teams, improving the architecture of the model to reach state-of-the-art performance.

The model divides the input image into a grid and predicts bounding boxes and class scores for each cell in one shot. Successive versions (YOLOv2, YOLOv3, etc.) have introduced improved backbones, anchor strategies, and loss functions to enhance both speed and precision.

2.2.1.4 DETR models

Transformer-based detectors (DETR) integrate transformer architectures, traditionally used in natural language processing, into visual recognition tasks. Introduced by Facebook AI in 2020, DETR presents a novel approach by treating object detection as a direct set prediction problem, eliminating the need for traditional components like anchor boxes and complex post-processing steps such as Non-Maximum Suppression (NMS)[18]. At its core, DETR uses a standard CNN backbone, typically ResNet-50, for initial feature extraction. This is followed by a transformer that consists of an encoder and a decoder where the encoder processes the spatial features across the image and the decoder uses learned object queries to predict the presence of objects along with their categories and bounding boxes.

An example of a DETR model is RF-DETR, developed by Roboflow [19], is a real-time DETR that outperforms models like YOLOv11 and LW-DETR on benchmarks such as COCO and RF100-V [18].

2.2.1.5 Evaluation metrics

During the training phase, several evaluation metrics are employed to assess the performance of the trained models. The most popular ones are: precision, recall, F1 score, Intersection over Union and Mean average precision [20][21].

Intersection over Union (IoU): Evaluates the degree of overlap between two areas: being the first the intersection between the predicted bounding box and the ground truth bounding box; and the second the union between the predicted bounding box and the ground truth bounding box [22].

$$IoU = \frac{area(B_p \cap B_t)}{area(B_p \cup B_t)}$$

wherein: B_p = predicted bounding box, B_t = ground truth bounding box.

Applying a threshold α, IoU allows to determine if the detection is correct. The detection is classified as:

$$\begin{gathered}\text{True Positive: } (TP) \text{ if } IoU > \alpha \\ \text{False Positive: } (FP) \text{ if } IoU < \alpha \\ \text{False Negative: } (FN) \text{ if } IoU = 0.\end{gathered}$$

Figure 2.4 shows an example of how IoU can be used for determining a true positive (TP), a false positive (FP) or a false negative (FN) based on a threshold α of 0.6.

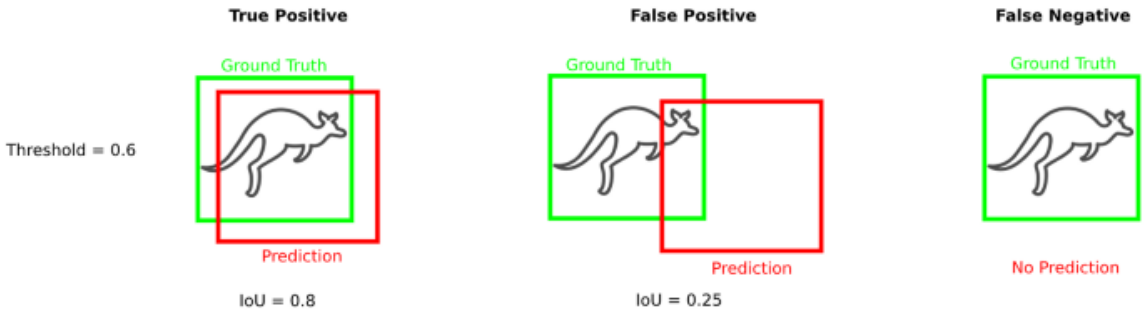


Figure 2.4: IoU threshold examples [22]

The IoU metric ranges between 0 and 1, where 0 shows no overlap and 1 shows the perfect overlap between the ground truth and the predicted area. It serves as a metric to assess the accuracy of a predicted bounding box in comparison to the ground truth bounding box.

Precision (P): In object detection, precision is directly related to the IoU measure metric. It is defined as the ratio of the *correctly* predicted positives divided by the total predicted positives. The higher the precision, the higher the proportion of true positives (TP) detected.

$$P = \frac{TP}{TP + FP}$$

wherein: TP = true positives, FP = false positives.

Recall (R): In object detection, precision is directly related to the IoU measure metric. It is defined as the ratio of correctly predicted positives divided by the total of actual positives. The higher the recall, the fewer actual positives that the model missed.

$$R = \frac{TP}{TP + FN}$$

wherein: TP = true positives, FN = false negatives.

F1 score ($F1$): Is defined as the harmonic mean of precision and recall. It shows precision and recall in balance for a model, and is used when there is a need to find the equilibrium between precision and recall.

$$F1 = 2 \times \frac{P \times R}{P + R}$$

wherein: $F1$ = F1 score, P = precision, R = recall.

Mean average precision (mAP): Is defined as the average precision across multiple classes and threshold values, serving as a key metric to evaluate the performance of object detection models.

$$mAP = \frac{1}{N} \times \sum_{i=1}^{N} AP_i$$

wherein: N = number of classes, AP_i = average precision for class i.

The mAPα notation is used in many data evaluation challenges, one of the most popular being the COCO data evaluation challenge referred to as COCO metrics [23]. In COCO metrics, the primary challenge metric mAP for object classes is evaluated at different IoU thresholds (α), ranging from 0.5 to 0.95 in 0.05 increments [23].

2.2.2 Synthetic Data

Synthetic data generation refers to the process of creating artificial data in order to replicate the statistical properties and complexities of real-world datasets [2]. This approach is getting more attention in the field of machine learning since the process of collecting and annotating data is both time-consuming and expensive [24]. 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 [25].
- **Data scarcity** is another relevant challenge. Data is not always easily available or the number of accessible datasets is insufficient [26].
- **Data privacy** is a challenge which is solved as soon as synthetic data is used rather than real. 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 modelling 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 [3].

One of the key strengths 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 simulating extreme weather conditions or complex traffic scenarios can be easier than finding them in reality.

However, synthetic data suffers from discrepancies when compared to real-world data, mainly in the photorealistic aspect. Image renders often fail to replicate the richness, complexity, and noise characteristic of real-world images [6]. This mismatch is known as the simulation-to-reality gap or sim-to-real gap and is the main reason why synthetic data has not yet been broadly adopted as a replacement for dataset creation.

Models trained purely on synthetic data may struggle to generalise effectively when deployed in actual environments. In order to bridge this reality gap, methods like domain (DR), and domain adaptation (DA) have emerged as strategies to better generalise the simulated data.

2.2.2.1 Domain Randomization

In his work, Tobin et al. [6] make use of low-quality renders optimised for speed and not carefully matched to real-world textures, lightning, and scene configurations. That is, 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 [27]. This strategy allows models to reduce the ”reality gap” by training on synthetic images that incorporate a big range of randomised parameters.

Following the definition from [28], the key idea behind DR is that a real domain can be covered under the broad distribution of a random one. Let the environment to which we have full access (i.e. simulator) be called *source domain* and the environment that we would like to transfer the model to *target domain* (i.e., real world). Training happens in the source domain, wherein a set of N randomisation parameters can be controlled.

Synthetic data from de source domain is generated with a randomisation applied. By doing so, the trained model is exposed to a variety of environments and learns to generalise as the target domain becomes a subset of the source one (see figure 2.5). For instance, variations in lighting, textures and/or object positions ensure that models learn from robust and invariant features rather than just visual cues.

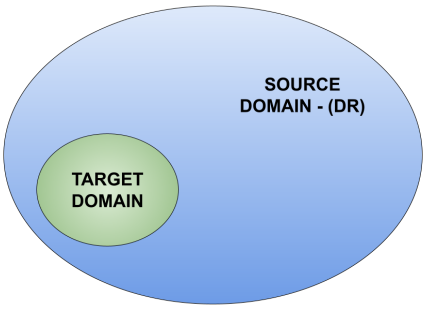


Figure 2.5: DR: the target domain (i.e. real world) is a subset within the source domain (i.e. simulation) due to the wide range of random variations applied.

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 so that no corner case falls outside of the randomised domain. For instance, in object detection tasks, randomising the background textures, camera angles and lighting conditions can help mitigate overfitting and improve performance in the real world.

Several research groups have already performed ablation studies to investigate the effects of randomising various parameters. These include the aforementioned (camera angles, lighting, textures) and also flying distractors, and random noise [27].

2.2.2.2 Guided Domain Randomization

DR assumes no access to real data, making the randomisation as broad and uniform as possible in the simulation and hoping that the target domain falls within the broad distribution of the source domain. However, this approach can easily be improved if real information from the target domain is available, providing the simulation with guidance that can improve its performance and quality.

This is called Guided Domain Randomisation (GDR) [28] and can be used to save computation resources by avoiding training models in unrealistic environments (see figure 2.6. Another benefit is to avoid infeasible solutions that might arise from overly wide randomisation distributions and thus might hinder successful learning. In this context, GDR introduces finner constraints and real-world data distributions to guide the randomisation process.

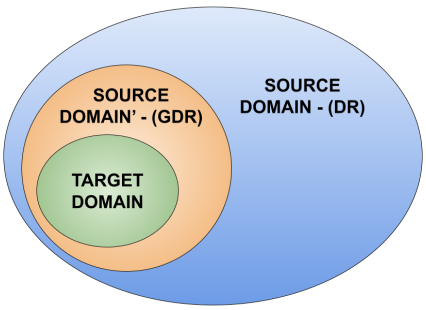


Figure 2.6: GDR: the target domain (i.e. real world) is a subset within the source domain' (i.e. simulation) which due to real-world data is narrower than the DR one.

By following this approach, the generated synthetic data remains not only diverse but also relevant to the specific task and domain being targeted. Moreover, GDR narrows the gap between synthetic and real-world data.

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 GDR frameworks to dynamically optimize training data generation. These mechanisms evaluate the performance of the model on validation or test data and iteratively adjust the randomisation parameters to enhance training efficiency. Heindl et al. [29] showcased this approach in BlendTorch, where adaptive randomization was used to improve the sim-to-real transfer for robotic perception tasks.

One interesting project for creating DR or GDR projects is BlenderProc [30]. 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 [31] in order to add new functionalities useful for synthetic data generation. Some of the functionalities added are random samples of positions, randomization of textures and trajectories among others. It was 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 library is that it allows one to obtain image annotations at the same time that it renders the RGB images of the scene, removing the need to render a simulation more than one time if multiple annotations are needed. The resulting data is stored and compressed as an hdf5 file, making it easy to access the stored information through their corresponding keys (colors, depth, etc).

Another project by Ritvik Singh et al. [32] leverages NVIDIA Omniverse Isaac Sim with the Replicator toolkit [33] to simulate and generate 2.7 million synthetic images for robot perception. In it, two complementary pipelines are used: (1) Indoor-room scenes, where rooms are furnished and a table is placed with Yale-CMU-Berkeley (YCB) objects plus a set of distractors (from Objaverse, Google Scanned Objects, and NVIDIA's asset library). (2) HDRI-background enclosures, in which objects are dropped to an unseen ground plane. To maximise diversity and better bridge the sim-to-real gap, each frame undergoes extensive randomisation (see table A.2 in appendix) and on-the-fly augmentations. Rendering randomisations include sampling material properties, lighting parameters and post-processing effects.

2.2.3 Diffusion-Based Data Augmentation

Traditional data augmentation for object detection involves rotation, scaling, flipping and other manipulation of each image which encourages the model to learn more invariant features, improving the robustness of the trained model. More advanced augmentation techniques involve generative methods, leveraging advances in diffusion models such as Contrastive Language–Image Pre-training (CLIP) and Stable Diffusion [34], which make use of text-to-image for augmentation strategies.

In this project, Stable Diffusion XL (SDXL) [35] has been used together with light-weight condition modules or controlnets [36] and IP-Adapter for inpainting styles [37]. The main idea is to generate

diverse object appearances while preserving the precise annotations obtained from the render pipeline.

2.2.3.1 Stable Diffusion XL (SDXL)

SDXL is a two-stage latent diffusion architecture developed by Stability AI. It consists of a "base" U-Net that produces coarse, low-resolution samples in a CLIP-based latent space, followed by a "refiner" U-Net that enhances fine details and fidelity [35]. SDXL employs an improved text encoder (Commune XL) trained on hundreds of millions of image–text pairs, enabling rich, semantically guided synthesis. By operating in a lower-dimensional latent space, SDXL achieves high-quality outputs at relatively fast sampling speeds, making it well-suited for large-scale data augmentation pipelines.

2.2.3.2 ControlNets

Pure text prompts offer rich guidance but lack precise control over spatial or structural constraints—critical for object detection augmentation. ControlNets [36] address this by attaching trainable "controller" branches to the frozen SDXL U-Net. Each controller learns to condition the diffusion process on auxiliary inputs such as edge maps, depth estimates, semantic masks or bounding-box layouts. In the pipeline of this project, Canny Edges and Depth maps are used as conditioning signals so that SDXL synthesises object variations (e.g., texture, colour, style) strictly within the background, without perturbing the target objects.

2.2.3.3 Inpainting with IP-Adapter

To further refine object-centric edits while maintaining scene coherence, the IP-Adapter module [37] was integrated. The IP-Adapter introduces a lightweight attention-based adapter to the diffusion model, trained specifically for inpainting tasks. Given an object mask and its surrounding context, IP-Adapter enables SDXL to "erase and replace" the object region—producing novel appearances, poses or partial occlusions—while preserving lighting, shadows, and background geometry. Crucially, the object's bounding-box annotation remains valid for training, since the synthesized content never drifts outside the original mask.

2.3 Old render pipeline

As described in the project from the last semester [8], the Mercedes-Benz team at the Arena2036 already had a preliminary version of a render pipeline that was used for obtaining synthetic data from CAD models of their automotive dataset [38], obtaining rendered images and annotations automatically.

Inspired by the work of Christopher Mayershofer et al. [39], the old pipeline used Blender [40] as the simulation platform for applying both DR and GDR to the set-up scene. The team leveraged these synthetically generated datasets to train multi-class object detection models, such as YOLOv8, addressing challenges like limited real-data availability and the high cost of manual annotation. By employing a GDR approach, the old pipeline aimed to minimise the sim-to-real gap and enhance the usability of synthetic data in real-world applications.

Figure 2.7 illustrates the conceptual parts 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 [41] was performed for filtering the generated images by their semantic content.

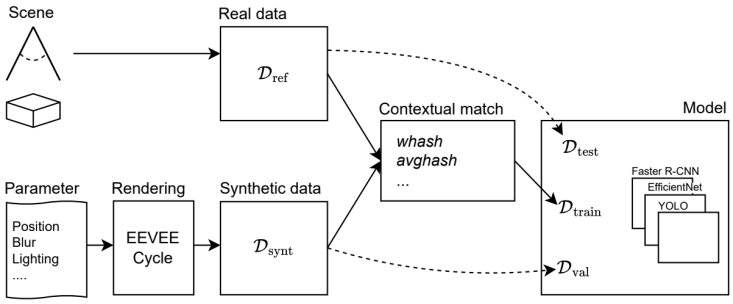


Figure 2.7: Old Synthetic data generation pipeline. It comprises a domain randomizer pipeline for photorealistic image generation and a context-aware image selection based on high-level (semantic) and low-level (pixel information) features.

As previously mentioned, the render part of the pipeline was implemented using Blender for creating multiple layouts with various backgrounds and distractor elements common in an industrial environment, so that there was a semantic context. The simulation was also able to adjust the lighting conditions, camera parameters and the pose of all the target objects in the scene as illustrated in figure 2.8, which shows the starting point of the simulation. It is composed of distractor models manually placed in distinct positions and orientations, atop planes with different materials which are used as background textures. The target objects and the camera orbit around an ”empty object”, an invisible blender object without mesh and material data, that is moved randomly in the scene while advancing from one layout to the next one. Finally, the lighting was composed of both area lights and environmental lighting to have better control of the illumination of the scene.

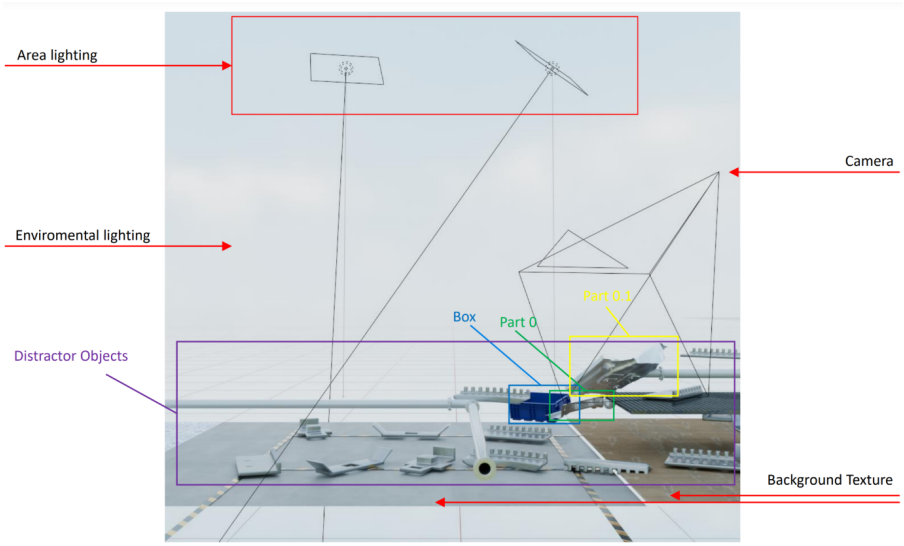


Figure 2.8: Elements present in a typical scene of the old synthetic data generation pipeline.

The simulation was designed to generate 1000 frames in total, where the target objects (Box, Part 0, and Part 0.1) move in front of the camera, orbiting and spinning at the same time. Among the 1000 frames, annotations were automatically generated as well through two separate rendering steps.

First, cycles render engine [42], known for its photorealism and ray-tracing capabilities, was used for rendering frames with as much quality as possible. During this phase, the empty object followed a

predefined path across the planes, which were lined up side by side (see figure 2.9). Both the target objects and the camera follow the empty object trajectory while also adding random rotations and translations to themselves while being occasionally occluded by distracting objects like metal rods, plates and pieces of furniture. Adding to the complexity, the lights in the scene moved along the empty object as well, changing their brightness to make dynamic lighting effects.

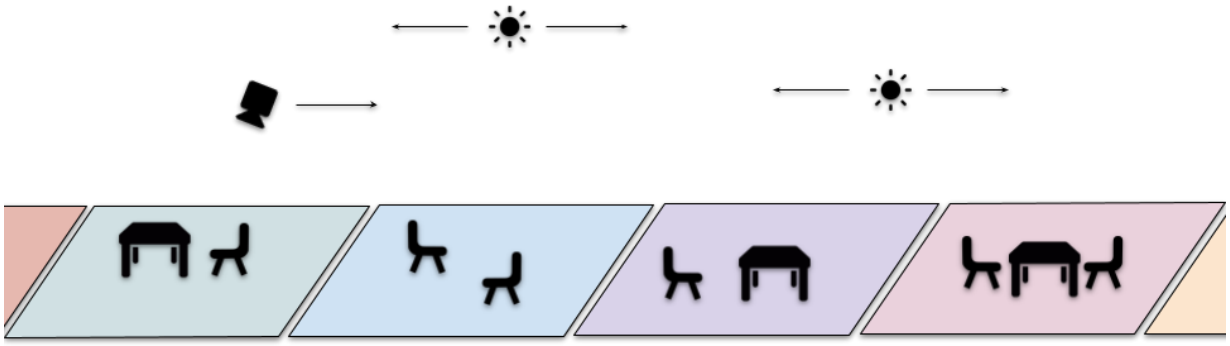
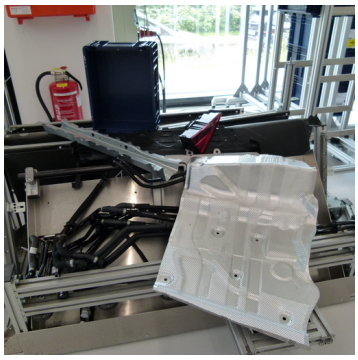


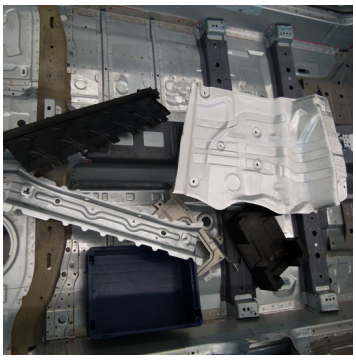
Figure 2.9: Concept of old pipeline with a plane for each setup.

A total of 1000 RGB images were obtained once the render using cycles was finished. However, these images had no annotations that could be used for training a detection model. For that, a second render was done to obtain the segmentation masks of these RGB images. To do so, all the lights in the scene were turned off, making the blender scene completely dark. Next, a glowing material of unique colour was applied to each of the target objects so that they emit light, becoming the only visible elements in the scene. The rest of the simulation was the same as the first one so the result will be the same. But this time the result was rendered using EEVEE, a faster rasterisation render engine from Blender, since the segmentation masks did not need to be photorealistic, cutting rendering time compared to cycles.

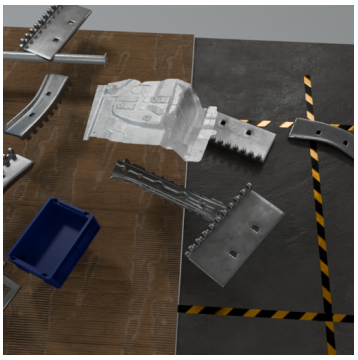
Once the data was generated, YOLOv8 was trained on the training dataset D_{train}, which included RGB images and annotations for each bounding box obtained from the segmentation masks created using EEVEE, and validated on D_{val}, which comprises 10% of D_{train}. Figure 2.10 shows how the target objects (Box, Part 0 and Part 0.1) look like, rendered using Cycles, alongside real examples.



(a) Real scene $\in D_{test}$



(b) Real scene $\in D_{ref}$



(c) Rendered scene $\in D_{train}$

Figure 2.10: Examples of images used by the old pipeline. (a) Sample of a real image used in the test set (D_{test}) for testing; (b) sample of a real image used in the reference set (D_{ref}) for hash filtering; (c) sample of a rendered image used in the train set (D_{train}) for training.

Furthermore, table A.1 in the appendix shows an example of the ranges of values the input parameters used in the old render pipeline [38].

This pipeline rendered a total of 1000 annotated frames with a resolution of 512x512 using the Cycles render engine for photorealism. The time it took for the first render using Cycles was 350 minutes, while the time it took for the second render using EEVEE was 13 minutes. In total, the pipeline was able to create a whole synthetic dataset with both RGB images and annotations in 6.05 hours, with an average of 21.78 s/frame. The resulting mAP50-95 after training YOLOv8 was 68.60%.

2.4 Datasets

This section describes the different datasets used for testing the pipeline. The selection of datasets is a challenging task since public datasets containing images of real-world objects and their CAD models at the same time are scarce. Most of the time, datasets which include CAD models only provide synthetically generated images, missing real images that could be used for comparing the sim-to-real gap.

2.4.1 Automotive dataset

The Automotive dataset [38] is a proprietary dataset created by the Mercedes-Benz team in Arena2036. It consists of 50 real-world images of three parts, being the two body-in-white objects relevant to the automotive industry and the R-KLT box relevant to logistics of multiple industry sectors (see figure 2.11). The selection of these parts was made considering the inclusion of different characteristics such as color, texture, material, surface and symmetry. Furthermore, the dataset includes the CAD models of each of the three parts as well as their textures.

The real-world images were taken with a Panasonic DMC-FZ100 camera with a resolution of up to 4320 x 3240 pixels and a Leica DC Vario-Elmarit 4.5-108mm f/2.8-5.2 ASPH objective lens. The images were later downscaled to 512x512 so that they could more easily be used for training detection models. Lastly, during the acquisition process, the camera used its automatic focus to allow for mechanical optic variations in the captured images.

