

Bachelor's Thesis

Computer vision methods for marker-less
identification of 3D printed components

4213600: Bachelor's thesis

Chirag Anand Reddy

Bachelor's Thesis

Computer vision methods for marker-less identification of 3D printed components

by

Chirag Anand Reddy

Student Name	Student Number
Chirag Reddy	4221020

Fraunhofer Supervisor: Daniel Müller
THWS Supervisor: Rainer Herrler
Project Duration: 01/08/24 - 31/12/24
Faculty: Faculty of Electrical Engineering, THWS

Cover: 3D printer and finished 3D printed components
Style: TU Delft Report Style, with modifications by Daan Zwaneveld



Contents

1	Introduction	1
1.1	Context and Motivation	1
1.2	Problem Statement	2
1.3	Research Goals	2
1.4	Thesis Structure	2
2	Related work	3
2.1	Generating Datasets from 3D CAD Models for Object Detection[8]	3
2.1.1	Methodology	3
2.1.2	Results	4
2.2	A Fully Automated Method for Object Detection Using CAD Models[14]	4
2.2.1	Methodology	4
2.2.2	Results	5
2.3	H3DNet: 3D Object Detection Using Hybrid Geometric Primitives[17]	5
2.3.1	Methodology	5
2.3.2	Results	6
3	Basics	7
3.1	Pipeline	7
3.2	Neural Networks	8
3.2.1	Key Concepts	8
3.2.2	Types of Neural Networks	8
3.2.3	How Neural Networks Work	9
3.2.4	Applications in the Thesis	9
3.3	YOLO	9
3.3.1	Key Features of YOLO	9
3.3.2	YOLOv11-OBB	9
3.3.3	Applications in the Thesis	10
3.4	Blender	10
3.4.1	Key Features of Blender	11
3.4.2	Applications in the Thesis	11
3.5	BlenderProc	11
3.5.1	Key Features of BlenderProc	11
3.5.2	Applications in the Thesis	11
3.6	Style transfer	12
3.6.1	Canny Edge Detection	12
3.6.2	Active Canny Edge Detection	13
3.6.3	Holistically-Nested Edge Detection (HED)	14
3.6.4	Informative Drawings	15
4	Design	16
4.1	3D Rendering Software	16
4.2	Rendering Environment	16
4.3	API for Automation	17
4.4	Style Transfer Algorithms	18

5	Implementation	19
5.1	Image generation	19
5.2	Dataset Preparation	21
5.3	Model training	22
6	Evaluation	24
6.1	Validation Metrics	24
6.2	Inference time	26
6.3	Confidence	27
6.4	Weighted Confidence	29
7	Conclusion	32
	References	35

List of Figures

3.1	Pipeline Overview	7
3.2	Basic illustration of a Neural Network.	9
3.3	Yolo: Default vs OBB	10
3.4	Example style transfers	12
3.5	Color vs Canny	13
3.6	Color vs Active Canny	14
3.7	Color vs HED layers	15
3.8	Colored image	15
3.9	Anime style	15
3.10	Contour style	15
3.11	Opensketch style	15
3.12	Color vs Line drawings	15
4.1	Blender scene for rendering	17
5.1	Pipeline: Image Generation	19
5.2	Pipeline: Image Generation Control	20
5.3	Generated images	21
5.4	Pipeline: Dataset Preperation	21
6.1	Prediction: Control Image	30
6.2	Prediction: Canny Image	30
6.3	Prediction: Active Canny Image	30
6.4	Prediction: HED1 Image	30
6.5	Prediction: HED2 Image	30
6.6	Prediction: Anime Style Image	30
6.6	Prediction: Contour Style Image	31
6.7	Prediction: Opensketch Style Image	31
7.1	Prediction: New colors	33

List of Tables

5.1	Performance Metrics of YOLOv11-OBB Models	23
6.1	Small Model Metrics	25
6.2	Nano Model Metrics	25
6.3	Medium Model Metrics	25
6.4	Large Model Metrics	25
6.5	Extra-Large Model Metrics	26
6.6	NVIDIA A100 vs GTX 1650	26
6.7	Inference Time: Nano model(ms)	27
6.8	Inference Time: Small model(ms)	27
6.9	Inference Time: Medium model(ms)	27
6.10	Inference Time: Large model(ms)	27
6.11	Inference Time: Extra Large model(ms)	27
6.12	Average Confidence: Nano size	28
6.13	Average Confidence: Small size	28
6.14	Average Confidence: Medium size	28
6.15	Average Confidence: Large size	28
6.16	Average Confidence: Extra-Large size	28
6.17	Weighted Confidence: Small Model	29
6.18	Weighted Confidence: Nano Model	29
6.19	Weighted Confidence: Medium Model	29
6.20	Weighted Confidence: Large Model	29
6.21	Weighted Confidence: Extra Large Model	29

Introduction

The integration of computer vision into manufacturing has revolutionized quality control, automation, and traceability, particularly in additive manufacturing (3D printing). As industries increasingly rely on 3D printing for rapid prototyping and customized production, the ability to accurately identify and track printed components is becoming crucial. However, developing robust object detection systems for 3D-printed parts presents several challenges, including dataset scarcity, domain adaptation issues, and the persistent gap between simulated and real-world data. This thesis addresses these challenges by developing a marker-less, automated pipeline for detecting 3D-printed components, leveraging synthetic data, advanced computer vision techniques, and style transfer methods to enhance model generalization and bridge the domain gap.