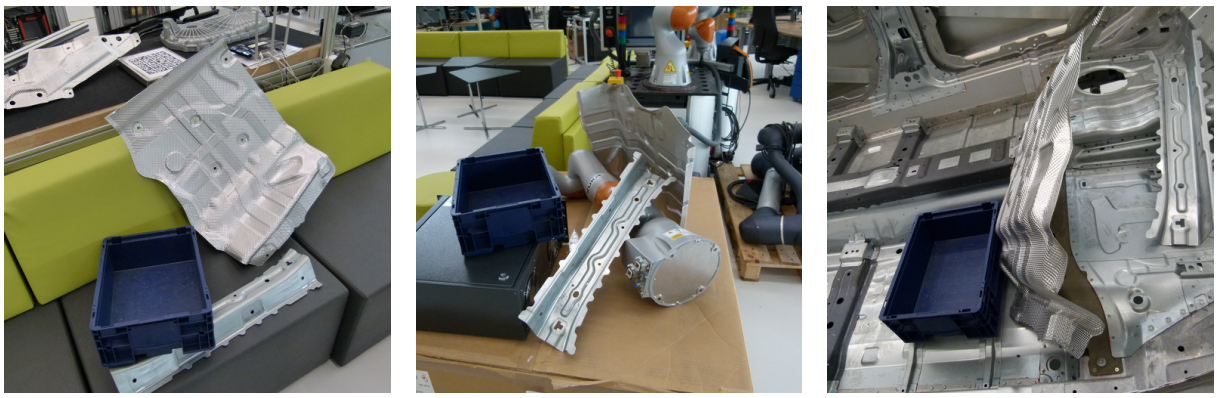


Figure 2.11: Examples of real test images from the Automotive dataset.

2.4.2 Robotics dataset

The Robotics dataset was introduced by Dániel Horváth et al. [43]. In their work, they propose a sim2real transfer learning method based on domain randomisation for object detection. With this

work they tackle one of the main obstacles of deep-learning-based models in the field of robotics, the lack of domain-specific labelled data for industrial applications.

The dataset is composed of 10 industrial parts, as can be seen in figure 2.12 showing three examples of the captured images. In total, 190 real images of 920 object instances were taken with different layouts and illumination settings. Using a frame for holding an Intel RealSense D435 camera at 310 mm above the ground. The light blue wooden base that supports the camera is included in the captured images as well. Moreover, the dataset also includes distractors such as cubes and spheres to increase the visual cluttering of the captured frames.

Object diversity as well as object similarity were the two major points of consideration. The former helps to evaluate the detection performance of the model for various types of objects, whereas the latter is important in assessing the classification performance of the model. This is because it is easier to misclassify objects with similar features. Thus, this dataset can be considered to be more challenging than the detection of less complex and fairly different shapes such as cubes and spheres.

(a) Example of target objects.

(b) Example with repeated target objects.

(c) Example with target and distractor objects.

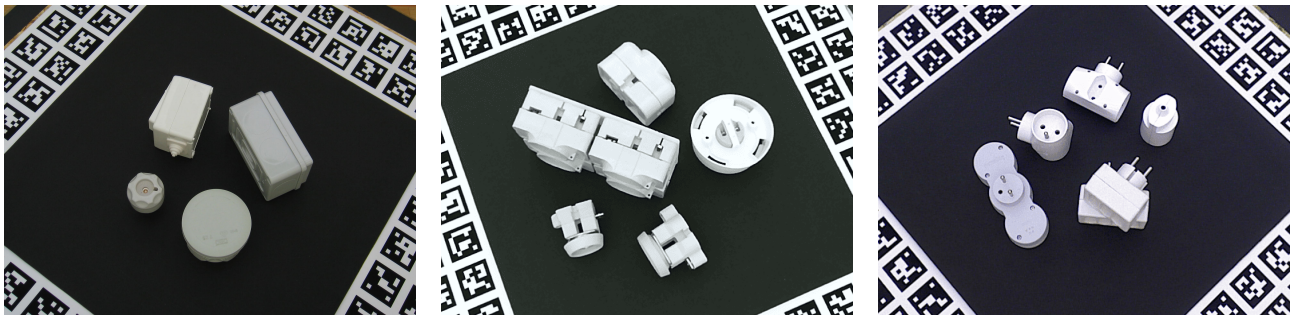
Figure 2.12: Example of real test images from the Robotics dataset.

2.4.3 T-Less dataset

The T-Less dataset created by Hodan et al. [44] features 30 industry-relevant textureless objects without discriminative colors of reflectance properties. The objects also exhibit symmetries and similarities with each other. The images from the T-Less dataset were captured with three synchronized cameras: a structured light RGB-D camera, a time-of-flight RGB-D and a high-resolution RGB camera, where they ended up with 39K training and 10K testing images from each camera. Furthermore, for each object type, there are two 3D models provided, i.e. a manually created CAD model, and a semi-automatically re-constructed one. The training images have individual objects against a black background, whereas the testing images have varying complexity, increasing from scenes with multiple isolated objects to challenging ones with many instances of several objects, to instances of several objects and also a high amount of clutter and occlusion.

The images were captured with an automatic procedure that involves sampling images from a view sphere.

The ground truth poses were annotated differently depending on the training and testing images. The training images had a more strict set-up containing a turning table with many markers. An example of an image from the T-Less dataset can be seen in figure 2.13



(a) Real image taken with Canon camera.

(b) Real image taken with Kinect camera.

(c) Real image taken with Primesense camera.

Figure 2.13: Example images from the T-Less dataset.

2.5 Summary of the Problem Analysis

For this thesis, which focuses on generating synthetic images and evaluating their impact on the performance of detection models, this section will summarize the findings from the Problem Analysis.

As previously discussed in section 2.2.1.4, although transformer-based detectors such as RF-DETR offer a promising new direction, their evaluation lies beyond the scope of this work. Instead, any robust object detector suffices for comparing the quality of synthetic data, and YOLOv8 has been chosen as the primary benchmark due to its state-of-the-art accuracy, widespread adoption across domains (e.g. autonomous driving, and medical imaging), and its efficiency in both training and inference, which supports rapid experimental iterations.

The legacy Blender-based pipeline described in section 2.3 suffered from two principal limitations: first, the manual creation of separate Blender scenes for each background or distractor led to poor scalability and slow render times; second, annotations could not be generated on the fly, necessitating multiple render passes for different label types. This resulted in slow render times, excessive memory usage, and low scalability.

To address these shortcomings, the proposed pipeline is a complete redesign of the old pipeline that automates the generation of fully annotated synthetic data in a performance-efficient way. Importing CAD models of the target objects into Blender and rendering them with the Cycles engine to achieve photorealistic base images. On top of this, Domain Randomization (DR) and Guided Domain Randomization (GDR) are implemented to automatically vary non-essential scene parameters such as lighting, textures, camera poses, and clutter distributions, so as to cover both broad and realistic target distributions. Finally, GenAI-based augmentation techniques such as Stable Diffusion XL conditioned via ControlNet and IP-Adapter inject randomized backgrounds and diverse object appearances while preserving exact annotation masks.

Taken together, these insights motivate the development of a unified, automated synthetic data pipeline that seamlessly integrates high-throughput photorealistic rendering, dynamic annotation, guided randomness, and diffusion-based augmentation to systematically close the reality gap and accelerate detector training.

Chapter 3

Problem Formulation

This chapter defines the final problem statement for the thesis, which will serve as a backbone, and define the research objectives which will help to evaluate the proposed solution to the problem. The objectives are based on both the problem analysis and the needs of the Mercedes-Benz team at Arena2036 in order to keep experimenting and researching in the domain of synthetic data for industrial contexts.

3.1 Problem formulation

Based on the problem analysis previously discussed in section 2, the problem statement can be formulated as the following:

How can Blender be utilised for the automated creation of industrial datasets for object detection, using CAD models as starting points?

3.2 Project objectives

The defined final problem formulation is supported by the project objectives, which are defined to provide some perspective on the direction of the following chapters. The objectives are defined, as previously mentioned, based on the analysis in chapter 2 and the needs of the Mercedes-Benz team.

Objective 1: Design and development of a new render pipeline.
The new pipeline must be able to leverage the render capabilities of Blender for creating synthetic data for an industrial domain using DR and GDR.

Objective 2: Extend pipeline to be able to render different models and scenes.
Contrary to the original pipeline which lacked scalability and flexibility, the new one must be able to load any target/distractor models and generate random layouts that will be rendered without the need to manually prepare each scene as seen in figure 2.9. Moreover, it should be able to create its own models to be used as distractors.

Objective 3: Allow the user to have a high level of control.
The new pipeline has to be highly configurable in order for the user to be able to use it in multiple use-case scenarios. Previously, the user needed a deep knowledge of the simulation software and render engine used (Blender and Cycles). The pipeline should provide the user with an interface that abstracts such complexity while allowing the user to easily adapt the pipeline to its desired use case.

Settings shall include at least all already available options in the old pipeline A.1 and introduce new ones such as the ones from Synthetic A.2.

Objective 4: Extend the pipeline to use data augmentation and filter
Implement as well data augmentation techniques using a diffusion model and data filtering of the generated synthetic data for optimising rendering and training times minimizing irrelevant features.

Objective 5: Benchmark new pipeline against multiple datasets
Benchmark the generation time and the quality of the generated synthetic data against using different datasets (public and proprietary). The dataset creation for the automotive dataset (the one used with the old pipeline in [38]) must be faster and have better quality.

Chapter 4

Methodology

This chapter focuses on explaining and describing the approach followed for making a working pipeline capable of creating synthetic data using CAD models (and optionally example images) as input.

As previously described in chapter 2, the creation of fully annotated datasets containing enough data for industrial environments is challenging. This is due to some issues such as data availability, scarcity, scalability, and dataset size among others. These problems can be solved using synthetic data generated with techniques such as DR and GDR, but it can be challenging to set up a simulation capable of generating photo-realistic data.

This pipeline aims to solve this problem and increase the amount of data that can be used for training detection models such as YOLOv8.

4.1 Pipeline Architecture

This section will discuss the general structure of the pipeline, which consists of the render, augmentation and filtering pipelines. All of them combined form the developed pipeline. The idea behind the synthetic data generation can be seen in figure 4.1, and is composed of three main elements: the render pipeline, the data filtering and the data augmentation pipeline.

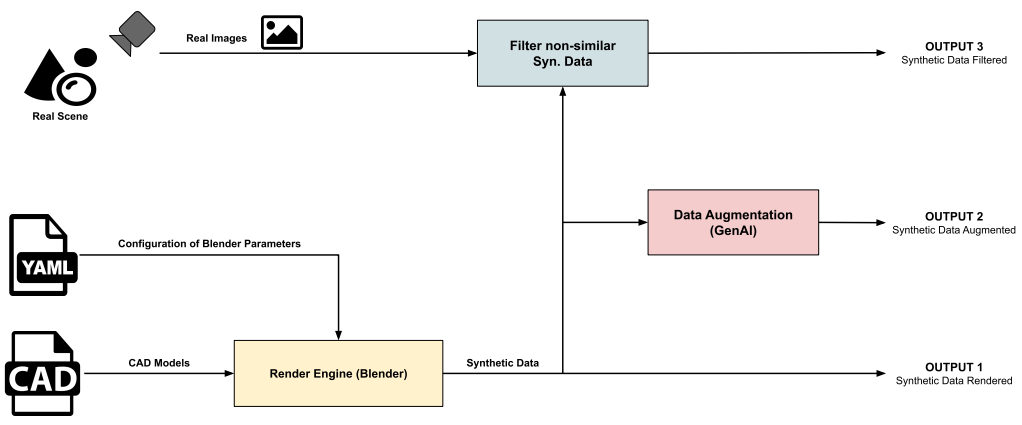
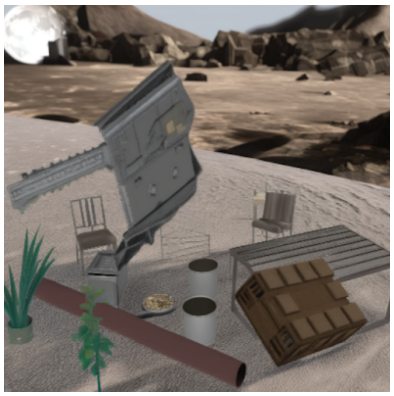


Figure 4.1: Diagram of the synthetic data generation.

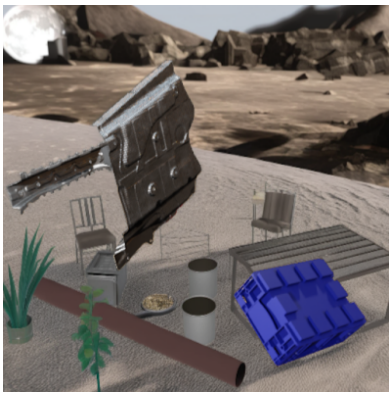
The pipeline is capable of generating data by itself without the need for any real-world information. It receives as input the CAD models of the target objects (objects from which we intend to generate synthetic images and annotations) and applies DR or GDR (depending on the configuration specified by the user) to generate as many randomised images as desired. Both the simulation and render are performed in Blender, leveraging the Path Tracing capabilities of its render engine Cycles. While different randomized layouts are rendered, annotations are also calculated on the fly for each of the frames without the need to run a separate render as was the case for the old pipeline. Once all the frames have been rendered, the data is saved as HDF5 files which contain the RGB, Depth maps, Normal maps, Segmentation masks and values of the parameters used on each frame. If desired, and using the generated HDF5 files as input, datasets can also be generated in COCO, YOLO and BOP formats.

Furthermore, if real-world data of the target objects is provided, it can be used for both augmenting and filtering the generated synthetic data. Augmentation of the rendered data can be carried out by making use of Stable Diffusion XL (SDXL), together with ControlNet and IP-Adapter. During the generation process, depth data is obtained as part of the annotations, while canny edge information is obtained from OpenCV's Canny Edge detector [45]. Both conditions are forwarded to Stable Diffusion with ControlNet as additional channels to be used along with the text prompt for image generation. To keep part of the style from the original image, the RGB image is passed as well to SDXL through an IP-Adapter. The use of both conditions along with the IP-Adapter ensures that the original composition of the synthetic image is maintained, while the objects of interest (target objects) are copied from the original image onto the augmented one to avoid undesired object modifications. Figure 4.2 shows an example of this augmentation process applied to the Automotive dataset introduced in section 2.4.1.

(a) Rendered image of Automotive dataset



(b) rendered image after SDXL



(c) Composting of original objects and augmented image.

Figure 4.2: Augmentation applied to the Automotive dataset. (a) Synthetic image generated with the render pipeline; (b) result image after applying SDXL with ControlNet and IP-Adapter; (c) final result after copying the original objects of interest on top of the augmented image.

Finally, filtering of the generated data can be done through both brightness and perceptual hashing. Using a selection of real-world images, a context-aware synthetic data selection of the randomised frames is carried out. Image Hashing is used for representing the semantic information of each image as a single hash number, bringing a simple yet effective method to match images based on their semantic content while being resilient to changes in pixel space such as brightness, cropping and

scaling. Brightness filtering can be performed as well for selecting synthetic images based on their pixel contents.

The following sections will focus on describing in detail each of the parts of the pipeline.

4.2 Render Pipeline

As already mentioned in section 2.3, the old render pipeline was designed in such a way that it was not possible to scale it to new CAD models or layouts. It was necessary to manually set up each of the models and materials used, the trajectories followed by them and the camera, and the background used.

The new implementation uses Blender [40] as well. Making use of its Python API [31], it is possible to code and automate any of the processes that can be done through the Blender GUI. As shown in figure 4.3, the pipeline receives CAD models that will be used for generating as many random scenes as needed. In each randomised scene, the models are placed over a plane with a randomised texture and both the lights and the camera are placed randomly around it. Once all the scenes have been generated, they are rendered using Cycles, obtaining both the RGB images and their annotations.

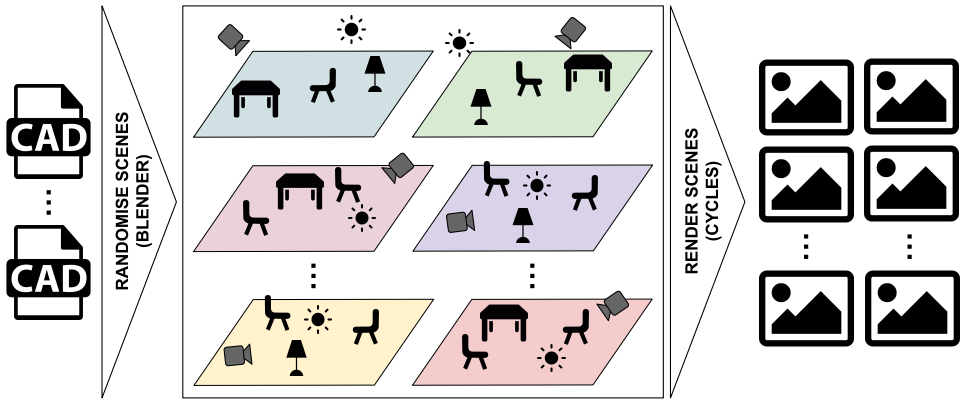


Figure 4.3: Diagram of the synthetic data generation.

The main steps carried out by the render pipeline will be now described.

4.2.1 Load and process data

The pipeline loads a configuration file (see example of config_template.yaml in appendix B.0.1) wherein it has been specified the paths to the CAD models, materials and ranges of values that the simulation should use as part of the DR approach used.

Both train and distractor models are loaded into Blender and undergo a processing step to make sure that they are compatible with the format of the pipeline (if the loaded model is composed of many meshes rather than formed of just one, a common parent is set to all of them for further easier management). After that, the children meshes of the loaded model, if there are any, are deleted based on a whitelist and a blacklist that can be specified on the configuration file. If specified in the configuration file, all the children meshes of a model can be joined together so that the segmentation will show all of them as a single object. Other properties from the configuration file such as the scale of the models or their identifier for further annotations are also applied to the loaded model at this stage.

Once train and distractor models have been loaded and processed, fake models can also be added to the scene if specified in the configuration file. The copies are called fake trains/distractors and can be made of simple geometric shapes (fake simples) such as cubes, spheres or cylinders, or made out of one of the already loaded models (fake similar). When a fake similar model is created, a deformation is applied to it so that its geometry does not look like the original model, while still keeping the same materials.

Due to blender design, it is not possible to keyframe a material (see next section 4.2.2 for further detail), because of that, a plane mesh is created for each of the loaded plane materials and randomly shown/hidden for making the effect of changing the material. Each new plane mesh is created as a ”linked” copy so that they all share the same mesh data and therefore do not increase the space in memory.

Finally, some other elements for the Blender simulation are loaded such as a default Blender scene for digital twin simulation, the area lights positions (which follow a three-point studio style lighting set-up) and the HDRI environmental images. If physics is enabled in the configuration file, a rigid body node is assigned to each of the models in the scene. Train models (and the fake trains) will have an active rigid body, being affected by forces. Distractor models (and the fake distractors) will have a passive rigid body, not affected by forces themselves but still affecting the active rigid bodies.

4.2.2 Set up scenes

Previous to the rendering, different scenes need to be generated (see figure 4.3) by randomising the simulation parameters, which include the model’s visibility, position, rotation and light intensity among others. Although one scene can be rendered at a time by calculating their randomised values, applying them to Blender and starting the render, this is not efficient if we intend to render thousands of images, as all the data (models meshes, materials, properties, metadata) needs to be loaded and unloaded from the GPU each time a render is performed, resulting in unnecessary overhead.

Therefore, to properly optimise the simulation the pipeline makes use of Blender’s keyframing options [46]. Keyframes are defined as ”a marker of time which stores the value of a property”. This feature is mainly used in Blender for creating animations since the value of a specific property (such as a position or light power) is stored or ”keyframed” into a specific frame. When the render is executed, it renders all the frames within a specified interval, with each stored keyframe being automatically reproduced before the rendering happens. In summary, this means that it is possible to associate the state of each scene to a frame and render them all at once, loading all the data on the GPU only once and, avoiding the overload of re-loading it for each frame.

This is the stage wherein the DR techniques are applied since the way the parameters for each of the generated scenes are randomised will result in very different results. The parameters being randomised are:

- **Environmental Background:** A random HDRI image is used for the background of the world. However, similar to what happened with the planes, environmental background is not a property that can be keyframed. Instead, the render is split into as many renders as HDRI images are available and the background is changed before the start of each render.
- **Environmental lighting (World lighting):** The strength of the light emitted by the HDRI image is randomised, with the range of possible values being defined in the configuration file.
- **Plane sampling:** As previously mentioned, Blender does not allow the keyframe of a material,

so it is not possible to just randomly sample a material for the plane. Instead, one of the multiple created planes with unique materials is randomly selected, mimicking the behaviour of changing the material.

- **Empty object pose:** The empty object works as the dynamic origin for the train models, the lights and the camera. Its position and rotation are randomly sampled from the volume of a sphere defined in the configuration file.
- **Camera (pose, depth of field):** The camera is always looking towards the empty object, with its focus on it, so that we can know where is the camera looking at every moment. Its position and rotation are randomly sampled from the volume of a sphere whose origin is defined as the pose of the empty object. Moreover, its depth of field is randomised as well by setting different values of f-stop.
- **Area lights (power, colour)**: The lighting in the scene uses three area lights with a configuration mimicking three-point lighting. These lights are always directed towards the empty object, making sure that anything within its surroundings (and hence any model within the POV of the camera) is affected by the light. The lights are designed to keep their pose relative to the empty object, so changes in the empty object pose result in new positions for the lights. This means that only the power and colour of the lights need to be randomised within the ranges of the configuration file.
- **Train model (sampling, poses):** The train models (targets) are those which will have annotations after rendering (although fake models do not generate annotations). For scene is composed of a different set of real and fake train models, as these are randomly sampled. Those which are selected to be part of the scene are randomly placed within the volume of a cube (of size defined in the configuration file) which can have its origin defined by the empty object or as fixed world coordinates.
- **Distractor model (sampling, poses):** The distractor models are the only models which do not use the empty object as the origin. In a similar approach to the train models, both fake and real distractors are randomly sampled to define the set that will compose the scene. Those which are selected are randomly placed within the volume of a cube of size and origin defined in the configuration file. For the default configuration, they are placed on top of the plane by just setting both their minimum and maximum height to 0.

These values are calculated for each frame (f) from 0 to $n-1$, being n the total number of scenes to generate. However, not all the generated scenes need to be rendered. For instance, it could be that only the randomised layout generated for the frame f_a needs to be rendered, being a waste of time and resources to render them all starting from f_0 till reaching f_a. Therefore, a render interval is used so that only[1] frames within the start-end (s-e) ranges are keyframed and rendered.

$$\text{Render interval} \in [f_s \leq f_i \leq f_e]; \text{ with } i \in [0, n) \tag{4.1}$$

Finally, only the sampled models should be visible in the simulation during a specific frame. For instance, for the planes that means that the non-selected should be hidden, making the illusion that a material has changed (while actually the whole plane has been replaced for another one). This not

[1] It is worth mentioning that the randomised values within the range [0, $s-1$] are still calculated (but not keyframed nor rendered) in order to keep the deterministic behaviour of the simulation (as this is determined by a randomisation seed that can be specified in the configuration file).

only helps with the scene creation but also during the rendering, since rendering all the models for each frame would be slower than just a set of them.

4.2.3 Render frames

Once all the random scenes have been generated $[f_0, f_{n-1}]$, rendering proceeds by selecting only the frames within the render interval $[f_s, f_e]$. However, as previously explained Blender cannot keyframe the value of the environmental background, so the render interval is split into subintervals (one per loaded HDRI image). Before rendering each subinterval, a random HDRI image is selected and assigned as the new HDRI environmental background.

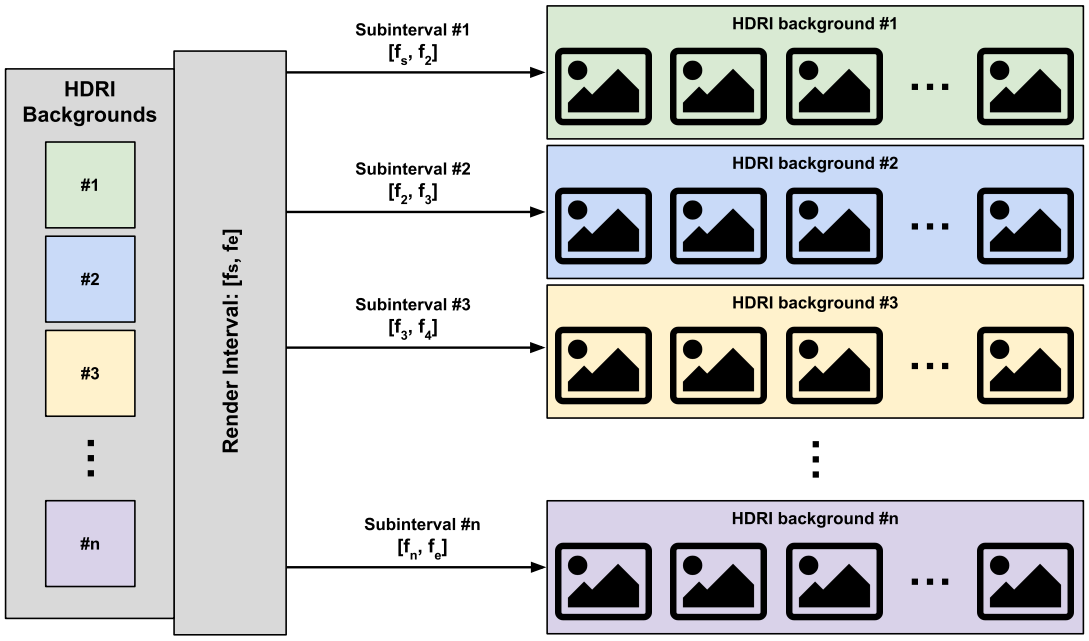


Figure 4.4: Diagram of the render interval being split into subintervals based on the number of HDRI images

The render outputs multiple passes simultaneously according to the configuration file. These are the RGB images and the following annotations: segmentation masks for the real train models, instance segmentation masks for all the models in the rendered scene (including distractors and fake models), depth map, normal map and the simulation metadata (raw keyframed values including poses, light intensities, camera parameters, etc).

All the output data (RGB and annotations) are stored in HDF5 files in order to reduce filesystem overhead and simplify downstream processing. Although it is also possible to store directly the data in separate folders, the HDF5 file output is preferred as it is used for creating the COCO, BOP and YOLO datasets.

4.2.4 Dataset creation

In the final phase, the pipeline reads each HDF5 file to retrieve rendered images and their associated annotations, then exports them into the selected dataset formats:

For COCO and BOP, the pipeline extends BlenderProc functions to produce validated JSON files. This process involves tracing exact object contours, computing bounding boxes, embedding camera

parameters, and verifying that all category and image IDs are consistent before writing.

For YOLO, labels are generated directly from the COCO annotations. Images are split into train and validation, and each image receives a text file containing normalized class IDs and bounding-box coordinates. By reusing the COCO metadata, this step avoids redundant computations.

4.3 Data Augmentation and Filtering Pipeline

In order to extend the options for synthetic data generation, the second part of the pipeline is focused on leveraging augmentation and filtering techniques to try to improve the generated data and guide it towards the domain of the real use case.

4.3.1 Data Augmentation

Data augmentation is performed through Stable Diffusion XL (SDXL). However, SDXL by itself only processes text prompts as inputs, making it difficult if not impossible to generate new synthetic images within the context of the rendered ones. To solve this problem, both ControlNet [36] and IP-Adapter [37] are used to add more layers of control to the diffusion model. IP-Adapter is used for giving the original RGB image as an input to the model so that it can be aware of how it should look like. Furthermore, Canny and Depth conditions from ControlNet are used as well to control the resulting layout of the generated image. Figure 4.5 shows a scheme of the inputs that the augmentation part uses for increasing the amount of synthetic data.

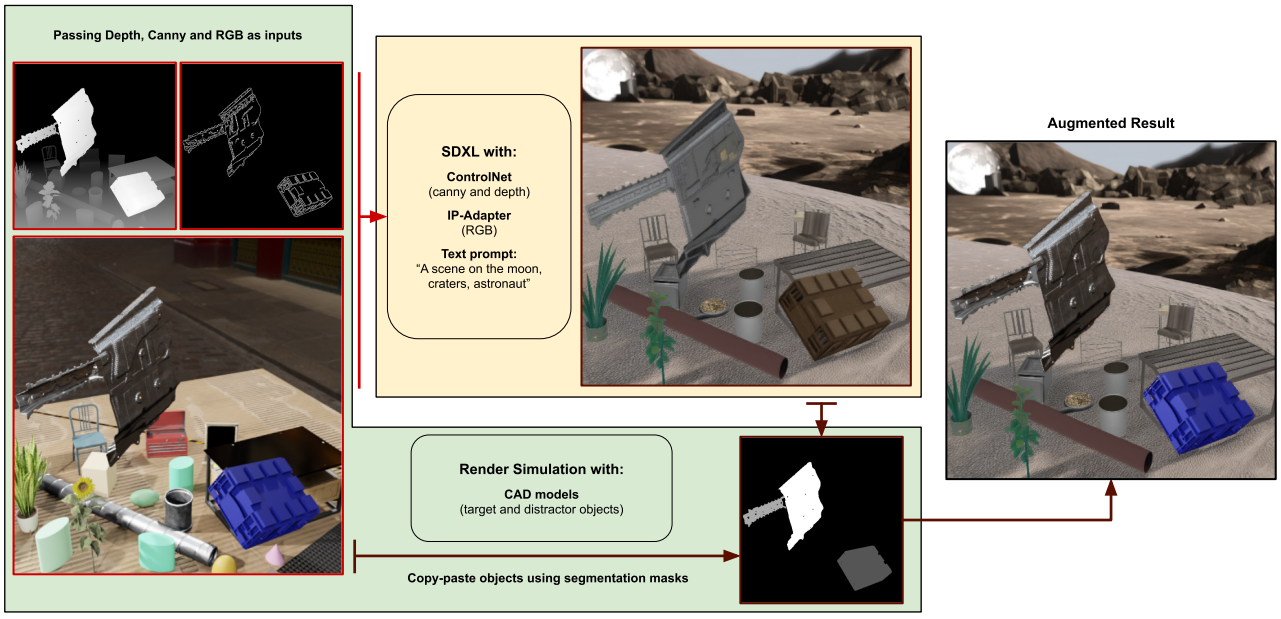


Figure 4.5: Scheme of data augmentation with SDXL

The augmentation is carried on over the rendered images in Blender. Since the annotations used for the ControlNet (Depth and Canny) come from the simulation itself, there are no outliers or noise in the annotated data. Having pixel-level precision in the annotations for segmentation and depth, rather than having to use a segmentation model or a depth estimator that can induce some level of noise.

As the bounding box of the train models in the original RGB image is known and the position of the train models in the new image did not change, the annotations of the original RGB image can used

for the new ones as well. However, the new image will most likely have multiple shapes and items where the train models used to be in the original one rather than an accurate representation of the target object. The semantic segmentation masks calculated during the rendering are used as binary masks for composting both the train objects of the original image and the augmented image created with SDXL. This is done to make sure that the detection model does not train with a deformed and less accurate representation of the target objects.

4.3.2 Data Filtering

The other way in which the rendered synthetic data can be used is as a baseline for applying filtering techniques.

DR alone is a non-trivial task, as finding a general solution to generate enough synthetic variability to enclose real-world variations while avoiding unnecessary scene complexity can be challenging. Therefore, apart from a pure DR approach and as already mentioned in section 2.2.2.2, a context-aware synthetic data selection process can be carried out to perform GDR.

The selection process consists of filtering the already generated data so that the final result is closer to de use case domain by its similarities to a reference test set composed of real images that are not used for training or testing the trained model. The following two approaches have been implemented:

- **Using low-level features:** Synthetic images can be filtered based on their pixel contents and similarity. For this case, the train set is filtered by its brightness similarity to real images contained in the reference set. Brightness is calculated as the average normalized pixel value across each image with the pixel value being denoted as $I_{i,j}$ and I_{max} representing the maximum pixel value.

$$B_{brightness} = \frac{1}{M \cdot N} \sum_{i=1}^{N} \sum_{j=1}^{n} \frac{I_{i,j}}{I_{max}} \tag{4.2}$$

- **Using high-level features:** Synthetic images can be filtered based on their semantic information. Perceptual hashing allows the representation of the semantic information of an image as a single hash number. The operations involved in the extraction of the hash numbers are designed to preserve the hash after changes in pixel space such as brightness, cropping and scaling [41]. Therefore, the semantic content of two images can be compared by calculating the Hamming Distance [47] between their binary hash arrays. For instance, being A and B the perceptual hash numbers of two images, the Hamming distance $\Delta_{Hamming}$ is calculated by counting the number of binary elements that are different in each hash array.

$$\Delta Hamming = |\{i \in \{1,\ldots,n\} A_i \neq B_i\}| \tag{4.3}$$

In summary, by combining both low-level and high-level feature-based data selection, the filtering stage aims to prune away synthetic samples whose illumination or semantic structure diverges too far from the target real-world domain. The brightness criterion ensures that only those images whose overall luminance matches the reference distribution are retained, while the perceptual-hash criterion further guarantees that the retained images exhibit scene content structurally similar to real examples. Taken together, these two complementary filters yield a curated subset of renders that balance diversity with domain fidelity.

Chapter 5

Implementation

This chapter focuses on explaining and describing the implementation process followed in order to carry out the pipeline as described in chapter 4. Additionally, it explores implementations for each of the introduced datasets in Section 2.4.

5.1 Render pipeline: SynthRender

The render pipeline was implemented using the Blender Python API [31], and leveraging BlenderProc [30] for synthetic data generation. The code was developed in Python 3.12 and is compatible with both Linux and Windows.

For the render pipeline, a python module called **SynthRender** [48] was developed, which automates the whole process of synthetic data creation: CAD models loading, scene randomisation, and rendering as described in section 4.2. Moreover, a configuration such as the one given in Appendix B.0.1 is used for the pipeline to be adjusted to specific use cases. Figure 5.1 shows how the main code calls the different scripts for producing a fully synthetic YOLO dataset.

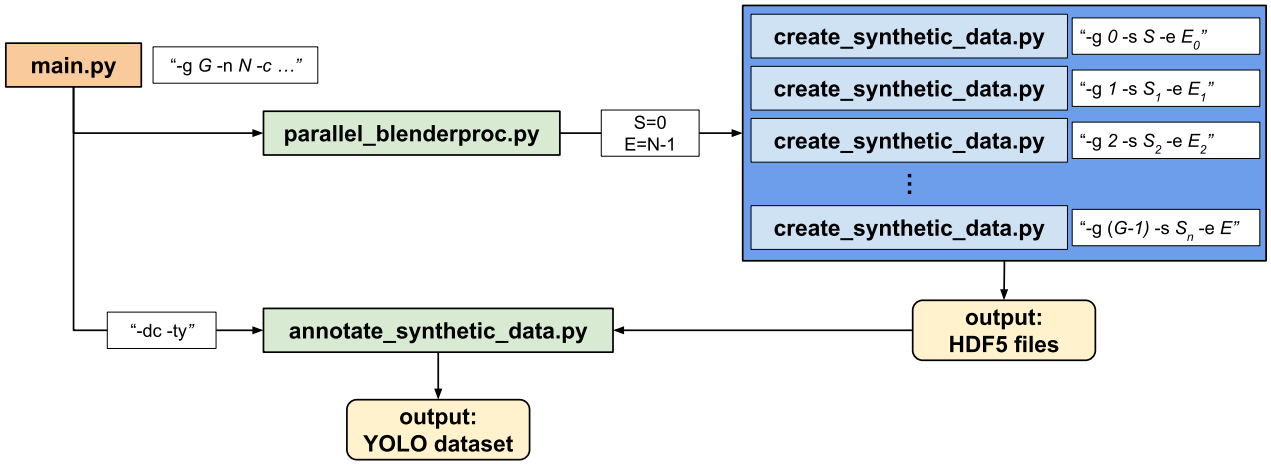


Figure 5.1: Scrips being called by main.py. The diagram shows the parallelisation of the render interval and the annotation into YOLO.

The module contains the following scripts:

- **main.py:** Renders and annotates the data automatically.
- **create_synthetic_data.py:** Renders data and stores it as HDF5 and/or raw files.
- **parallel_blenderproc.py:** Runs independent instances of create_synthetic_data.py on each GPU for parallelising rendering.
- **annotate_synthetic_data.py:** Annotates the HDF5 files as COCO or BOP format and/or creates a YOLO dataset with it.
- **vis_data.py:** Allows visualising HDF5 files and annotations.

A detailed description of each of these scripts and their operation can be found below. While running main.py is enough for generating data, the other scripts are still useful if only a specific part of the pipeline needs to be executed or if debugging is needed.

5.1.1 Scripts: main.py

The main.py script automatically calls and executes all the necessary functions and scripts for creating synthetic data based on a loaded configuration file and the loaded CAD models. This will make one (or many, depending on the [-g] parameter, if set) instance of *create_synthetic_data.py* for rendering all the data as HDF5 files. Once it has finished, it will call the *annotate_synthetic_data.py* script to create the needed annotations.

The script accepts the following arguments:

Input parameters	
`Variable`	**Description**
`-n`	Number of frames that the pipeline will render.
`-c`	Path to the config file used for adjusting the pipeline.
`-g`	The script will uniformly split the frames to render into *g* intervals, each running on its own dedicated GPU.
`-dc`	Whether to annotate the result in COCO format.
`-db`	Whether to annotate the result in BOP format.
`-ty`	Whether to turn the COCO annotations into a YOLO dataset.

Table 5.1: List of input arguments for the main.py script.

Moreover, the script can be called as:

```
$ python main.py -n <number_frames>
    [-c <config_file_path>][-g <number_of_GPUs>][-dc] [-db] [-ty]
```

5.1.2 Scripts: create_synthetic_data.py

This is the script in charge of creating the randomised scenes. Moreover, the scenes will also be rendered if a render interval is defined with the *-s* and *-e* arguments. There is also the option to visualise the generated scenes and their keyframes by setting the command with the ”debug” mode rather than with ”run”.

```
$ blenderproc <debug|run> src/create_synthetic_data.py -n <number_keyframes>
    [-c <config_path>][-s <render_start-frame>][-e <render_stop-frame>]
```

When running, *create_synthetic_data.py* will internally call two separate scripts. The first one is called *simulation_setup* and is in charge of setting up the simulated scenes (loading models, randomising parameters and generating scenes) as described in section 4.2.2. The second is called *simulation_render* and is in charge of setting up the render configuration and rendering the generated scenes as described in section 4.2.3.

5.1.2.1 Simulation package: simulation_setup

set_up_scene:
This part of the code is in charge of initialising the Blender simulation and loading the models and the materials (see whole code under appendix C.0.1).

A part of the code for loading the target models (also referred to as train models), and similarly, the distractors are shown in the snippet 5.1.1 (while the entire code can be seen in the appendix C.0.2 under the function *load_train_models*). It first checks whether the specified folder containing the models exists and, if so, the models are loaded and stored in the **train_models** list. Both a whitelist and blacklist are applied to make sure that only the desired models in the folder are loaded.

Snippet 5.1.1 Example of the code for loading the train models.

```
if os.path.isdir(dir_path:=config["train_models_dir"]):
    pos_callback = lambda x, i: (range(-len(x)//2+1, len(x)//2+1, 1)[i], -5, 0) # Setting
    ↪  up initial position of models
    whitelist = config["train_models_whitelist"]
    blacklist = config["train_models_blacklist"]

    models_config = {"default_config": config["models"]["trains"]}
    models_config.update(config.get("custom_models", {}))

    train_models = bproc_utils.load_models_folder(dir_path, whitelist, blacklist,
    ↪  pos_callback, collection, models_config)

    # Setting category_id to train objects.
    bproc_utils.set_category_to_meshes(train_models, models_config)
```

Both a default configuration and a custom_models configuration are applied. The first is applied to all the loaded models and includes options such as the number of copies of each model or their scale. The second is applied only to the models with a specific name and allows the user to have more detailed control of the loaded models (an example of the default and custom configuration can be seen in the appendix B.0.1 under ”trains” and ”example.obj” keys).

For each of the loaded models, a processing operation is performed as shown in the appendix C.0.3. The *load_model* function merges all the children meshes of a model under the same parent, making sure that all the non-desired children are removed.

After all the real models are loaded, additional fake models can be created if specified in the configuration file. The appendix C.0.2 shows the rest of the implementation for the *load_train_models* function and how the *similar_trains* and the *simple_trains* are added into the simulation. The fake simple models are just geometric figures such as cubes, cylinders and spheres. The similar fakes are real models that have undergone deformation so that they have the same material but are not the same mode as shown in figure 5.2.

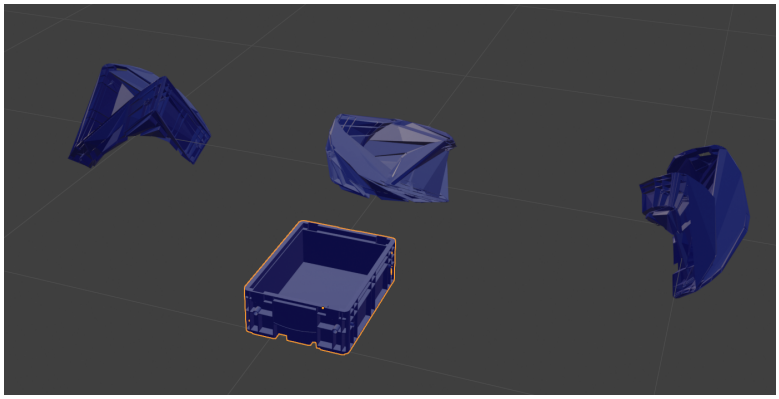


Figure 5.2: Example of the blue box from the automotive dataset and three similar fake generated from it.

.

The only big difference between the loading process of the train models and the distractors is that only real train models get an ID which will later be used while annotating the data. This is done with the function *set_category_to_meshes* (shown in the snippet 5.1.2), which is called from the function *load_train_models*, and sets the custom property ”category_id”, used by BlenderProc for annotating the models while rendering.

Snippet 5.1.2: Code for setting a custom property only to the real train models.

```
def set_category_to_meshes(meshes:list[MeshObject], models_config:dict[str, dict] =
↪  None):
    for mesh in bproc.object.get_all_mesh_objects():
        mesh.set_cp("category_id", 0)

    models_config = models_config or {}

    count = itertools.count(start=1) # Iterator that counts starting from 1.
    for mesh in meshes:
        parent_name = mesh.get_name()
        model_config = {}
        model_config.update(models_config.get("default_config", {}))
        model_config.update(models_config.get(parent_name, {}))

        whitelist = set(model_config.get("segment_whitelist", []))
        blacklist = set(model_config.get("segment_blacklist", []))
```

```
        for child_mesh in get_all_child_meshes(mesh):
            if child_mesh.has_cp("combined_mesh"): continue # Skipping combined_meshes

            child_name = child_mesh.get_name()

            if whitelist and child_name not in whitelist: continue # child not in
            ↪ whitelist, not annotated.
            if blacklist and child_name in blacklist: continue # child in blacklist, not
            ↪ annotated.

            child_mesh.set_cp("category_id", next(count))  # Set a unique category ID
            ↪ for each mesh.
```

The rest of the *set_up_scene* function focuses on other main aspects, including the creation of the empty object (a Blender mesh object without mesh data) that is used as a reference, the creation of the area lights, the creation of copies of each model for having multiple instances of them if required, and the setting up of the physics for the models through rigid body nodes.

set_up_keyframes
This function manages the generation of every random scene by leveraging Blender's keyframes. In order to do so, it first hides all models from the render (via each model's ".hide_render" attribute) to ensure a clean starting point for the simulation. Then, for each frame in the range $[0, n-1]$, it gets a random sample of train and distractor models, randomises their pose, and randomises the rest of the simulation parameters. This process is repeated for each of the frames from 0 to $n-1$, but none of these calculated values is applied to the simulation yet.

Only when the current frame lies within the designated render interval $[s, e]$, does it apply all the calculated values. That means unhiding the selected models, setting their pose and setting the rest of the parameters to the scene. The snippet 5.1.3 is a reduced version of the whole loop for generating a scene (which can be seen in the appendix C.0.4). After setting all the values in the simulation, the current state of the scene is keyframed for later reproduction. Since Blender operations (keyframing, moving models, etc) are slow, these changes are applied and keyframed only to those frames within the render interval. However, all the random operations are still performed for every frame in order to maintain the deterministic behaviour of the pipeline.

Snippet 5.1.3: Simplified code for randomising and keyframing scenes within the render interval.

```
for frame in tqdm(range(0, num_keyframes), desc="Preparing keyframes", unit=" keyframe",
↪ disable=verbose):
    ##########################################
    # Generate random values for the scene... #
    #     (model poses, light power, etc)     #
    ##########################################
```

```
    if start_frame <= frame <= stop_frame:
        ###################################
        # Set the values for the scene... #
        # (move models, set lights, etc)  #
        ###################################

        # Keyframe state of the scene:
        bproc_utils.save_keyframe(frame, items_children)

        # Recalculate train objects' position with physics simulation.
        scene_setter.do_physics(self.config, set_train_models, set_distr_models, frame)

        # Set sampled models to be hidden again for a fresh start.
        to_hide_models = [*set_train_models, *set_distr_models, plane]
        bproc_utils.hide_render_view(to_hide_models)
```

As previously described in section 4.2.2, the parameters that are randomised are:

- **Empty object:** The empty object is a mesh object with no mesh data, which makes it invisible. Yet it can still be used as a frame of reference for placing the models and the camera in the scene. Therefore, its pose is randomly sampled from the volume of a sphere.
- **Environmental light:** its value is randomly drawn from a defined range in the configuration file.
- **Area lights:** Rather than specifying Blender's input energy E directly, the code draws uniformly a light intensity value (I) from within the desired range specified in the configuration file for the common target (empty object). This sampled intensity value is mapped to its equivalent energy according to the known distance from the area light to the target:

$$\text{Intensity } (I) = \frac{Energy\ (E)}{Area\ (A)} \tag{5.1}$$

  This makes sure that no matter the distance from the light to the target, its intensity will be the same. Figure 5.3 shows the two different methods in which light intensity can be sampled from the $[I_{\min}, I_{\max}]$ range defined in the configuration file.

  (1) Linear: the intensity value I is picked from the defined range $[I_{\min}, I_{\max}]$ at random.

$$\text{Intensity } (I) = random.uniform(I_{\min}, I_{\max}) \tag{5.2}$$

  (2) Exponential: the intensity is obtained as the mapped value between the frame number and the $[I_{\min}, I_{\max}]$ range using an exponential function so that its growing rate can be adjusted by a factor. By doing so, it is possible to change the ratio of low-lit and overexposed images as desired.

$$\text{Intensity } (I) = I_{max} \cdot \left(\frac{frame\_number}{Distance^2} \right)^{factor} \tag{5.3}$$

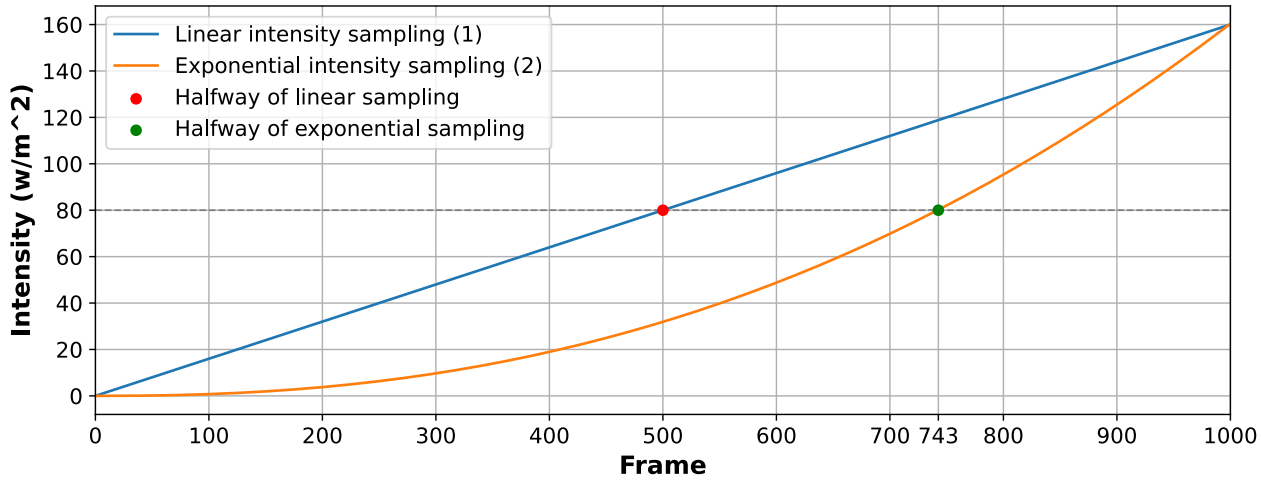


Figure 5.3: Sampling intensity values ($[0, 160]$ w/m^2) with both methods for 1,000 frames. (1) Linear method gets half of the maximum intensity at frame 500; (2) Exponential method gets half of the maximum intensity at frame 743 (exponential factor of 2.33).

- **Plane:** A plane is randomly sampled to be part of the scene. The visibility of the sampled plane is enabled (while the rest of the planes are hidden) so that a random plane with a material appears in the scene.

- **Models**: Models are randomly selected to form part of the scene. A free-of-collision pose is calculated using algorithms such as SAT (see section 5.1.7 for more details) and using the empty object as the origin.

- **Camera**: The location of the camera is sampled from the volume of a random sphere centred on the empty object. The direction of the camera is set as the direction that looks towards the empty object to make sure that the camera is always looking wherein the train models are spawned. Finally, a random f-stop value can also be calculated to adjust the depth-of-field of the camera.

Furthermore, if the physics option is enabled in the configuration file, the rigid-body node is enabled for all the unhidden models in the scene, letting the train models (designated as active rigid bodies) fall, collide and come to rest. Once the simulation has finished, the keyframe storing the poses of the moved models is updated. As a final step, the sampled models are hidden again so that the next iteration begins with an entirely clean scene.

An example of how the visualisation of a generated scene using the debug mode in Blender is shown in figure 5.4.

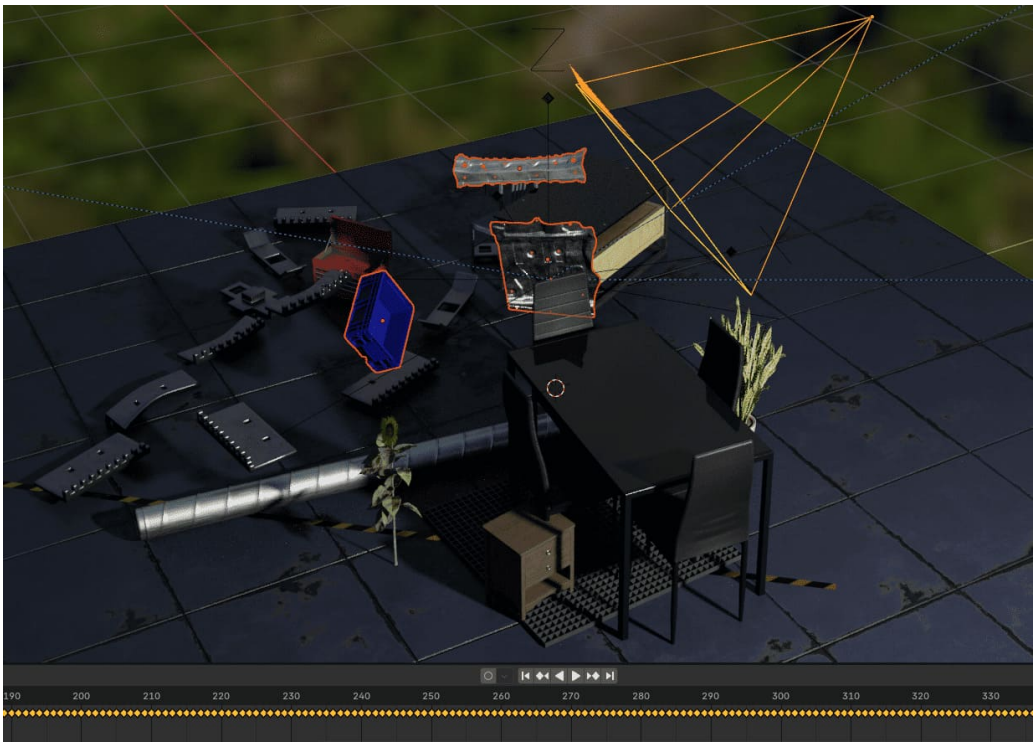


Figure 5.4: Generated scene visualised from the debug mode showing the train models highlighted and the keyframes at the bottom.

5.1.2.2 Simulation package: simulation_render

set_up_renderer:
This part of the code is in charge of initialising the Blender's render engine Cycles based on the values set in the configuration file. Namely, the types of annotations that should be generated while rendering, the type of outputs, the number of samples used by cycles and the number of GPUs used while rendering.

In order to have an easier interface for Blender's API, BlenderProc is used since it has already implemented functions to enable different types of annotations. The depth maps, normal maps and segmentation masks are enabled by making use of the following BlenderProc functions:

```
bproc.renderer.enable_segmentation_output()
bproc.renderer.enable_normals_output()
bproc.renderer.enable_depth_output()
```

One important aspect of how the segmentation module works in BlenderProc is that it will generate both instance segmentation masks and semantic segmentation masks. Calling the function *enable_segmentation_output* will enable the segmentation for all the models which have a specific "custom property". In this case, the default property suggested by BlenderProc in their documentation [49] is used for segmenting only the real train models. The assignment of the custom property "category_id" to the desired models was previously described in section 5.1.2.1.

The other two main options that are set during the *set_up_renderer* function are the amount of samples Cycles uses and the number of GPUs used while rendering. Sampling is the process of tracing rays from the camera into the scene and bouncing them around until they reach a light source such as a

Light object, an emissive mesh, or the world background [50]. The number of samples used can be set in the configuration file (it is set to 100 by default), a higher number results in a cleaner image at the cost of a longer render time.

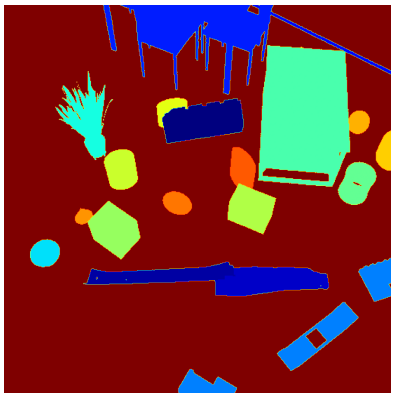
```
bproc.renderer.set_max_amount_of_samples()
bproc.renderer.set_render_devices()
```

render_scene:
This function is in charge of rendering all the generated scenes that have been keyframed. The snippet 5.1.4 shows a simplified slice of the code used. The function is in charge of iterating through all the frames within the render interval $[s, e]$ and rendering them. However, as previously explained the environmental background image can not be keyframed. The solution consists of splitting the render interval into smaller sub-intervals (one for each background) wherein the background of the world is changed before calling the render function.

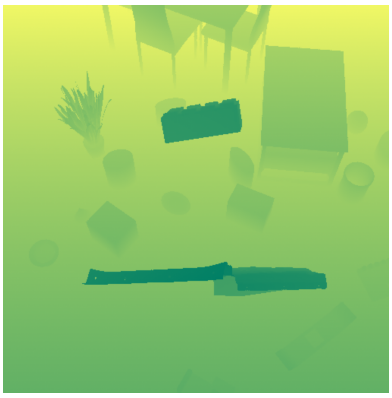
Snippet 5.1.4 Simplified code for rendering

```
for i, (interval, background_path) in enumerate(zip(intervals, backgrounds)):
    start, stop = interval
    if start_frame > stop or stop_frame < start:
        continue

    start = max(start, start_frame) # If start_frame is in between an interval, fix start.
    stop = min(stop, stop_frame)    # If stop_frame is in between an interval, fix stop.

    if background_path:
        scene_setter.set_background_texture(background_path, loadfile=True)

    bproc.utility.set_keyframe_render_interval(start, stop+1) # set render interval
    data = bproc.renderer.render(output_dir=self.output_dir_rgb, verbose=verbose)

    if self.config["save_hdf5"]:
        # .. get keyframed data ..
        bproc.writer.write_hdf5(self.output_dir_hdf5, data)
```

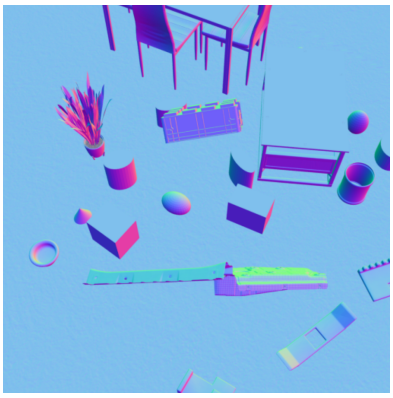
After the completion of each sub-interval, the rendered data is saved as HDF5 files. Each frame has its own HDF5 file containing all the necessary information (metadata, annotations, RGB, randomisation seed, etc). The following figure 5.5 shows an example of the stored data in one of these files.



(a) Instance segmentation

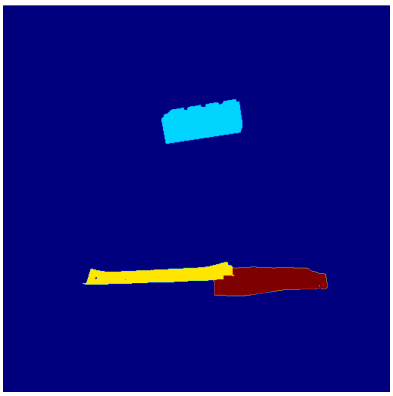


(b) Depth map



(c) Normal map

(d) RGB image



(e) Semantic segmentation

Figure 5.5: Renders result from the Automotive dataset showing both the RGB image and the annotations.

5.1.3 Scripts: parallel_blenderproc.py

The render engine Cycles is capable of distributing the sampling [50] of a scene between multiple GPUs. However, as noticed during the work of the last semester [8], if more than one GPU is available it is possible to further parallelise the whole rendering process by creating multiple dedicated instances of the code, each running on a single GPU.

It leverages the implementation of the code *create_synthetic_data*, which as previously mentioned accepts both a number of random scenes to calculate (n) and a render interval $[s, e]$ as well as a GPU index (g) (see section 5.1.2 for further details of the render interval). The script splits the original render interval into as many sub-intervals as GPUs are available based on the g argument. Next, for each sub-interval $[s'_i, e'_i]$ a dedicated instance of the *create_synthetic_data* is executed.

```
$ python src/parallel_blenderproc.py -n <number_frames>
    [-c <config_file_path>][-s <render_start-frame>]
    [-e <render_stop-frame>][-g <number_gpus>]
```

Figure 5.6 shows an example of how it looks if the script is called with $-n100$ and $-g4$. As can be seen, it splits the 100 frames into four intervals, of 250 frames each, leaving each GPU in charge of rendering only its respective section.

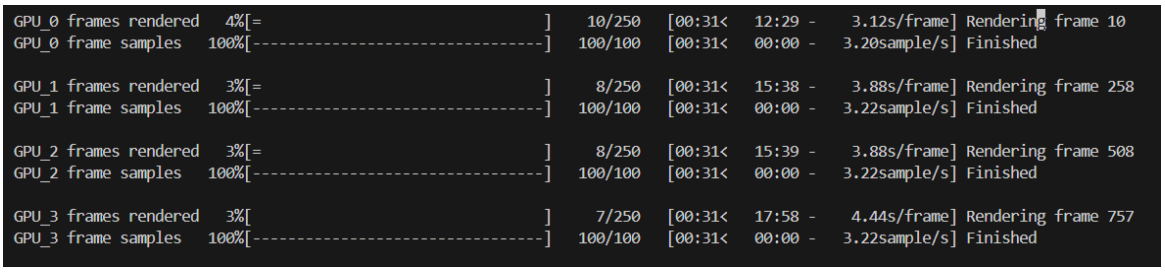


Figure 5.6: Parallel rendering with 4 dedicated processes.

5.1.4 Scripts: annotate_synthetic_data.py

This is the script in charge of creating multiple datasets from the generated HDF5 files. It leverages the generated annotations for creating datasets in COCO, BOP and YOLO formats. In order to execute it, it is enough to pass the path to the configuration file used for rendering the scenes as an argument and indicating the desired format of the output dataset (similar to the arguments in table 5.1).

```
$ blenderproc run src/annotate_synthetic_data.py
    [-c <config_file_path>][-dc] [-db] [-ty]
```

The script uses two main functions *generate_annotations* and *coco2yolo*. The first one uses the BlenderProc methods for formatting data as COCO and BOP while the second one adapts an already generated COCO dataset into a YOLO one.

generate_annotations:
The function scans the contents inside of the output directory for the HDF5 files and opens them in batches using the python package *h5py*. The loaded data is then converted into a Python dictionary wherein each key contains a list with the data for each frame. The keys used for accessing the data are: 'category_id_segmaps', 'colors', 'depth', 'instance_attribute_maps', 'instance_segmaps', 'normals', 'models_data'. All the keys and their content are generated after rendering the frames except for the 'models_data' key. This key contains all the metadata of the simulation for that specific frame. This includes the position of all the models in the scene, the energy set for the lights, the selected plane and the HDRI background image used.

Each batch of loaded data is then processed by two different functions depending on the input arguments for generating the datasets. The snippet 5.1.5 shows how the COCO annotations are created based on the: 'instance_segmaps', 'instance_attribute_maps' and the 'colors'.

Snippet 5.1.5 Function from the Coco_Annotator class used for creating the dataset.

```
def annotate_data(self, batch, hdf5):
    bpy.context.scene.frame_start = 0
    bpy.context.scene.frame_end = batch

    bproc.writer.write_coco_annotations(
        output_dir=self.output_path,
        instance_segmaps=hdf5["instance_segmaps"],
        instance_attribute_maps=hdf5["instance_attribute_maps"],
        colors=hdf5["colors"],
        color_file_format="PNG",
        indent=4,
        append_to_existing_output=True,
        mask_encoding_format="polygon"
    )
```

Similarly, the following snippet 5.1.6 shows how the BOP annotations are created from the HDF5 files using the keys: 'colors', 'depth' and 'category.

Snippet 5.1.6 Function from the Bop_Annotator class used for creating the dataset.

```
def annotate_data(self, target_elements, start, batch, fixed_hdf5):
    bpy.context.scene.frame_start = start
    bpy.context.scene.frame_end = start + batch

    bproc.writer.write_bop(
        output_dir=self.output_path,
        target_objects=target_elements, #all_elements,
        depths=fixed_hdf5["depth"],
        colors=fixed_hdf5["colors"],
        calc_mask_info_coco=True,
        append_to_existing_output=True,
        frames_per_chunk=batch
    )
```

coco2yolo:
Once a COCO dataset has been generated from the HDF5 files. It can be converted into a YOLO dataset as well since all the required data is contained within it. If the argument '*ty*' is passed to the script, it will split the RGB images into train and val folders with an 80/20 ratio, respectively. Next, it will fetch the bounding boxes for each of the models from the COCO JSON file and create text files with them.

Once the function has finished, the resulting dataset can be used for training YOLO models.

5.1.5 Scripts: vis_data.py

This script can be used for visualising the content of the generated HDF5 files or COCO annotations. In order to do so, it automatically takes the directory of the HDF5 and the COCO dataset from the configuration file passed as argument. The script can be called to visualise a specific frame like:

```
$ python src/vis_data.py <hdf5|coco> <frame>[-c <config_path>]
```

Figure 5.5 shows what the output looks like if the "hdf5" option is passed as an argument.

5.1.6 Render pipeline: Key features

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.

5.1.6.1 Deterministic results using a seed

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.

5.1.6.2 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 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.

5.1.6.3 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.

5.1.6.4 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 paralleli-

sation 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.

5.1.7 Random pose validation

The code in the appendix C.0.4 shows that the position of selected trains and distractors is sampled randomly. However, not all the randomly sampled positions are valid, since the target models can be outside of the camera frustum or their pose can make them collide with other models present in the scene.

In order to discard the non-valid poses that are randomly sampled for the selected models in the scene, a series of checks are carried out, ensuring that the resulting poses are valid (within the frustum and with no collisions). Moreover, the implementation avoids using Blender operations (such as moving a model to a position or applying a camera transformation) in order to be faster and more efficient. The code uses the 3D bounding boxes (bbox) of the selected models to check whether their corners fall within the camera frustum or if they intersect with other bbox. When a new random pose is sampled, the bbox that the model would have in that candidate pose is calculated and used for performing the frustum and collision checkings.

The three conditions that are checked for testing a candidate pose are shown in the snippet 5.1.7. The first test is used as a quick check for easily discarding non-valid poses as it tests whether the location of the candidate pose falls within the frustum or not. The second test repeats the same operation but for each of the corners of the candidate bbox in the sampled pose to make sure that the object is not mostly out of the camera frustum. The last test ensures that the candidate bbox does not collide with previously accepted poses.

Snippet 5.1.7: Part of the code for sampling a valid pose.

```
# Placement logic with a max attempt count to prevent infinite loops
for i, model in enumerate(sampled_models):
    for attempt in range(max_attempts):
        success = True

        location, rotation = func(model, *func_args)

        # Checks whether the sampled location falls within the camera frustum or not.
        if success and frustum_check and not
        ↪  camera_utils.camera_frustum.is_point_in_frustum(camera_pose, location):
            success = False

        # A more advance camera frustum check! (Checks whether all the corners of the
        ↪  bbox of the model are inside of the camera frustum):
```

```
        if success and frustum_check and not
        ↪  camera_utils.camera_frustum.is_bbox_in_frustum(camera_pose, model, location,
        ↪  rotation, min_corners=6):
            success = False

        # Checks whether the sampled location intersects with other boundingboxes (SAT
        ↪  and AABB tests).
        obj = collisions_utils.PlacedModel(model, location, rotation)
        if success and collisions_utils.CollisionChecker.check_bbox_intersect(obj,
        ↪  [*placed_models, *samples_placed], sphere_check, AABB_check, SAT_check):
            success = False

        if success:
            break

    if success:
        samples_placed.append(obj)
        new_placed_models.append(obj)
```

If any of these checks fail, the pose will be randomised again until a successful pose is achieved or the maximum number of attempts is reached.

The test for detecting collisions between two bbox is composed of three parts (two broad-phase tests and one narrow-phase test) as can be seen in the snippet 5.1.8.

Snippet 5.1.8 Part of the code for checking whether the bounding box of an object intersects with the bounding box of another

```
for obj2 in placed_models:
    # Check if the spheres intersect (Broad-Phase Detection):
    if sphere_check and not CollisionChecker.sphere_intersect(obj1, obj2):
        continue

    # if spheres intersect, use AABB test (Broad-Phase Detection):
    elif AABB_check and not AABBCollisionTest.AABB_intersect(obj1, obj2):
        continue

    # If AABBs overlap, use the SAT test (Narrow-Phase Detection):
    elif SAT_check and not SATCollisionTest.SAT_intersect(obj1, obj2):
        continue

    return True # An intersection was found.

return False # No intersection detected.
```

5.1.8 Cloud computing

To take full advantage of scalable GPU resources, the entire render pipeline was deployed on Databricks [51] by installing the developed SynthRender module in a cluster running in Databricks. This setup makes use of the resources available on the cloud so that no local machine is needed.

Cluster configuration. A dedicated job cluster is defined with the following key parameters:

- `node_type_id = "Standard_NC64as_T4_v3"` (4 NVIDIA Tesla T4 GPU with 440 GB Memory and 64 Cores)
- `init_scripts:` installs requirements from the requirements.txt

Once the cluster is up, the code is executed from a notebook which calls the main.py script while passing the "-g 4" argument to indicate the code that it should use the four GPUs available in the cluster for rendering. This splits the assigned frame range into sub-intervals so that each GPU-bound process renders its segment in parallel.

5.2 Data Augmentation and Filtering Pipeline

Both the data augmentation and filtering parts of the pipeline were implemented using the render pipeline as the baseline. Because of that, they work with the generated HDF5 files that contain all the annotations for the train models.

5.2.1 Data Augmentation

As previously mentioned in the section 4.3.1, the data augmentation has been implemented with SDXL. More specifically the SDXL pipeline is provided by hugging face [52] as it allows for loading multiple ControlNet for conditioning the output of the model and the use of IP adapters.

The snippet 5.2.1 shows how the created class SDXL is used to define each of the models that will be used. loads the SDXL model and its ControlNet to be later used during the inference.

Snippet 5.2.1 Simplified example of the SDXL code

```
class SDXL(_StableDiffusion):
    _MODEL_PATH         = "stabilityai/stable-diffusion-xl-base-1.0"
    _Pipeline           = StableDiffusionXLControlNetPipeline
    _ControlNetModel    = ControlNetModel
    _vae                = "madebyollin/sdxl-vae-fp16-fix"

    _IpAdapter = {
        "model":       "h94/IP-Adapter",
        "subfolder":   "sdxl_models",
        "weight_name": "ip-adapter_sdxl.bin"
    }

    class CTRLN_TYPES(StrEnum):
        DEPTH = "diffusers/controlnet-depth-sdxl-1.0"
        CANNY = "diffusers/controlnet-canny-sdxl-1.0"
```

Once an instance of the SDXL class has been created, the models are loaded with the class method *_load_pipe* as shown in the snippet 5.2.2

Snippet 5.2.2 Loading of SDXL pipeline with ControlNet and IP-Adapter

```
def _load_pipe(self):
    self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    if self._vae is not None:
        vae = AutoencoderKL.from_pretrained(self._vae, torch_dtype=torch.float16)
    else:
        vae = None

    # Loading stable-diffusion with controlnets:
    self.pipe = self._Pipeline.from_pretrained(
        self._MODEL_PATH,
        controlnet=self.controlnets,
        torch_dtype=torch.float16,
        use_safetensors=True,
        vae=vae,
        variant="fp16"
    ).to(self.device)

    if self.ipAdapter is not None:
        self.pipe.load_ip_adapter(self.ipAdapter["model"],
        ↪  subfolder=self.ipAdapter["subfolder"],
        ↪  weight_name=self.ipAdapter["weight_name"])
```

This allows the configuration to be loaded on the GPU or CPU (depending on the selected device) and get ready for starting the inference for a specific frame. The augmentation of a rendered image does the following steps:

1. The HDF5 file containing all the annotations of the frames is loaded and its data (RGB, semantic segmentation masks, depth map) is extracted.
2. A canny edge image is obtained from the segmentation masks for each of the train models present in the frame using the CV2 implementation [45].
3. Both the depth map and the calculated canny edges are passed as conditions for the ControlNet loaded in the SDXL pipeline. The depth map contains information on distances between the elements in the original scene and is used to constrain the layout of the newly generated image. Whereas the canny image is used to refine the parts of the image at the location of the train models, so that the generated part looks more like the original model, simplifying the later image composting.
4. The RGB image of the target frame is passed as an input to the IP-Adapter so that the model has a context of the appearance of the original scene.

5. Finally, the semantic segmentation masks are used as binary masks for composting the train objects from the original RGB image into the new one.

Based on the discussion of a previous work [53], better results were obtained by using randomised prompts for synthetic rendered data augmentation with SDXL since it allows for a more complex and variated image generation. Hence, the following random prompts were used:

- ”A scene on the moon, craters, astronaut”
- ”A dense forest with sunlight filtering through the trees”
- ”A snow-covered mountain range with clear blue skies”
- ”An underwater coral reef teeming with fish”
- ”A peaceful meadow with wildflowers and tall grass swaying in the breeze”
- ”A peaceful beach with waves gently lapping the shore”
- ”A desert landscape with sand dunes and clear night sky”
- ”A grassy hillside with grazing animals under a bright blue sky”

Negative prompt: ”blurry, ugly, bad quality, worst quality, low quality, worst quality, deformed, distorted, disfigured, motion smear, motion artefacts, monochrome”

Figure 5.7 shows the result of the inference for a rendered frame as well as all the conditions used.

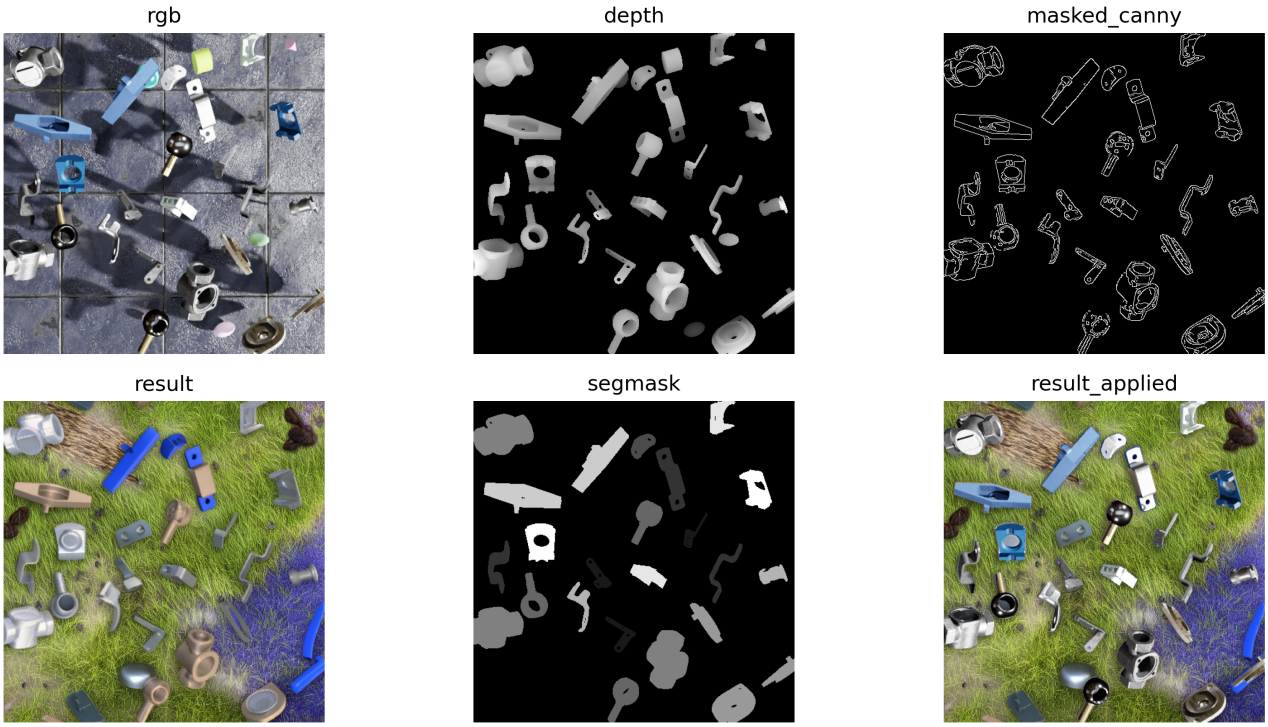


Figure 5.7: Robotics dataset augmented with prompt: ”A peaceful meadow with wildflowers and tall grass swaying in the breeze”

Furthermore, the following context-aware prompts were used for each of the three datasets in section 2.4:

- **Automotive:** "Realistic picture of an industrial scene with boxes, metallic pieces and tools"
- **Robotics:** "Cenital perspective of multiple pieces, items and shapes placed on top of a surface, cluttered"
- **T-LESS:** "Realistic picture of many pieces and items placed on a table with good light and composition"

Finally, as the train models keep their original position in the new augmented images, the annotation of their bounding boxes (which was generated by the render pipeline) can be obtained from their original frames.

5.2.2 Data Filtering

The data filtering takes the same approach in both brightness and hashing cases. The generated frames used for training are sorted by their minimum distance to a reference set. The distance can be the average brightness or the Hamming distance depending on the case.

Snippet 5.2.3 shows how the images are sorted by brightness. Furthermore, the function *get_brightness* has been defined under the class *brithnessFiltering* for calculating the brightness value of a certain image as can be seen in the snippet 5.2.4.

Snippet 5.2.3 Code for brightness sorting

```
# Gets the median of each image's brightness:
print("Getting test_set images median brightness...")
test_median = brightnessFiltering.get_images_median(test_imgs_paths)

# Gets the image brighness of the train set:
print("Getting rendered images brightness...")
for path in tqdm(images_to_sort, bar_format=f"\tImages processed: {fmt}", disable=False,
 ↪  **keyargs):
    remainder_imgs_bright.append((path, brightnessFiltering.get_brightness(path)))
```

Snippet 5.2.4 Code for brightness apply

```
def get_brightness(image:np.ndarray|str):
    if isinstance(image, str):
        image = cv2.imread(image)

    return image.mean() / 255
```

On the other hand, perceptual hashing is calculated using the Python module imagehash [54] which calculates the hash of a given image with the function *average_hash*. The snippet 5.2.5 shows how

the module is used for getting the hash of each image and then sorted based on their mean Hamming distance to the reference set.

Snippet 5.2.5: Code showing how perceptual hashing and Hamming distance are calculated.

```
# Gets the median of each image's brightness:
print("Getting ref_set images median hash...")
ref_imgs_hashes = [imagehash.average_hash(Image.open(path)) for path in ref_imgs_paths]

# Gets the image brightness of the train set:
print("Getting rendered images hash...")
remainder_imgs_hash = []
for render_path in tqdm(images_to_sort, bar_format=f"\tImages processed: {fmt}",
↪  **keyargs):
    mean = 0
    hash = imagehash.average_hash(Image.open(render_path))

    for ref_hash in ref_imgs_hashes:
        mean += abs(hash - test_hash)

    mean /= len(test_imgs_hashes)

    remainder_imgs_hash.append((render_path, mean))

remainder_imgs_hash = sorted(remainder_imgs_hash, key=lambda x: x[1])
```

5.3 Configuration for Each Dataset

This section will describe how the pipeline has been configured in order to generate data for each of the datasets described in 2.4. The pipeline is configured through a configuration file and allows to change things such as the camera position, target pose or number of distractors for each scene. An example of a configuration file can be seen in Appendix B.

5.3.1 Automotive

The Automotive dataset is composed of three target objects which appear in multiple locations and orientations, with no defined region or surface to be placed. Furthermore, different camera positions and distractors are used so that each image in the test set is unique.

In order to adapt the pipeline to reproduce similar images to the test set while closing the sim-to-real gap, the configuration in Appendix B has been modified so that the target or train models spawn around the empty object in mid-air so simulate different positions in x, y and z. Furthermore, the camera orbits the empty object as well, moving around the whole scene. Regarding the distractors, their location is randomised for the x and y coordinates but the z is set to be always at 0 m so that they appear to be placed on the floor without having to simulate their physics. Similarly, their rotation is randomised only over the z-axis so that they can have different poses.

In order to add background variations, a plane in which the material randomly changes for each frame is set, which is also used to define the limits at which the models can spawn. Table 5.2 shows the main values used in the configuration file for randomising the pose of the models in the scene, while figure 5.8 shows how a randomly generated scene for the Automotive dataset looks like.

Category	Parameter	Value
Misc	seed	0
Misc	cycles_samples	0
physics	simulate_physics	False
world	background_light_strength	[0.01, 0.5]
world	random_backgrounds	True
plane	x_length	6
plane	y_length	6
lights	light_size	0.25
lights	distance	6
lights	light_intensity	[0, 160]
lights	randomize_color	True
lights	exponential_lights	False
camera	f-stop	[2, 16]
camera	sensor_size	36
camera	pos_shell_radius	[2, 3]
camera	pos_shell_elevation	[10, 90]
empty_object	center	[0, 0, 1]
empty_object	pos_shell_radius	[0, 1.5]
empty_object	pos_shell_elevation	[0, 90]

Category	Parameter	Value
models-trains	sample_size	[-1, -1]
models-trains	n_copies	0
models-trains	pos_min	[-0.5, -0.5, -0.5]
models-trains	pos_max	[0.5, 0.5, 0.5]
models-trains	rot_min	[0, 0, 0]
models-trains	rot_max	[3.1415, 3.1415, 3.1415]
models-distractors	sample_size	[-1, -1]
models-distractors	n_copies	0
models-distractors	pos_min	[-1.5, -1.5, 0]
models-distractors	pos_max	[1.5, 1.5, 0]
models-distractors	rot_min	[0, 0, 0]
models-distractors	rot_max	[0, 0, 3.1415]
fake_models-trains	sample_size	[-1, -1]
fake_models-trains	n_copies	0
fake_models-trains	simple_trains	4
fake_models-trains	similar_trains	0
fake_models-distractors	sample_size	[-1, -1]
fake_models-distractors	n_copies	0
fake_models-distractors	simple_distractors	10

Table 5.2: Configuration parameters for the Automotive dataset.

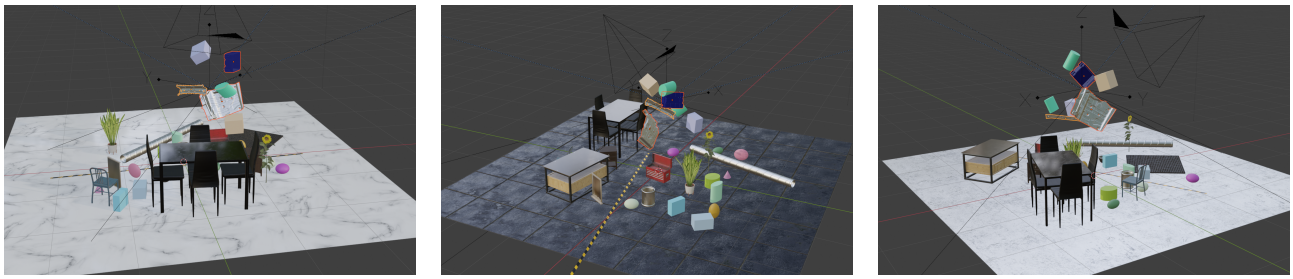


Figure 5.8: Examples of scenes generated for the Automotive dataset visualised using the debug mode of the pipeline. Target models are highlighted

5.3.2 Robotics

The Robotics dataset is composed of 10 target objects which appear in multiple positions and orientations but are always placed on a table. The background does not change and the camera is fixed at the top looking towards the center.

In order to adapt the pipeline to reproduce similar images to the test set while closing the sim-to-real gap, the configuration in Appendix B has been modified so that the target or train models spawn around the empty object, fixing its height very close to the floor, while their x and y position are randomised. Furthermore, the camera has been set to be fixed at the top of the scene rather than orbit around it. Regarding the distractors, their location is randomised for the x and y coordinates but the z is set to be always at 0 m so that they appear to be placed on the floor without having to simulate their

physics. Similarly, their rotation is randomised only over the z-axis so that they can have different poses.

Although there are no background changes in the test set, the plane material is yet randomised in order to make the train set more diverse. Table 5.3 shows the main values used in the configuration file for randomising the pose of the models in the scene, while figure 5.9 shows how a randomly generated scene for the Robotics dataset looks like.

Category	Parameter	Value
Misc	seed	0
Misc	cycles_samples	0
physics	simulate_physics	False
world	background_light_strength	[0.01, 0.5]
world	random_backgrounds	True
plane	x_length	1
plane	y_length	1
lights	light_size	0.25
lights	distance	6
lights	light_intensity	[0, 160]
lights	randomize_color	True
lights	exponential_lights	False
camera	f-stop	[2, 16]
camera	sensor_size	36
camera	pos_shell_radius	[0.5, 0.5]
camera	pos_shell_elevation	[90, 90]
empty_object	center	[0, 0, 0.04]
empty_object	pos_shell_radius	[0, 0]
empty_object	pos_shell_elevation	[0, 0]

Category	Parameter	Value
models-trains	sample_size	[10, -1]
models-trains	n_copies	2
models-trains	pos_min	[-0.3, -0.3, 0]
models-trains	pos_max	[0.3, 0.3, 0]
models-trains	rot_min	[0, 0, 0]
models-trains	rot_max	[3.1415, 3.1415, 3.1415]
models-distractors	sample_size	[5, -1]
models-distractors	n_copies	0
models-distractors	pos_min	[-0.4, -0.4, 0]
models-distractors	pos_max	[0.4, 0.4, 0]
models-distractors	rot_min	[0, 0, 0]
models-distractors	rot_max	[0, 0, 3.1415]
fake_models-trains	sample_size	[3, -1]
fake_models-trains	n_copies	0
fake_models-trains	simple_trains	10
fake_models-trains	similar_trains	0
fake_models-distractors	sample_size	[-1, -1]
fake_models-distractors	n_copies	0
fake_models-distractors	simple_distractors	20

Table 5.3: Configuration parameters for the Robotics dataset.

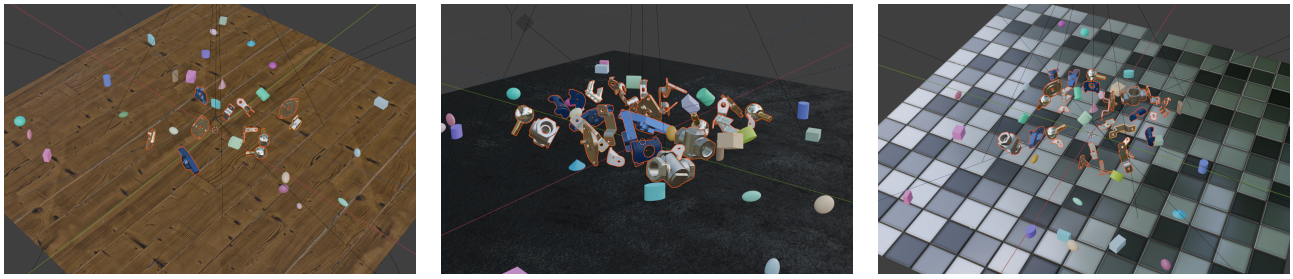


Figure 5.9: Examples of scenes generated for the Robotics dataset visualised using the debug mode of the pipeline. Target models are highlighted

5.3.3 T-LESS

The T-LESS dataset is composed of 30 target objects, but as explained in section 6, only 10 will be used. The target models appear in multiple positions and orientations but are always placed on a table in a similar way to the Robotics dataset, but in this case, the background does change and the camera orbit around the center of the scene rather than being fixed at the top.

In order to adapt the pipeline to reproduce similar images to the test set while closing the sim-to-real gap, the configuration in Appendix B has been modified so that the target or train models spawn

around the empty object, fixing its height very close to the floor, while their x and y position are randomised (similar behaviour to Robotics dataset). Furthermore, the camera has been set to orbit around the empty object so that multiple viewpoints of the scene and the models can be seen. Regarding the distractors, their location is randomised for the x and y coordinates but the z is set to be always at 0 m so that they appear to be placed on the floor without having to simulate their physics. Similarly, their rotation is randomised only over the z-axis so that they can have different poses.

Furthermore, the plane material is randomised in order to make the train set more diverse. Table 5.4 shows the main values used in the configuration file for randomising the pose of the models in the scene, while figure 5.10 shows how a randomly generated scene for the Robotics dataset looks like.

Category	Parameter	Value
Misc	seed	0
Misc	cycles_samples	0
physics	simulate_physics	False
world	background_light_strength	[0.01, 0.5]
world	random_backgrounds	True
plane	x_length	1
plane	y_length	1
lights	light_size	0.25
lights	distance	6
lights	light_intensity	[0, 160]
lights	randomize_color	True
lights	exponential_lights	False
camera	f-stop	[4, 16]
camera	sensor_size	36
camera	pos_shell_radius	[0.5, 0.5]
camera	pos_shell_elevation	[90, 90]
empty_object	center	[0, 0, 0.1]
empty_object	pos_shell_radius	[0, 0]
empty_object	pos_shell_elevation	[0, 0]

Category	Parameter	Value
models-trains	sample_size	[5, -1]
models-trains	n_copies	1
models-trains	pos_min	[-0.3, -0.3, 0]
models-trains	pos_max	[0.3, 0.3, 0]
models-trains	rot_min	[0, 0, 0]
models-trains	rot_max	[3.1415, 3.1415, 3.1415]
models-distractors	sample_size	[10, -1]
models-distractors	n_copies	0
models-distractors	pos_min	[-0.4, -0.4, 0]
models-distractors	pos_max	[0.4, 0.4, 0]
models-distractors	rot_min	[0, 0, 0]
models-distractors	rot_max	[0, 0, 3.1415]
fake_models-trains	sample_size	[2, 5]
fake_models-trains	n_copies	0
fake_models-trains	simple_trains	10
fake_models-trains	similar_trains	0
fake_models-distractors	sample_size	[-1, -1]
fake_models-distractors	n_copies	0
fake_models-distractors	simple_distractors	10

Table 5.4: Configuration parameters for the Robotics dataset.

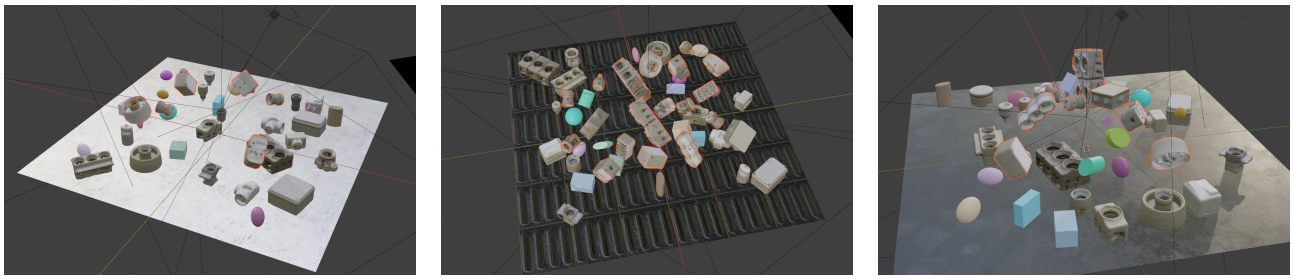


Figure 5.10: Examples of scenes generated for the T-LESS dataset visualised using the debug mode of the pipeline. Target models are highlighted

Chapter 6

Experiments

This chapter will describe the tests and experiments conducted in order to evaluate if the established objectives in Section 3.2 were fulfilled or not. The results of such tests will be addressed and further discussed in chapter 7, to provide insights into their implementation and overall impact on the framework.

The evaluation of the T-LESS dataset differs from the Automotive and Robotics test sets in two key respects. First, whereas the former two datasets consist of independently sampled images, T-LESS provides video sequences: three sensors (Kinect, PrimeSense, Canon) each capture 20 distinct scenes of 504 consecutive frames, for a total of 30,240 test images. However, these frames are time-consistent and exhibit minimal inter-frame variation (see Figure 6.1), offering little additional diversity.

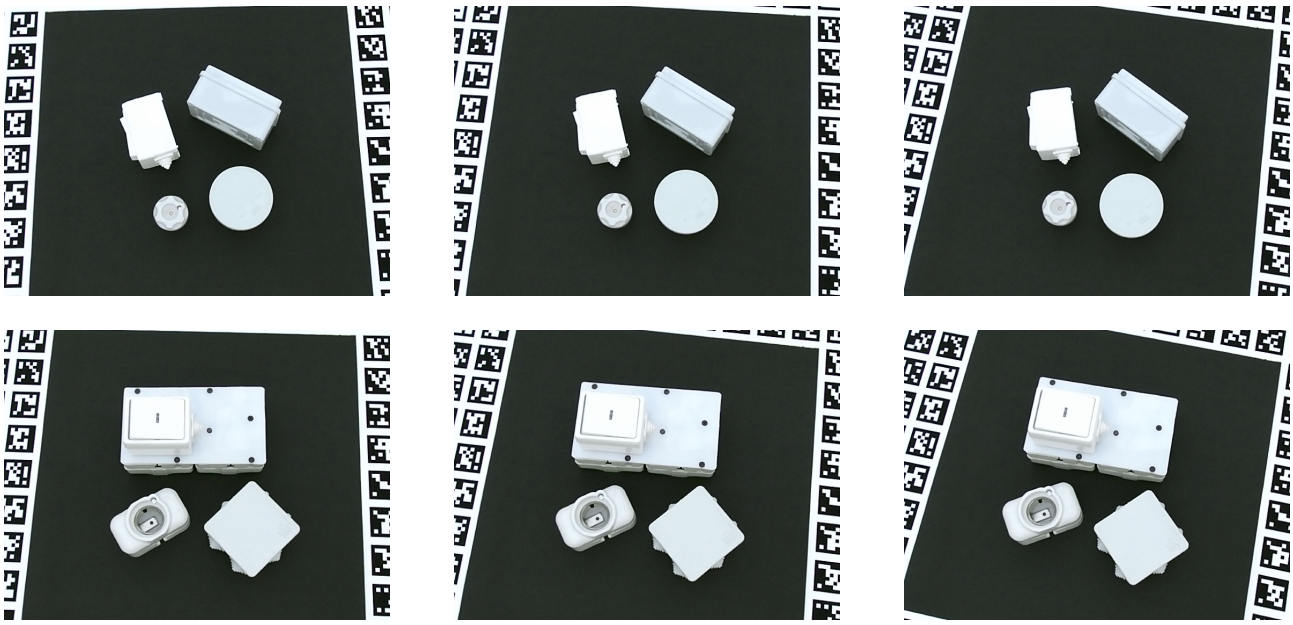


Figure 6.1: Example of T-LESS time consistency for Kinect camera. **Top row:** First three frames of scene 01 (0, 1, 2); **Bottom row:** First three frames of scene 04 (0, 1, 2).

In order to reduce the amount of redundant images while increasing data diversity, only one frame per 100 is used for each scene, removing redundant frames due to the continuous movement of the

camera. This approach reduces the number of test frames for each scene from 504 to only 6, being the resulting test set formed by 360 frames (120 per camera). Furthermore, in order to limit training effort and mirror the ten-class detection task for the Robotics dataset, only 10 out of the 30 models are used for detection. The remaining 20 models are still present in the scene but as distractors rather than target models. The selected model IDs are 2, 5, 8, 11, 15, 18, 20, 23, 26, 28 which as can be seen in the original T-LESS publication [44], share similar geometries and textures with the rest of the distractors, making the detection more challenging. The scenes wherein none of the selected models are present (5, 17, 18) have been removed from the test set as well, leaving a total of 17 scenes and 102 different images (17 scenes with 6 frames each) per camera.

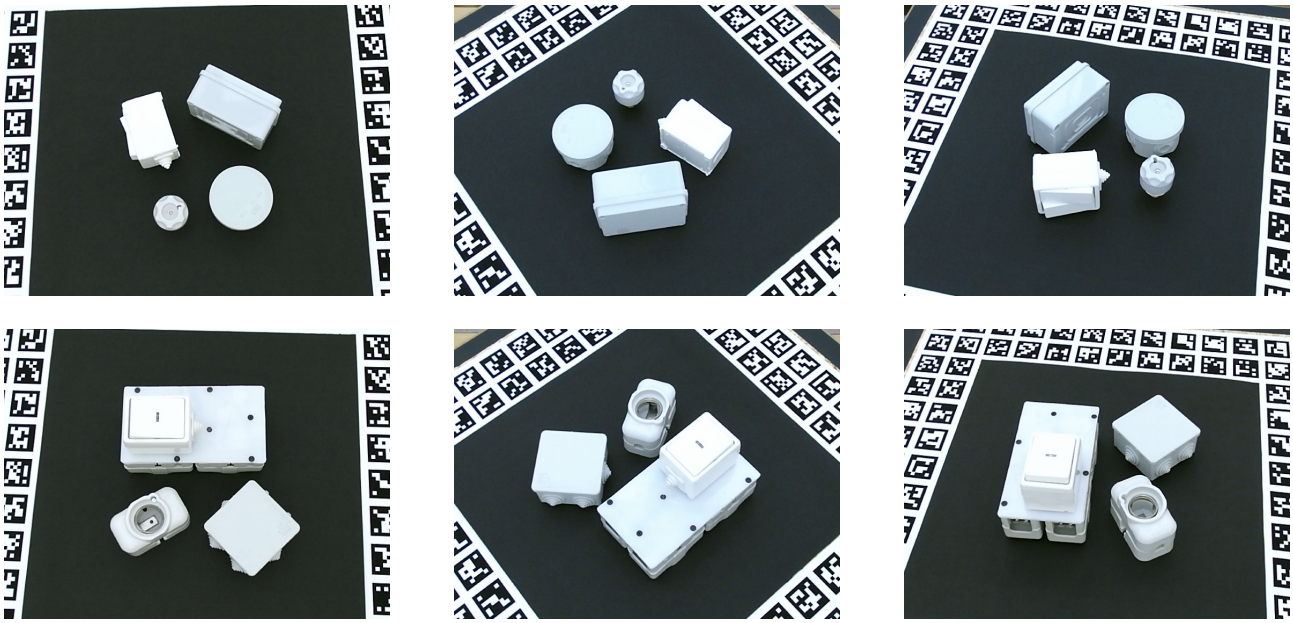


Figure 6.2: Example of the reduced T-LESS dataset for the Kinect camera. **Top row:** First three frames of scene 01 (0, 100, 200); **Bottom row:** First three frames of scene 04 (0, 100, 200).

To summarise, the structure of each dataset can be seen in the following table 6.1

Dataset	Type	Classes	Amount of images	Resolution
Automotive	Proprietary	3	50	512×512
Robotics	Public	10	190	1280×720
T-LESS	Public	10	306 (102×3 cameras)	720×50 and 2560×1920

Table 6.1: Details and composition of each used dataset

YOLOv8n has been used as the detection model for benchmarking the generated synthetic data. The training on the local machine has been done with: 500 epochs, an image resolution of 512, 720 or 1024 depending on the data, a learning rate of 0.001, a patience of 30 epochs and a batch size of 16. Furthermore, the local machine running the tests and in which the project has been developed has the following technical specifications:

Component	Specification
Operating System	Ubuntu 22.04.5 LTS
CPU	Intel® Core™ i9-14900K (32 cores) @ 6.00 GHz
Memory	125.35 GB RAM
Storage	2 TB SSD
GPU	NVIDIA GeForce RTX 4090
Video Memory	24 GB VRAM

Table 6.2: Technical Specifications of the Local Test Machine

6.1 Old vs New render pipelines

This experiment evaluates the new render pipeline developed in this thesis against the original version described in [38]. Both pipelines were run on the same computer equipped with an NVIDIA GTX 1080 Ti, and training was performed using the YOLOv8 hyperparameters: the batch size of 102, the learning rate of 0.001, 200 epochs, early stopping patience of 50, the AdamW optimizer, and a cosine learning-rate scheduler.

The original pipeline utilised only one of the four NVIDIA GTX 1080 Ti GPUs available in the computer. To assess the impact of the new parallelization scheme, two synthetic datasets are generated: one rendered using a single GPU and the other rendered across all four GPUs by splitting the render interval between each GPU. Each dataset consists of 1,000 frames at a resolution of 512×512 pixels.

Pipeline used	Nº GPUs	Creation time	Seconds/Frame	$\mathbf{mAP}_{50}$	$\mathbf{mAP}_{50\text{–}95}$
Old Pipeline	1	06:03:00	21.78	-	72.01 %
New Pipeline	1	01:02:30	3.75	95.1231 %	81.75 %
New Pipeline	4	00:26:21	1.58	95.1231 %	81.75 %

Table 6.3: Comparison of rendering performance and mAP for the old and new render pipelines using 1,000 frames at 512×512 resolution.

6.2 Dataset resolution

The following experiment evaluates how the resolution of the generated synthetic training data influence the times and resulting mean Average Precision (mAP) for YOLOv8n. For each of the three datasets (Automotive, Robotics, and T-LESS), 4,000 synthetic frames were generated at three resolutions: 512×512, 720×720, and 1024×1024.

Furthermore, once each synthetic dataset was created the images were split into a 80/20 data distribution, wherein 80% (3200 images) were used for training the model and the remaining 20% (800 images) were used for validation.

Dataset	Resolution	Creation time	Training time	mAP_{50}	$mAP_{50–95}$
Automotive	512×512	01:29:00	00:23:42	98.3206 %	**88.1410 %**
	720×720	02:24:50	00:40:30	98.3106 %	88.0867 %
	1024×1024	04:09:31	00:52:27	97.5851 %	83.8497 %
Robotics	512×512	01:22:22	00:37:42	97.2872 %	66.4223 %
	720×720	02:14:01	01:08:58	97.4772 %	68.6196 %
	1024×1024	04:03:40	02:02:54	95.2329 %	**68.6489 %**
T-LESS	512×512	01:20:57	00:40:39	23.5164 %	17.9741 %
	720×720	02:09:50	01:05:24	25.2790 %	19.9499 %
	1024×1024	03:57:21	01:23:35	27.1035 %	**21.2414 %**

Table 6.4: Results for the synthetic data on three different datasets

6.3 Dataset size, augmentation and filtering

This test evaluates the effect of the number of frames on the resulting mAP while using data augmentation and filtering techniques to improve the already generated synthetic data. Each technique is applied and the resulting map with YOLOv8 is shown. For each of the three datasets (Automotive, Robotics, and T-LESS), only 1,000 of the previously generated 4,000 synthetic frames with a resolution of 512×512 are used. To isolate the effect of the training size, the following steps are followed:

1. **Fixed train/val split:** The same train/validation split used in the previous test is used. Having 3,200 train images (80%) and 800 validation images (20%).

2. **Reduced train set:** From the 80% of train images, only a reduced amount will be used. The reduced train set is composed of 1,000 train images randomly sampled from the original 3,200.

3. **Reference set:** From the test set images, 10% of the frames are randomly removed and added to a new set called reference set (5 images from Automotive, 19 from Robotics and 30 from T-LESS). This set is only used to get contextual information on the domain of the test images. The images in the reference set are not used for validation or training.

4. **Train set size:** In order to measure the impact of the train set size, it starts with a fixed amount of 250 rendered frames which will be increased in steps of 75 frames up to 1,000. Each larger subset contains all the images from the smaller subset plus the next batch of new 75 frames. Different methods are used for selecting the images to add: Stable Diffusion XL (SDXL), Brightness, and Perceptual Hashing.

    - **Base:** Each extra batch of 75 images is taken using the same randomisation seed on the 1,000 train images.

    - **SDXL:** Each extra batch of 75 images are generated using SDXL for augmenting the fixed 250 images up to 3 times to sum up to 1,000 frames in total.

- **Brightness:** Each extra batch of 75 images is taken by sorting the remaining 750 that have not been fixed by brightness similarity (see equation 4.2) with the reference set.

- **Perceptual Hashing:** Each extra batch of 75 images is taken by sorting the remaining 750 that have not been fixed by calculating their Hamming distance (see equation 4.3) to the reference set using the ImageHash library [54] to calculate the image hashes.

5. **Training strategy:** For each subset of size $N \in [250, 1000]$, YOLOv8 is trained and evaluated on the fixed validation set.

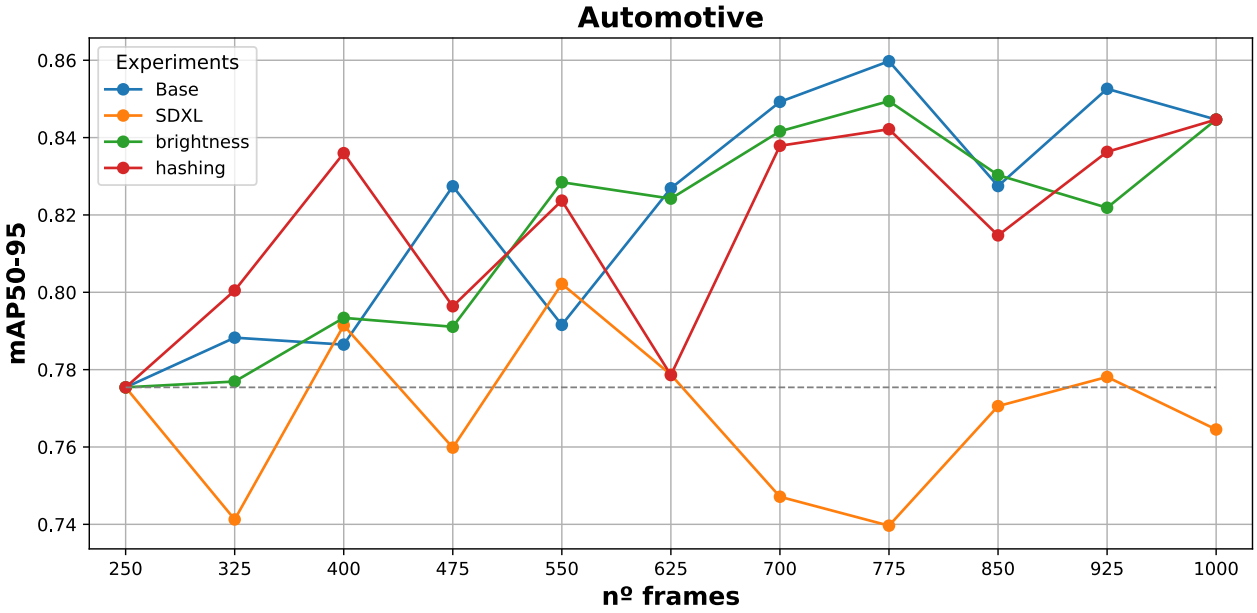


Figure 6.3: Training YOLOv8 on Automotive dataset with different train sizes and approaches (base, SDXL, brightness and Hashing)

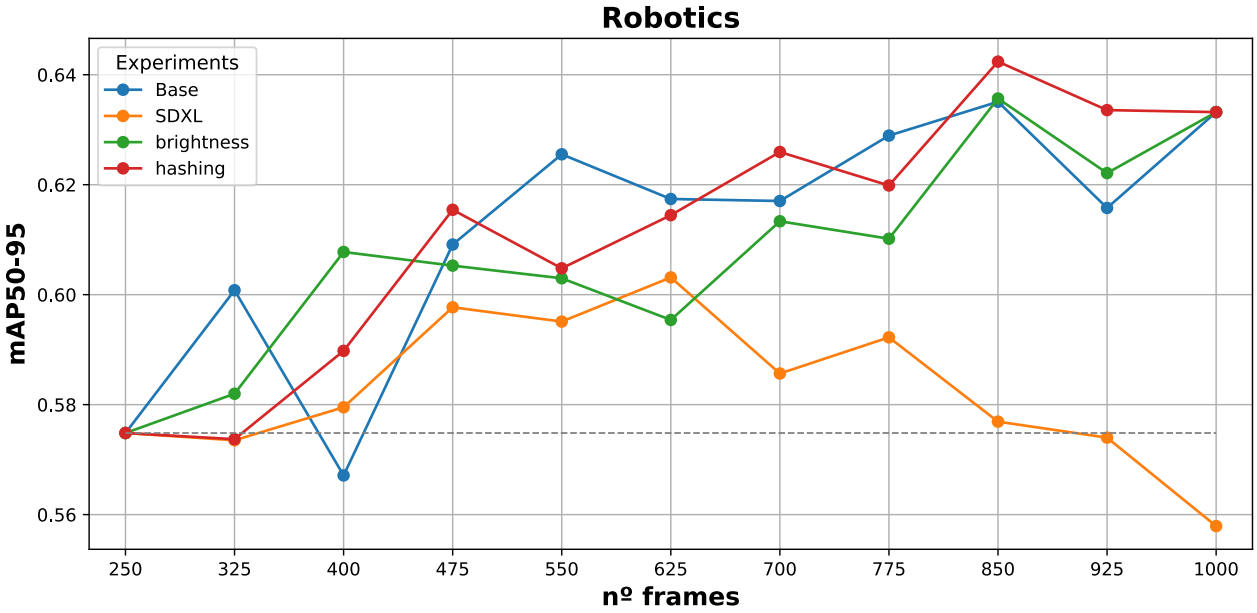


Figure 6.4: Training YOLOv8 on Robotics dataset with different train sizes and approaches (base, SDXL, brightness and Hashing)

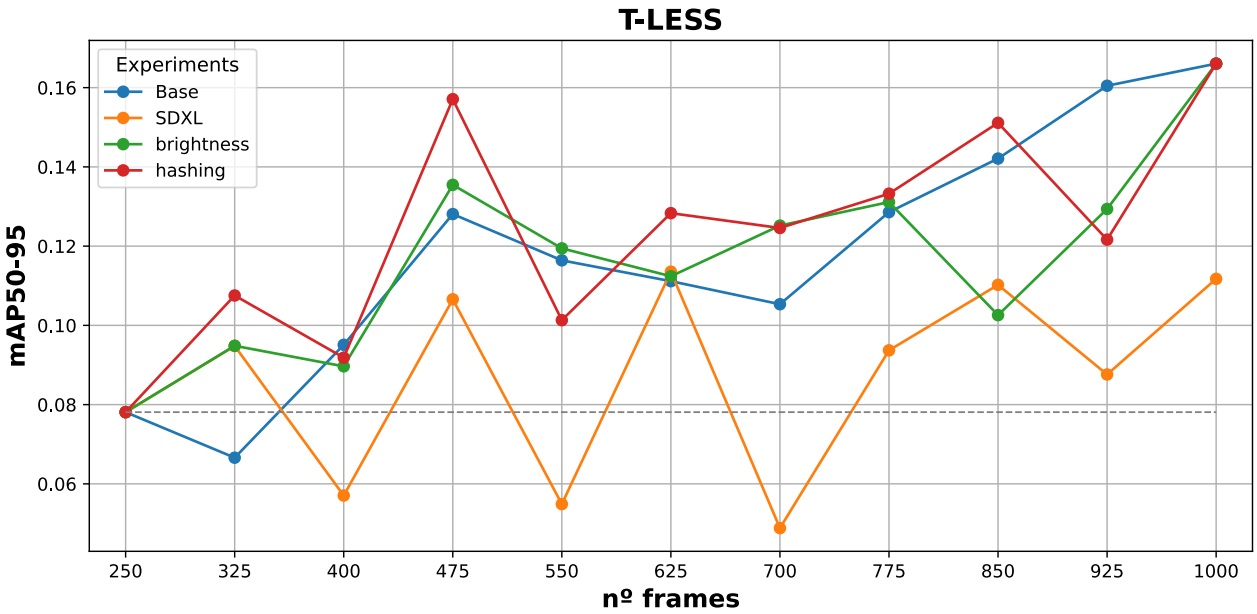


Figure 6.5: Training YOLOv8 on T-LESS with different train sizes and approaches (base, SDXL, brightness and Hashing)

6.4 Render times (local vs cloud)

The following experiment evaluates the time spent in creating a whole model trained on fully synthetic data using the local machine and the cloud. Creating a trained model includes the render time of synthetic data, the time annotating the dataset in YOLO format and the time training the model.

For the cloud setup, a Databricks compute with four T4 GPUs and 440GB of RAM is used for creating a synthetic dataset with 4,000 frames and at a resolution of 1024×1024. The results are then compared against those obtained with the local machine for the same dataset size and resolution in the test 6.2.

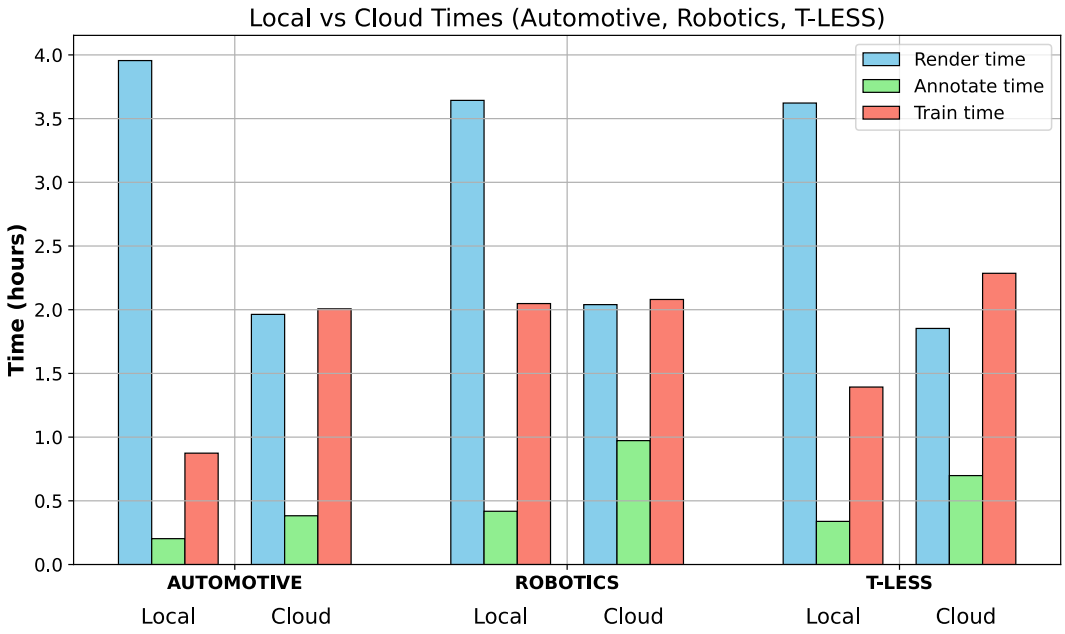


Figure 6.6: Times for each dataset on both local and cloud computing.

6.5 Ablation study

The render pipeline also includes options such as colour lights (RGB), exponential light intensity (see figure 5.3), and physic simulations for better adjusting the synthetically generated data depending on the use case. This test aims to evaluate how each of these options affects the result for each dataset. Nine new datasets (three for each dataset type) were created, each of which had a different simulation option. The resolution was kept at 512×512 for all of them so that the render and train times were kept simpler and the size is 4,000 frames.

Dataset	NONE	RGB	EXPONENTIAL	PHYSICS
Automotive	83.0307 %	**85.9837 %**	84.0850 %	72.3453 %
Robotics	50.1922 %	66.6678 %	52.2080 %	35.3129 %
T-LESS	12.5018 %	16.8714 %	14.5588 %	7.1233 %

Table 6.5: mAP50–95 results for each dataset type using each simulation option.

Dataset	PHYS + RGB	PHYS + EXP	RGB + EXP	PHYS + EXP + RGB
Automotive	78.2600 %	74.4812	84.8649 %	76.8107 %
Robotics	57.9633 %	37.6100	**66.7355 %**	56.5612 %
T-LESS	12.8391 %	7.3369	**21.9134 %**	9.6362 %

Table 6.5: mAP50–95 results for each dataset type using multiple options combined.

6.6 Zero-shot vs One-shot vs Few-shot

To test how having real images affects the performance of the detection model, a real image has been removed from the test set of each dataset (Automotive, Robotics, T-LESS) and added to their respective train set. This has been done for the previously generated dataset of 4,000 frames and 512×512 of resolution of the test 6.2.

Furthermore, few-shot has been tested as well by combining both the train set and the reference set previously introduced in 6.3 for having even more real examples available while training.

Real images added to the train sets are not present in the validation or test sets.

	Zero-shot		One-shot		Few-shot	
Dataset	**mAP50**	**mAP50-95**	**mAP50**	**mAP50-95**	**mAP50**	**mAP50-95**
Automotive	98.3206 %	88.1410 %	98.5921 %	88.4730 %	**99.0382 %**	**89.5444 %**
Robotics	97.2872 %	66.4223 %	97.4025 %	67.3637 %	**98.5541 %**	**81.3615 %**
T-LESS	23.5164 %	17.9741 %	25.9680 %	20.0246 %	**56.4422 %**	**46.1659 %**

Table 6.6: Measured mAP on various datasets trained on fully synthetic images (zero-shot), trained after adding one real image (one-shot), and trained after adding the images from the reference set (few-shot).

Chapter 7

Discussion

This chapter aims to discuss the developed pipeline for synthetic data generation, its features and limitations. Next, the tests and results obtained in chapter 6 will be discussed and evaluated as well as the project objectives. Lastly, suggestions for further development and improvement of the project will be proposed.

7.1 Overall Pipeline

The developed pipeline for synthetic data generation is efficient and flexible, allowing users to create different types of datasets using CAD models as input without requiring an expert level of programming skills or deep knowledge of Blender. That being said, a minimum understanding of how the render pipeline works is still recommended for better configuration and adaptation to the use case of the user.

The main part of the developed pipeline is the render pipeline SynthRender, which leverages the Cycles render engine from Blender in order to simulate and generate fully annotated synthetic data. Using a configuration file as an interface, it is possible to adjust parameters such as the lights, the poses in the scene and the materials loaded without having to open the Blender GUI. Although if debugging is needed to check up the randomised scenes, it is also possible to run the pipeline in debug mode for better control. Furthermore, a default Blender scene can be loaded to reproduce digital twins or desired environmental layouts.

Parallelisation is also possible if multiple GPUs are available as the pipeline can automatically start multiple instances of the render for distributing the charge. Moreover, as it is possible to just render within an interval of the total generated scenes, multiple computers can be used for even faster renderings as long as all of them use the same randomisation seed in the configuration file.

Furthermore, one of the strengths of the pipeline is its ability to generate pixel-perfect annotations of the target models, while also being able to select which parts of a model with multiple parts are annotated or rendered, outputting depth maps, segmentation masks and normal maps along with the rendered RGB if desired. These annotations can be used for creating datasets in YOLO, COCO, and BOP formats, and for applying different techniques such as augmentation through GenAI and filtering through brightness and perceptual hashing.

7.2 Testing Results

The results for the tests introduced in the chapter 6 will now be discussed in this section.

7.2.1 Old vs New render pipelines

The results in the comparison between the old and the new render pipelines demonstrate a significant performance improvement. While the old pipeline is able to render and annotate 1,000 frames in approximately six hours, the new one can do so in just one hour. In total, the new approach is **5.81 times faster** and reached an extra 9,74% in the mAP_{50-90}.

The improvement in performance can be attributed to several key optimisations such as the use of BlenderProc render functions, to easily set up the simultaneous generation of RGB images and their annotations, eliminating the need for multiple rendering phases as required in the old version. Moreover, the new version is more memory efficient as well, as it avoids the repeated load and storage in memory of hundreds of layouts and all its models. Enabling and disabling the render visibility of the models in the scene played a significant role as Blender only needed to load and work with the elements visible in a single scene, avoiding unnecessary computational overhead.

Furthermore, the new pipeline is also able to leverage multiple GPUs and/or computers for better distribution of the frames to render. The same 1,000 frames were rendered using 4 GPUs instead of just one. This makes the new render pipeline **13.74 times faster** than the old pipeline, and 2.37 times faster than using just a single GPU.

It is worth mentioning that the time for the parallelised version being 2.37 times faster instead of 4 times (the number of GPUs used) is due to having the whole render being executed in the same machine, as the same CPU is used for loading and processing the data of each GPU. However, if four computers with one GPU each were used, the render would then take less than 16 minutes.

7.2.2 Dataset resolution

The results for the resolution test show that, as one would expect, the time taken for creating a dataset (render + annotation times) increases with the resolution of the rendered images. Similarly, the mAP_{50-95} reached is also affected by the resolution, with the models that were trained on a similar resolution as their test set images getting the best results.

The time taken for training YOLOv8 is also affected by the resolution of the train images, highlighting the importance of correctly selecting the render resolution of the train images to avoid the waste of computational resources and time.

In the case of the Automotive dataset, the rate of image creation using the default setup with a GPU Nvidia RTX 4090 at a resolution of 512×512 is 1.34 s/frame. This rate is very close to the one achieved by the parallelised version of the previous test, indicating that it can be interesting to use setups with multiple cheaper GPUs, rather than an expensive one.

For the case of the Robotics dataset, Horváth et al. introduce their synthetic generation pipeline in the paper [43]. With a randomised resolution ranging from 640×640 to 1300×1300 and a size of 4,000 images, they report a mAP50 of 86.32% with YOLOv4. Although no direct comparison can be made since YOLOv8n was used for the thesis, the proposed render pipeline achieved a mAP_{50} of 97.29% with the same number of images and at a lower resolution (512×512).

Finally, the T-LESS dataset got the worst results of all the datasets evaluated. The reasons for this are

most likely due to the design of the dataset and the CAD models used:

- The T-LESS dataset is designed for pose estimation, even after removing time-consistent frames, the dataset has situations wherein target models are partially or mostly occluded.
- The CAD models used for generating synthetic data are the reconstructed versions, which contrary to the used by the other two use cases are approximations of the real objects rather than precise CAD models with manually selected textures.

Nevertheless, the test shows how increasing resolution does improve the mAP_{50-96}, scoring 21.24% and suggesting that higher dataset resolution can lead to even better results.

In conclusion, the results of time and mAP indicate that the pipeline can generate data capable of closing the sim-to-real gap and being used for training object detection models.

7.2.3 Dataset size, augmentation and filtering

The results for the dataset size test show how for every approach (base, SDXL, brightness and hashing) the increase in the number of frames leads to an improvement in the resulting mAP_{50-95} for the Automotive, Robotics and T-LESS datasets.

The results for the 'base' case prove that the detection model can train and improve its performance using only synthetic data generated through the render pipeline. Furthermore, the result shows that better results can be obtained depending on the size of the train set.

For the augmented images using SDXL, the results show that increasing the frames through the augmentation of an initial fixed amount (250 for these tests) up to 1,000 can only increase the mAP up to a maximum value which is lower than the other methods. The reason for this is the lack of new model poses, overfitting on those present in the initial 250 frames. Although adding new images with augmented backgrounds affects positively the mAP, the model quickly reaches a plateau since only background changes for the augmented images, while poses and annotations of the target objects remain the same. Furthermore, as the proportion of augmented images increases, the model lowers its mAP as the augmented images do not close the sim-to-real gap as well as the rendered ones, worsening the overall performance of the model.

Regarding the filtering methods ('brightness' and 'hashing') no better results for mAP are observed in comparison to the 'base' results. Nevertheless, they were able to reach a higher mAP with fewer images.

7.2.4 Render times (local vs cloud)

The results of the test for measuring the difference between local and cloud computing proved that it is possible to speed up the render times even more by making use of the GPUs available on the cloud.

However, training time got much slower on the cloud compared to the local training. This is because the I/O operations are much slower on the cloud implementation since the rendered data and training results are stored in an Azure container, needing extra time to fetch and upload the data that in the local machine is faster.

7.2.5 Ablation study

The results of the ablation study highlight the use of RGB lights as the most important part of getting good results. The reason for that is that whilst it is possible to create very accurate CAD models that

represent the geometry of the target objects in the simulation, the same cannot be straightforwardly done for their materials. For this reason, most of the time the pipeline will work with precise CAD models and just approximations of the colours and materials of the target models.

Under the assumption that the materials of the simulated target objects will never be the same as the ones from the real objects (and it would take too much time to get as close as possible), the pipeline can randomise the colour of the lights so that the detection model can learn to generalise the materials present on the target object.

This is also the reason why the T-LESS gives such lower results of mAP compared to the other datasets since, as discussed in section 7.2.2, the CAD models for the other two datasets are precise and its textures have been manually picked so that they can better benefit from the randomisation of the light colour changes. The T-LESS dataset instead, makes use of reconstructed CAD models, making the geometry and material of the CAD models an approximation.

The use of exponential lights alone also showed an improvement over using none of the possible modes for rendering, although the mode is not as impactful for the mAP as the RGB lights can be.

Regarding the physics, the reason why they got worse results than the none case is due to a limitation in the design of the render pipeline. Before placing the target objects in the scene, several checks are run in order to assess whether the candidate's pose is valid or not. One of them is the camera frustum check, which makes sure that the train models do not spawn out of the POV of the camera. However, these checks are performed right before running the physics simulation, which results in many objects falling and bouncing out of the camera frustum and being missing in the rendered scene.

Finally, the possible permutations of all the previous modes described have been tested as well to find out which could be the best combination. Table 6.5 shows that the best results were obtained with a combination of both exponential and RGB lights, getting very close to the results of test 6.2 with the highest resolution and even beating the best result for T-LESS with the maximum resolution.

7.2.6 Zero-shot vs One-shot

The final test focused on a comparison of the mAP_{50-95} results obtained between zero-shot, one-shot and few-shot. As can be seen in the table 6.6, just adding a single real image in the train set already improved the mAP_{50-95}, helping to close the sim-to-real gap even more.

Furthermore, the few-shot test was able to improve even more the best marks for all the datasets. Namely the mAP_{50-95} for the Robotics and the T-LESS, which received a considerable improvement by just adding a few images into the train set.

These results indicate that it is possible to improve even more a fully synthetic train set with little effort by just adding on a few real images, rather than creating an entire whole dataset and annotating it.

7.3 Evaluation of Project Objectives

In this section, the fulfilment of the project objectives outlined in chapter 3.2 will be evaluated.

Both the implementation and the discussion in 7.2.1 proved that the new render pipeline was successfully implemented and can make use of DR and GDR techniques for creating fully synthetic datasets, improving not only the previous render times but the mAP_{50-95} reached as well, fulfilling objective 3.2.

Furthermore, objectives 3.2 and 3.2 were proved to be fulfilled in both during the implementation and the subsection 5.1.2.1, wherein it is explained how CAD models are loaded, processed and randomised in the scene. All of this is adjusted and configured through a configuration file, which works as an interface between the user and the code so that no expert in Blender or coding is needed to make it work. All these factors result in an improved render pipeline, scalable and flexible.

Objective 3.2 has been implemented as well in the section 5.2, giving the user the possibility to not only render synthetic data but augment it using SDXL or filter the images through brightness or perceptual hashing so that the model is trained with few but more relevant images.

Finally, objective 3.2 has been fulfilled as well, as can be seen in subsection 7.2.1 and through all the section 7. Having tested the render pipeline against 3 different datasets: Automotive, Robotics and a reduced version of T-LESS.

7.4 Future Work

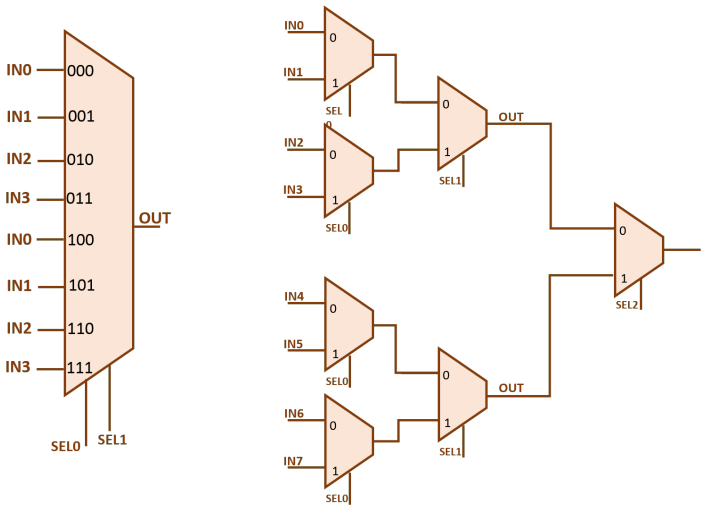
In this section, improvements and fixes will be discussed for a future work in order to tackle limitations and extend the work done.

7.4.1 Use of shader mixers for changing materials

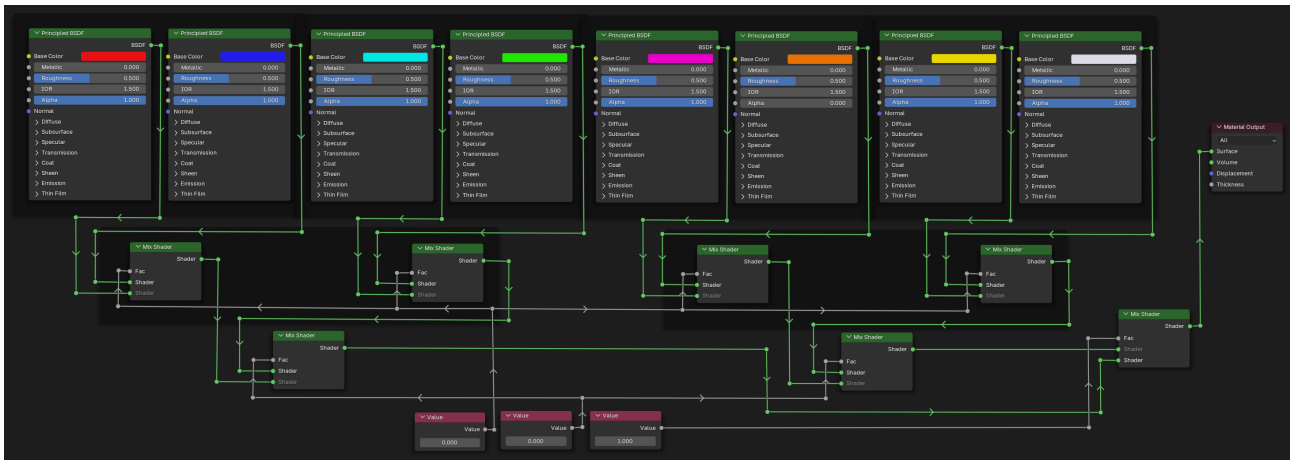
As already mentioned in 5.1.2.1, it is not possible to keyframe the material used by a model in Blender. So far the approach of this project for tackling this limitation has been creating multiple instances of the same object with a different material each, or render in intervals wherein each interval loads a different material (see the case for the HDRI background images in 5.1.2.2).

However, this means overloading the memory with replicated models just to show different materials during a render or having to restart and reload the data each time. Instead, a trick to avoid creating multiple copies of a model can be done.

An object material can be the result of combining two different materials using a node called mix shader [55]. The mix shader combines two materials with a percentage value called 'factor' (similar to image composting with an alpha channel). Unfortunately, the mix shader node only supports up to two materials, the reason why it has not been used since the idea would be to have many more materials available. However, multiple mix shaders can be combined to create the effect of having a mix shader with as many inputs as needed. Then, the 'factor' can be keyframed for each mix shader to change the material as desired. The behaviour mimics a multiplexer n:1 made out of multiplexers 2:1, as can be seen in figure 7.1.



(a) Schematics of a 8:1 multiplexer made with seven 2:1 multiplexers



(b) Example of a 8:1 mix shader made with seven 2:1 mix shaders

Figure 7.1: Example of a 8:1 mix shader based on a 8:1 MUX made of multiple 2:1 MUX.

The creation of this ”combined mix shader” could be automated through a Python script using the API of Blender and would solve the problem of having to create multiple instances of the same model to show a different material for it.

7.4.2 Test in other environments such as Nvidia Omniverse

Although the whole project has been developed for Blender, other promising environments such as Nvidia Omniverse could be used to test which one results in a better data generation pipeline.

Even if most of the functions have been implemented using the API of Blender, the core code is executed without calling any Blender function in order to avoid overheads that can slow down the randomisation of scenes. This includes sections such as object collision checking, and parameters randomisation among others, which could be used as the skeleton for implementations in other platforms.

7.4.3 Parallelise annotations as well

In this project, the main focus has been on speeding up the rendering of synthetic images as much as possible. As a result, a parallelised mode can be used to run multiple independent instances of Blender with each working on a different part of a render interval.

However, once the rendering part is done, data storage as HDF5 files and dataset creation in COCO and BOP formats make use of BlenderProc methods that work on a single thread. This makes this part of the project a good candidate for parallelisation through multiprocessing or concurrency to cut down the loading times of I/O operations.

Even more, right now the HDF5 files are stored once a render interval has finished, this means that the render does not continue until all the files have been stored. Depending on the size of the dataset this can take many minutes, slowing down the average s/frame even if the render has been improved to be faster.

A simple solution would be to start a worker on another thread that stores the rendered data on the disk while the main thread starts with the next render interval.

7.4.4 Randomise geometry

So far the discussion regarding randomisation improvements has been focused on randomising the material of the models. The reason for this is that it is easier to assume that the geometry of a CAD model is accurate and precise, and resembles the one from the real model. Because of this assumption, the focus was on the materials.

However, it could also happen that a CAD model does not resemble completely the target model or that the same model can have small variations, the results obtained for the T-LESS dataset proved what happens when a case like this happens.

A solution to this could be just adding to the pipeline the option to set a desired value to the attribute 'category_id' (this attribute is used for telling the pipeline which models should be annotated and with which id) of a model. With this new feature, two variations of the same model could be loaded and their 'category_id' attribute set to be the same, which would result in both of them being under the same category in the annotations.

This future work would improve the result for the T-LESS dataset, as it provides 'reproduced' models (the ones used for testing, were automatically generated) and the manually created CAD models. These manually created CAD models are exact CAD models of the reproduced ones but without textures. The new approach would allow loading both the texture-less geometrically accurate version of a model and its approximated less-accurate but textured version, allowing the model to learn how to generalise the geometry of the models as well.

7.4.5 Real images as ground truth

A problem noticed while working on this thesis is the lack of publicly available datasets for object detection which provide both real images and CAD models that can be used for synthetic data generation.

An interesting test for measuring how good a pipeline for generating synthetic data is would be comparing the performance of a model fully trained with real images against one fully trained on synthetics. However, this would require thousands of annotated real images which is one of the main

problems that synthetic data generation tries to solve.

Nevertheless, the benefit of having a ground truth made of real images could help to measure how well a pipeline can close the sim-to-real gap. This is also useful for challenging datasets, since for instance, knowing what mAP can be achieved through real images can help understand if the obtained mAP through synthetic images is very poor or if it is possible to get even higher sores.

Chapter 8

Conclusion

This thesis at Mercedes-Benz Group AG has been a valuable experience that has helped me develop both professionally and personally. The opportunity to keep working on new technologies helped me better understand fields such as artificial intelligence and mainly synthetic data generation. With it, what seemed like abstract concepts before now are part of my daily work, research and interest.

For this thesis, a synthetic data generation framework has been successfully implemented, improved and tested against multiple datasets. In doing so, the developed pipeline has proven to be able to create synthetic images through both domain randomisation and guided domain randomisation for training an object detection model as YOLOv8.

The developed pipeline is comprised of two main parts: the render pipeline and the augmentation/filtering pipeline. The first one is in charge of creating synthetic data based on CAD models while the second one is used for augmenting this data or filtering it. After several tests, it has been concluded that the augmentation of synthetic data does give better results up to two times the original set. While the filtering of generated data works better with small amounts of images. It should be mentioned, that rendering data is way faster to obtain than augmented data. Running SDXL on the computer used for testing, the times were up to 9 times slower than just rendering more frames, indicating that it may be better to just generate many more images through the render pipeline and filter them if needed for reducing the train set and getting faster training.

The render pipeline is a whole project that has been made accessible for everyone to use as a Python module called SynthRender [48] and helped for the release of three papers during my time with the Mercedes-Benz team. It has proven to be robust enough to be used for data generation and successfully allows users to use it without the need for a deeper understanding of Blender.

Overall, the goals for the thesis have been achieved and the results leave space for further development as the future works section already pointed out.

Bibliography

[1] N. Baumgart, M. Lange-Hegermann, and M. Mücke, Investigation of the impact of synthetic training data in the industrial application of terminal strip object detection, 2024. arXiv: 2403.04809 [cs.CV].
URL: https://arxiv.org/abs/2403.04809.

[2] 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.

[3] 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.

[4] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black, A naturalistic open source movie for optical flow evaluation, in *Computer Vision – ECCV 2012*, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 611–625, ISBN: 978-3-642-33783-3.

[5] A. Handa, V. Patraucean, V. Badrinarayanan, S. Stent, and R. Cipolla, Scenenet: Understanding real world indoor scenes with synthetic data, 2015. arXiv: 1511.07041 [cs.CV].
URL: https://arxiv.org/abs/1511.07041.

[6] 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, 2017. arXiv: 1703.06907 [cs.RO].
URL: https://arxiv.org/abs/1703.06907.

[7] J. M. Araya-Martinez, T. Tushar, S. Sardari, et al., Domain adaptation using vision transformers and xai for fully synthetic industrial training, in *Procedia CIRP*, To appear (in press), 2025.

[8] A. Sanchis Reig, Optimizing synthetic data pipelines and explainable ai for industrial applications, 2025.
URL: https://kbdk-aub.primo.exlibrisgroup.com/permalink/45KBDK_AUB/a7me0f/alma99219577889057

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

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

[11] A. Wang, Y. Sun, A. Kortylewski, and A. L. Yuille, Robust object detection under occlusion with context-aware compositionalnets, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 12 645–12 654.

[12] J. Walsh, N. O' Mahony, S. Campbell, et al., Advances in Computer Vision: Proceedings of the 2019 Computer Vision Conference (CVC), Volume 1. Springer International Publishing, 2020,

ISBN: 9783030177959. DOI: 10.1007/978-3-030-17795-9.
URL: http://dx.doi.org/10.1007/978-3-030-17795-9.

[13] J. Josifovski, M. Kerzel, C. Pregizer, L. Posniak, and S. Wermter, Object detection and pose estimation based on convolutional neural networks trained with synthetic data, Oct. 2018. DOI: 10.1109/IROS.2018.8594379.

[14] S. Ren, K. He, R. Girshick, and J. Sun, Faster r-cnn: Towards real-time object detection with region proposal networks, *Advances in neural information processing systems*, vol. 28, 2015.

[15] R. Girshick, Fast r-cnn, 2015. arXiv: 1504.08083 [cs.CV].
URL: https://arxiv.org/abs/1504.08083.

[16] W. Liu, D. Anguelov, D. Erhan, et al., Ssd: Single shot multibox detector, in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, Springer, 2016, pp. 21–37.

[17] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, You only look once: Unified, real-time object detection, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[18] R. Sapkota, R. H. Cheppally, A. Sharda, and M. Karkee, Rf-detr object detection vs yolov12: A study of transformer-based and cnn-based architectures for single-class and multi-class greenfruit detection in complex orchard environments under label ambiguity, *arXiv preprint arXiv:2504.13099*, 2025.

[19] P. Robicheaux, J. Gallagher, J. Nelson, and I. Robinson, Rf-detr: A sota real-time object detection model, 2025.
URL: https://blog.roboflow.com/rf-detr/, (Retrieved: 10-05-2025).

[20] W. A. Shobaki and M. Milanova, A comparative study of yolo, ssd, faster r-cnn, and more for optimized eye-gaze writing, *Sci*, vol. 7, no. 2, 2025, ISSN: 2413-4155. DOI: 10.3390/sci7020047.
URL: https://www.mdpi.com/2413-4155/7/2/47.

[21] J. Hui, Map (mean average precision) for object detection, 2018.
URL: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173, (Retrieved: 08-05-2025).

[22] H. Zanini, Custom object detection in the browser using tensorflow.js, 2021.
URL: https://blog.tensorflow.org/2021/01/custom-object-detection-in-browser.html, (Retrieved: 08-05-2025).

[23] X. Chen, H. Fang, T.-Y. Lin, et al., Microsoft coco captions: Data collection and evaluation server, 2015. arXiv: 1504.00325 [cs.CV].
URL: https://arxiv.org/abs/1504.00325.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] L. Weng, Domain randomization for sim2real transfer, *lilianweng.github.io*, 2019.
URL: https://lilianweng.github.io/posts/2019-05-05-domain-randomization/.

[29] 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.

[30] M. Denninger, D. Winkelbauer, M. Sundermeyer, et al., Blenderproc2: A procedural pipeline for photorealistic rendering, *Journal of Open Source Software*, vol. 8, no. 82, p. 4901, 2023. DOI: 10.21105/joss.04901.
URL: https://doi.org/10.21105/joss.04901.

[31] The blender foundation: Blender 4.4 python api documentation, 2025.
URL: https://docs.blender.org/api/current/index.html, (Retrieved: 17-05-2025).

[32] R. Singh, J. Liu, K. V. Wyk, et al., Synthetica: Large scale synthetic data for robot perception, 2024. arXiv: 2410.21153 [cs.CV].
URL: https://arxiv.org/abs/2410.21153.

[33] N. Corporation, Isaac sim simulator documentation.
URL: https://docs.omniverse.nvidia.com/isaacsim/latest/index.html, (Retrieved: 12-05-2025).

[34] J. Ho, A. Jain, and P. Abbeel, Denoising diffusion probabilistic models, in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 6840–6851.
URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf.

[35] D. Podell, Z. English, K. Lacey, et al., Sdxl: Improving latent diffusion models for high-resolution image synthesis, 2023. arXiv: 2307.01952 [cs.CV].
URL: https://arxiv.org/abs/2307.01952.

[36] L. Zhang, A. Rao, and M. Agrawala, Adding conditional control to text-to-image diffusion models, 2023. arXiv: 2302.05543 [cs.CV].
URL: https://arxiv.org/abs/2302.05543.

[37] H. Ye, J. Zhang, S. Liu, X. Han, and W. Yang, Ip-adapter: Text compatible image prompt adapter for text-to-image diffusion models, 2023. arXiv: 2308.06721 [cs.CV].
URL: https://arxiv.org/abs/2308.06721.

[38] J. M. Araya-Martinez, S. Sardari, M. Lambert, et al., A data-centric evaluation of leading multi-class object detection algorithms using synthetic industrial data, in *Proceedings of the 3rd Stuttgart Conference on Automotive Production (SCAP 2024)*, To appear (in press), Stuttgart, Germany, 2024.

[39] C. Mayershofer, T. Ge, and J. Fottner, Towards fully-synthetic training for industrial applications, in *10th International Conference on Logistics, Informatics and Service Sciences (LISS)*, 2020. DOI: noDOIavailable.

[40] The blender foundation: About blender.
URL: https://www.blender.org/about/, (Retrieved: 17-05-2025).

[41] 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.

[42] The blender foundation: The cycles render engine, 2025.
URL: https://docs.blender.org/manual/en/latest/render/cycles/introduction.html, (Retrieved: 17-05-2025).

[43] 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.

[44] T. Hodan, P. Haluza, S. Obdrzalek, J. Matas, M. Lourakis, and X. Zabulis, T-less: An rgb-d dataset for 6d pose estimation of texture-less objects, 2017. arXiv: 1701.05498 [cs.CV].
URL: https://arxiv.org/abs/1701.05498.

[45] OpenCV, Canny edge detector.
URL: https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html, (Retrieved: 17-05-2025).

[46] The blender foundation: Introduction to keyframes, 2025.
URL: https://docs.blender.org/manual/en/latest/animation/keyframes/introduction.html, (Retrieved: 17-05-2025).

[47] R. W. (W. Hamming, Coding and information theory / Richard W. Hamming. eng, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1986, ISBN: 0131390724.

[48] A. S. Reig, Synthrender, 2025.
URL: https://github.com/Adriagent/Thesis_SyntheticPipeline/tree/main/modules/render_pipeline.

[49] DLR, Blenderproc manual: Semantic segmentation.
URL: https://dlr-rm.github.io/BlenderProc/examples/basics/semantic_segmentation/README.html, (Retrieved: 20-05-2025).

[50] The blender foundation: Sampling, 2025.
URL: https://docs.blender.org/manual/en/latest/render/cycles/render_settings/sampling.html, (Retrieved: 20-05-2025).

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

[52] Huggingface: Controlnet with stable diffusion xl.
URL: https://huggingface.co/docs/diffusers/api/pipelines/controlnet_sdxl, (Retrieved: 27-05-2025).

[53] J. M. Araya-Martinez, A. S. reig, G. Mohan, S. Sardari, J. Lambrecht, and J. Krüger, Synthetic industrial object detection: Genai vs. feature-based methods, in *19th CIRP Conference on Intelligent Computation in Manufacturing Engineering*, To appear (in press), Stuttgart, Germany, 2025.

[54] J. Buchner, Imagehash.
URL: https://github.com/JohannesBuchner/imagehash?tab=readme-ov-file, (Retrieved: 14-05-2025).

[55] The blender foundation: Mix shader, 2025.
URL: https://docs.blender.org/manual/en/latest/render/shader_nodes/shader/mix.html, (Retrieved: 19-05-2025).

Appendices

Appendix A

Randomisation parameters

Input parameters	
Variable	**Range**
Object-to-camera translation $f(t_x)$	$[0\text{m}, 1.6\text{m}]$
Object-to-camera translation $f(t_y)$	$[0\text{m}, 1.2\text{m}]$
Object-to-camera translation $f(t_z)$	$[0.75\text{m}, 4.15\text{m}]$
Object-to-camera rotation $f(\alpha, \beta, \gamma)$	$\{(\alpha, \beta, \gamma) \mid [-180°, 180°] \in \mathbb{R}^3\}$
In-frame object percentage	$[0\%, 100\%]$
Amount of distractors	$[0, 10] \in \mathbb{N}$ obstacles per frame
Environment lighting	$[0, 100]$ W/m^2
Area lighting	$[0, 300]$ W
Background textures	$[0, 10] \in \mathbb{N}$ moving planes
Blur / depth of field	F-Stop variation $[1.0, 10.0]$
Amount of images	1000

Table A.1: List of accepted input variables in the old rendering engine [38].

Material Properties (once every 20 frames)		
Parameter	**Distribution**	**Range**
Albedo Desaturation	Uniform	(0.0, 0.4)
Albedo Add	Uniform	(-0.03, 0.5)
Albedo Brightness	Uniform	(3.0, 4.0)
Diffuse Tint	Uniform	((0.2,0.2,0.2), (1,1,1))
Reflection Roughness Constant	Uniform	(0.5, 0.7)
Metallic Constant	Uniform	(0.5, 0.55)
Specular Level	Uniform	(0.0, 1.0)
Emissive Color	Uniform	((0.0,0.0,0.0), (0.3,0.3,0.3))
Post-Processing (once per frame)		
Enable TV Noise	Bernoulli	(True: 0.1, False: 0.9)
Enable Scan Lines	Bernoulli	(True: 0.1, False: 0.9)
Scan Line Spread	Uniform	(0.1, 0.2)
Enable Vertical Lines	Bernoulli	(True: 0.1, False: 0.9)
Enable Random Splotches	Bernoulli	(True: 0.1, False: 0.9)
Enable Film Grain	Bernoulli	(True: 0.1, False: 0.9)
Grain Amount	Uniform	(0.0, 0.1)
Grain Size	Uniform	(0.7, 1.0)
Color Amount	Uniform	(0.0, 0.15)
Enable Vignetting	Bernoulli	(True: 0.1, False: 0.9)
Lighting		
Ambient Light Intensity (every frame)	Uniform	(0.1, 0.5)
HDRI Background (every 2K frames)	Uniform	50 HDRI backgrounds
Configuration		
Object Height	Uniform	(1.0, 5.0)
Camera Position	Uniform	on sphere of radius 1.0

Table A.2: Randomisation Parameters used in Synthetica [32]

Appendix B

Examples of configuration files

Snippet B.0.1: Example of config_template.yaml file for configuring the rendering.

```
default_scene:      ""
output_dir:         "" # "/media/vrt/shared/Synthetic_Data/output_Framework/test"
train_models_dir:   "" # "/media/vrt/shared/Synthetic_Data/assets/train/"
distractors_dir:    "" # "/media/vrt/shared/Synthetic_Data/assets/distractors/"
backgrounds_dir:    "" # "/media/vrt/shared/Synthetic_Data/assets/backgrounds/"
planes_dir:         "" #
 ↪  "/media/vrt/shared/Synthetic_Data/assets/materials/plane_materials"

seed: 0                                                 # If seed is set to -1, it
 ↪  will be a random seed
cycles_samples: 100                                     # By default it is set to
 ↪  1024
render_options: ["segmasks", "depth", "normals"]        # Possible render options:
 ↪  [segmasks, depth, normals]
save_hdf5: True                                         # If enabled, saves the
 ↪  rendered data as an hdf5 file (Needed for coco and yolo annotations)
save_raw: False                                         # If enabled, saves the
 ↪  rendered raw data separated in (rgb/, depth/, normals/, segmasks/)

coco_to_yolo_category_mapping: {"1": 1, "2": 2, "3": 0}

# Dataload white lists:
train_models_whitelist:         []
distraction_models_whitelist:   []
backgrounds_whitelist:          ["machine_shop_02_4k.exr", "leadenhall_market_4k.exr",
 ↪  "warm_restaurant_night_4k.exr", "sylvan_highlands.exr"]
planes_whitelist:               []
```

```yaml
# Dataload black lists:
train_models_blacklist:         ["washer.ply"]
distraction_models_blacklist:   ["Dining_table_2.blend"]
backgrounds_blacklist:          []
planes_blacklist:               []

physics:
    simulate_physics: False             # If enabled, simulates the physics.
    simulate_actives: True              # Sets train models as rigidbody active (can
    ↪  move and can collide with other rigidbodies).
    simulate_passives: True             # Sets the distractors as rigidbody passive
    ↪  (fixed but can collide with other rigidbodies).
    reorientate_camera: True            # Recalculates the orientation of the camera to
    ↪  look towards the active objects.
    default_scene_as_passives: True     # Enables pasive physics for the loaded models
    ↪  of the default scene.
    create_ground_plane: True           # Create invisible ground so objects can fall on
    ↪  it.
    max_simulation_time: 10             # Simulation is accelerated: 1 simulation second
    ↪  != 1 real second (this depends on how slow is the simulation)
    check_interval: 2.5                 # Interval at which check if objects stopped
    ↪  moving (In simulation seconds).
    stopped_location_threshold: 0.5     # Minimum movement per second to be considered
    ↪  as stopped.
    stopped_rotation_threshold: 2       # Minimum rotation per second to be considered
    ↪  as stopped.

world:
    background_light_strength: [0.01, 0.5]            # with steps of 0.01
    default_background_img: "sylvan_highlands.exr"
    random_backgrounds: True

lights:
    lights_dir: [[0.323, -0.855, 0.405], [0.914, 0.028, 0.405], [-0.895, -0.186, 0.405]]
    light_size: 0.25
    contrasts: [1, 1, 1]            # Energy multiplier for each light.
    distance: 6
    light_intensity: [0, 160]
    randomize_color: True
    exponential_lights: False
    exponential_factor: 2.33

camera:
    resolution: [512, 512]
    f-stop: [1, 16]                     # A higher fstop value makes the resulting image
    ↪  look sharper, while a low value decreases the sharpness.
```

```
    sensor_size: 36                     # Sensor size in milimiters.
    intrinsic_parameters_path: ""       # Path to a YAML file containing the camera
     ↪ intrinsic parameters matrix (k)
    pos_shell_radius: [2, 3]
    pos_shell_elevation: [10, 90]       # Angle in degrees (min and max)

empty_object:
    center: [0, 0, 0.5]
    pos_shell_radius: [0, 1.5]
    pos_shell_elevation: [0, 90]

models:
    trains:
        sample_size: [-1, -1]               # If value is set to -1, it will be the
         ↪ maximum
        scale: 1
        n_copies: 0                         # Creates copies of loaded models. If the
         ↪ model has a custom n_copies value it will use the custom one.
        dynamic_origin: True                # Uses the emtpy object as origin.
        static_origin: [0, 0, 0.5]          # If "dynamic_origin" is False, it will use
         ↪ this location as fixed origin.
        pos_min: [-0.5, -0.5, -0.5]         # Min value for x,y,z using origin as
         ↪ reference frame.
        pos_max: [0.5, 0.5, 0.5]            # Max value of x,y,z using origin as
         ↪ reference frame.
        rot_min: [0,0,0]                    # Angles in radians
        rot_max: [3.1415, 3.1415, 3.1415]   # Angles in radians
        keep_children: True                 # If true, keeps al the mesh models loaded
         ↪ with a common parent. Else, joins all the meshes into a single model.
        combined_parent: True               # If true, sets as parent a merged copy of
         ↪ the children.
        simplify_models: False              # If true, runs convex hull decomposition
         ↪ (on parent if combined_parent enabled or on children otherwise)

    distractors:
        sample_size: [5, -1]                # If value is set to -1, it will be the
         ↪ maximum
        scale: 1
        n_copies: 0                         # Creates copies of loaded train_models. If
         ↪ the model has a custom n_copies value it will use the custom one.
        pos_min: [-1.5, -1.5, 0]
        pos_max: [1.5, 1.5, 0]
        rot_min: [0, 0, 0]                  # Angles in radians
        rot_max: [0, 0, 3.1415]             # Angles in radians
        keep_children: True                 # If true, keeps al the mesh models loaded
         ↪ with a common parent. Else, joins all the meshes into a single model.
```

```
        combined_parent: True               # If true, sets as parent a merged copy of
        ↪ the children.
        simplify_models: False              # If true, runs convex hull decomposition
        ↪ (on parent if combined_parent enabled or on children otherwise)

fake_models:
    trains:
        sample_size: [-1, -1]               # If value is set to -1, it will be the
        ↪ maximum
        scale: 1
        n_copies: 0
        simple_trains: 3                    # If bigger than 0, creates simple
        ↪ distractors that will replace train models.
        similar_trains: 0                   # If bigger than 0, creates n deformed
        ↪ copies of each loaded train model.

    distractors:
        sample_size: [-1, -1]
        scale: 1
        n_copies: 0
        simple_distractors: 20              # If bigger than 0, creates distractors with
        ↪ simple geometric shapes.

plane:
    simple_planes: 10
    x_length: 6
    y_length: 6

custom_models:
    example.obj:
        scale: 0.001            # Scale factor for the model.
        n_copies: 2             # Number of copies of the model.
        keep_children: True     # If true, keeps al the mesh models loaded with a common
        ↪ parent. Else, joins all the meshes into a single model.
        combined_parent: True   # If true, sets as parent a joined copy of the children.
        simplify_models: False   # Applies convex hull decomposition on the model.

        children_whitelist: []  # Childrens to load. (If empty, loads all of them)
        children_blacklist: []  # Childrens to discard.
        segment_whitelist:  []  # Childrens from which generate annotations. (If empty,
        ↪ loads all of them)
        segment_blacklist:  []  # Childrens from which not generate annotations.
```

Snippet B.0.2 Example of the config.json file for configuring the rendering used in the boxes use case.

```
"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]
        }
    }
}
```

Appendix C

Examples of python codes.

Snippet C.0.1: Code for setting up the Blender simulation.

```
def set_up_scene(self):
    assert self.config is not None, ValueError(f"Ensure that a configuration has been
    ↪  loaded before!")

    bproc.init()

    # Setting up camera:
    scene_loader.camera_setup(self.config)

    # Load a default blender scene:
    self.default_scene = scene_loader.load_default_scene(self.config)

    # Create empty object and assigning categories to models.
    self.empty = bproc.object.create_empty("empty", "arrows")

    # Loading world backgrounds:
    self.backgrounds = scene_loader.load_backgrounds(self.config)

    # Load train models:
    self.train_models, self.fake_train_models =
    ↪  scene_loader.load_train_models(self.config)

    # Load distractors:
    self.distractors, self.fake_distractors =
    ↪  scene_loader.load_distractor_models(self.config)

    # Load planes and their materials:
    self.planes = scene_loader.load_plane_materials(self.config)
```

```
    # Configure the lights:
    self.lights = scene_loader.load_lights(self.config, self.empty)

    # Setting up rigid bodies for the first time for all the models:
    if self.config["physics"]["simulate_physics"]:
        print("Setting models as active rigidbodies...")
        scene_setter.setup_models_physics(self.config, [*self.train_models,
        ↪  *self.fake_train_models], as_active=True, enable=False,
        ↪  default_config=self.config['models']['trains'])
        scene_setter.setup_models_physics(self.config, [*self.distractors,
        ↪  *self.fake_distractors], as_active=False, enable=False,
        ↪  default_config=self.config['models']['distractors'])

    # Create copies for train models:
    bproc_utils.create_duplicates(self.train_models, self.config,
    ↪  self.config["models"]["trains"])

    # Create copies for fake train models:
    bproc_utils.create_duplicates(self.fake_train_models, self.config,
    ↪  self.config["fake_models"]["trains"])

    # Create copies for distractors models:
    bproc_utils.create_duplicates(self.distractors, self.config,
    ↪  self.config["models"]["distractors"])

    # Create copies for fake distractors models:
    bproc_utils.create_duplicates(self.fake_distractors, self.config,
    ↪  self.config["fake_models"]["distractors"])


    return self.empty, self.train_models, self.distractors, self.lights, self.planes,
    ↪  self.fake_train_models, self.fake_distractors, self.default_scene
```

Snippet C.0.2: Code used for loading the train models into the simulation

```
def load_train_models(config:dict[str, dict]):
    """
    Loads the train models specified in the config file and sets a category_id to all of
    ↪  them so that segmentation masks can be obtained.

    It can also create fake train models which will spawn as if they were train objects
    ↪  but with no category_id:
```

```
        - Creates deformed copies of the train models if 'fake_similar_trains' is
        ↪  enabled.
        - Creates simple geometric shapes if 'fake_simple_trains' is enabled.

    Parameters:
        config (dict): Configuration dictionary containing the train settings.
    Returns:
        tuple: A tuple of train models (loaded, fakes)
            - train_models (list[MeshObject]): A list of objects representing the loaded
            ↪  train models.
            - fake_models (list[MeshObject]): A list of objects representing the fake
            ↪  train models.
    """

    train_models:list[MeshObject] = []
    fake_models:list[MeshObject] = []

    # Creating model collection:
    collection = bpy.data.collections.new("Train_Models")
    bpy.context.scene.collection.children.link(collection)  # Link the collection to the
    ↪  current scene

    if os.path.isdir(dir_path:=config["train_models_dir"]):
        pos_callback = lambda x, i: (range(-len(x)//2+1, len(x)//2+1, 1)[i], -5, 0) #
        ↪  Setting up initial position of models
        whitelist = config["train_models_whitelist"]
        blacklist = config["train_models_blacklist"]

        models_config = {"default_config": config["models"]["trains"]}
        models_config.update(config.get("custom_models", {}))

        train_models = bproc_utils.load_models_folder(dir_path, whitelist, blacklist,
        ↪  pos_callback, collection, models_config)

        # Setting category_id to train objects.
        bproc_utils.set_category_to_meshes(train_models, models_config)

    elif dir_path:
        print(f"Warning: Train folder not found!: {dir_path}")

    # Creates fake training models using deformed versions of the training models.
    if n_similar:=config['fake_models']["trains"].get("similar_trains"):
        pos_callback = lambda x, i: (range(-len(x)//2+1, len(x)//2+1, 1)[i], -6, 0) #
        ↪  Setting up initial position of models
        total_copies = n_similar * len(train_models)
        count = itertools.count()
```

```
        # Create n_similar copies of each train_model.
        for model in train_models:

            model_name = model.get_name()

            model_config = {}
            model_config.update(config["fake_models"]["trains"])
            model_config.update(config['custom_models'].get(model_name, {}))

            for _ in range(n_similar):
                pos = pos_callback(range(total_copies), next(count))
                similar_dis = bproc_utils.create_similar_distractor(pos, model,
                ↪  collection, model_config)
                fake_models.append(similar_dis)

    # Creates fake training models using simple geometric shapes.
    if n_fakes:=config['fake_models']["trains"].get("simple_trains"):
        pos_callback = lambda x, i: (range(-len(x)//2+1, len(x)//2+1, 1)[i], -7, 0) #
        ↪  Setting up initial position of models

        models_config = {"default_config": config["fake_models"]["trains"]}
        models_config.update(config.get("custom_models", {}))

        for i in range(n_fakes):
            pos = pos_callback(range(n_fakes), i)
            simple_model = bproc_utils.create_random_distractor(pos, collection,
            ↪  models_config)
            fake_models.append(simple_model)


    return train_models, fake_models
```

Snippet C.0.3: Code used for loading a model into the simulation

```
def load_model(model_path:str, collection=None, model_config:dict = None):
    """
    Loads the model from a specific model_path. and adds the model to a collection if
    ↪  specified.

    It can also set the following configuration if a dict is passed with the following
    ↪  keys:
```

```
    - 'scale' : 1.0 - Scale factor of the loaded model.
    - 'keep_children' : True - Whether to keep the children meshes or combine them
    ↪  into a single mesh.
    - 'combined_parent' : True - Whether to have a combined mesh as a parent.
    - 'children_whitelist' : [] - Children meshes to be accepted.
    - 'children_blacklist' : [] - Children meshes to be removed.

Parameters:
    model_path (str): Path to the model (.blend, .ply, .fbx, .obj, .glb).
    collection=None: Colleciton object in which include the loaded model.
    model_config (dict) = None: Model configuration.
Returns:
    loaded_model: A reference to the parent of the loaded model
"""

# Loading model depending on its extension:
extension = os.path.splitext(os.path.basename(model_path))[-1]

if extension == ".blend":
    models = bproc.loader.load_blend(model_path, obj_types=["mesh", "empty"])
elif extension in (".ply", ".fbx", ".obj", ".glb"):
    models = bproc.loader.load_obj(model_path)
else:
    return None

# Default custom model settings
model_config    = model_config or {}
scale           = model_config.get("scale", 1.0)
keep_children   = model_config.get("keep_children", True)
combined_parent = model_config.get("combined_parent", True)
whitelist       = set(model_config.get("children_whitelist", []) or
↪  [model.get_name() for model in models])
blacklist       = set(model_config.get("children_blacklist", []))


# Filter models based on whitelist and blacklist
valid_model_names   = (set([model.get_name() for model in models]) & whitelist) -
↪  blacklist
objects_to_delete   = [model for model in models if model.get_name() not in
↪  valid_model_names]
models              = [model for model in models if model.get_name() in
↪  valid_model_names]

# Process each model: adjust scale, get meshes, unparent them and update collection
↪  links if neccesary.
```

```
    meshes:list[MeshObject] = []
    to_delete:list[Entity] = []
    for model in models:
        # model.set_scale(model.get_scale()*scale)
        if model.get_attr("type") == "MESH":
            model.clear_parent()
            model.persist_transformation_into_mesh(location=False, rotation=False,
             ↪  scale=True)
            meshes.append(model)

            if collection:
                bpy.context.collection.objects.unlink(model.blender_obj)
                collection.objects.link(model.blender_obj)

        else:
            to_delete.append(model)


    # Now we handle parenting depending on whether we are joining the objects together
     ↪  or not.
    parent = meshes[0]

    if len(meshes) > 1:
        if keep_children:
            if combined_parent:
                parent = create_combined_parent(meshes)
            else:
                parent = bproc.object.create_with_empty_mesh("new_parent")
                # parent = bproc.object.merge_objects(meshes)
                parent.set_cp("combined_mesh", False)

            center = get_children_center(meshes)
            parent.set_location(center)

            if collection:
                bpy.context.collection.objects.unlink(parent.blender_obj)
                collection.objects.link(parent.blender_obj)

            # Setting common parent for all the meshes.
            for mesh in meshes:
                mesh.set_origin(center)
                mesh.set_parent(parent)

        else:
            parent.join_with_other_objects(meshes[1:])
```

```
    # Now we set the name of the parent to be the same as the file for easier
    ↪ clasification:
    if parent.get_attr("type") == "MESH":
        parent.set_origin(mode="CENTER_OF_MASS")

    parent.set_name(os.path.basename(model_path))

    bproc.object.delete_multiple(to_delete)
    bproc.object.delete_multiple(objects_to_delete)
    parent.set_scale(parent.get_scale()*scale)

    # parent.blender_obj.show_name = True

    return parent
```

Snippet C.0.4: Part of the code that randomises and keyframes the scenes.

```
for frame in tqdm(range(0, num_keyframes), desc="Preparing keyframes", unit=" keyframe",
↪ disable=verbose):

    # Generate random setup for the scene:
    back_strength                    = scene_randomizer.randomize_backg(self.config)
    plane                            = scene_randomizer.randomize_plane(self.planes)
    empty_pos, empty_rot             = scene_randomizer.randomize_empty(self.config)
    cam_pose, dof                    = scene_randomizer.randomize_camera(self.config,
    ↪ empty_pos)
    lights_energy, color_rgb         = scene_randomizer.randomize_lights(self.config,
    ↪ self.lights, num_keyframes, frame)
    set_train_models, set_train_poses =
    ↪ scene_randomizer.randomize_train_numba(self.config, self.train_models,
    ↪ empty_pos, self.fake_train_models, cam_pose)
    set_distr_models, set_distr_poses =
    ↪ scene_randomizer.randomize_distr_numba(self.config, self.distractors,
    ↪ self.fake_distractors, list(zip(set_train_models, set_train_poses)))

    if start_frame <= frame <= stop_frame:
        # Set background light strength
        bpy.context.scene.world.node_tree.nodes["Background"].inputs["Strength"].defaul
        ↪ t_value =
        ↪ back_strength
        scene_setter.set_plane(plane) # Set plane values
        scene_setter.set_empty(self.empty, empty_pos, empty_rot)  # Set location of
        ↪ empty object used as origin for train models.
```

```
        scene_setter.set_lights(self.config, self.lights, lights_energy, color_rgb) #
        ↪ Set lights values
        scene_setter.set_models_pose(set_train_models, set_train_poses) # Set train
        ↪ models values
        scene_setter.set_models_pose(set_distr_models, set_distr_poses) # Set
        ↪ distractors models values
        bproc.camera.add_camera_pose(cam_pose, frame=frame) # Set camera pose
        bproc.camera.add_depth_of_field(focal_point_obj=self.empty, fstop_value=dof) #
        ↪ Set camera f-stop

        # Keyframe scene:
        bproc_utils.save_keyframe(frame, items_children)

        # Recalculate train objects position with physic simulation.
        scene_setter.do_physics(self.config, set_train_models, set_distr_models, frame)

        to_hide_models = [*set_train_models, *set_distr_models, plane]  # Update models
        ↪ to be hidden to the current selected ones.
        bproc_utils.hide_render_view(to_hide_models)                    # Hide only
        ↪ models shown in previous iteration.
```