1.1. Context and Motivation

3D printing enables the fabrication of highly customized and complex geometries, directly from digital CAD models. However, post-production tasks such as inventory management, quality assurance, and robotic manipulation require reliable object detection systems. The primary challenge in developing such systems is the lack of large, annotated datasets for 3D-printed components. Unlike widely used object detection benchmarks, 3D-printed objects often vary significantly in shape, texture, and material properties, making it difficult to collect and annotate real-world training data at scale.

A common approach to overcoming dataset limitations is synthetic data generation. By rendering images of CAD models in a simulated environment, large-scale datasets can be produced without extensive manual effort. However, the transition from synthetic to real-world applications is non-trivial. Differences in lighting conditions, material reflectivity, environmental noise, and camera properties introduce a domain gap, often leading to poor model generalization when deployed in real-world settings. This sim-to-real gap remains a critical bottleneck in using synthetic datasets effectively.

Another challenge lies in the robustness of object detection models across varying conditions. In real-world applications, printed components may appear in different colors, under diverse lighting, and against cluttered backgrounds. Ensuring that the trained model is invariant to these changes is essential for practical deployment. The

use of style transfer techniques provides a potential solution by transforming synthetic images to better resemble real-world data, thereby improving generalization and reducing the domain gap.

1.2. Problem Statement

The core problem addressed in this thesis is the development of a fully automated pipeline that transforms CAD models into trained object detection models without manual intervention. Key challenges include:

- **Dataset Generation:** Collecting and annotating real-world images is expensive and impractical for a diverse range of 3D-printed components. While synthetic data provides an alternative, it often lacks realism and diversity.
- **Sim-to-Real Discrepancy:** Models trained on synthetic datasets frequently struggle to generalize due to discrepancies in lighting, material properties, and texture variations. Minimizing this gap is essential for real-world deployment.
- **Detection Robustness:** Object detection models need to be invariant to color, lighting variations, and environmental conditions to function reliably in industrial settings.
- **Computational Efficiency:** High-accuracy 3D object detection methods, such as point cloud-based approaches, are computationally expensive and challenging to deploy in real-time applications.

1.3. Research Goals

This thesis aims to:

1. Develop a modular pipeline for generating synthetic datasets, annotating images, and training object detection models directly from CAD files.
2. Investigate the impact of style transfer techniques (e.g., edge detection, neural style filters) in reducing the sim-to-real gap and improving model robustness.
3. Compare the performance of the computer vision models with to optimize the balance between detection accuracy and inference speed.

1.4. Thesis Structure

The remainder of this document is organized as follows:

- Chapter 2 reviews related work in synthetic dataset generation, 3D object detection, and style transfer techniques.
- Chapter 3 provides foundational concepts, including an overview of neural networks, YOLO-OBb, style transfer algorithms and the role of Blender in synthetic data generation.
- Chapter 4 details the design of the proposed pipeline, including rendering environments, automation APIs, and style transfer integration.
- Chapters 5 and 6 present the implementation and evaluation of the pipeline, covering dataset generation, model training, and performance assessment across different GPU platforms and model configurations.
- Chapter 7 concludes with key findings, limitations, and future research directions.

2

Related work

The papers discussed below aim to address the challenges mentioned in chapter 1 through novel methodologies, each contributing distinctively to the field. These works underscore the importance of leveraging CAD models, synthetic data, and innovative detection architectures to optimize object detection pipelines.

2.1. Generating Datasets from 3D CAD Models for Object Detection[8]

This study aims to reduce the cost and effort of dataset preparation for 3D object detection by using 3D CAD models instead of the traditional method of preparing a dataset from real world images/scans. The focus is on generating synthetic training data and converting CAD models into point clouds for deep learning.

2.1.1. Methodology

- **Dataset Generation:** The process began with the conversion of 3D CAD models into multi-view point clouds. Using Poisson disk sampling, a method known for its ability to produce evenly distributed points, the authors ensured the creation of high-density point clouds that closely represented the geometric details of the objects. This technique provided a robust foundation for synthetic dataset generation, effectively replicating the 3D structure of the objects. In total for each object, 360 views of the CAD models were generated by rotating the object one degree and capturing the point clouds, simulating the perspectives of a real-world depth camera.
- **Training Approach:** To leverage these datasets, the authors employed 3DmFV-Net[1], a convolutional neural network specifically designed to recognize patterns in the point clouds. This model is designed to work specifically with 3D data. To make it better at handling real-world variations, they randomly removed some points (downsampling) and added small amounts of noise to the data. This helped the model learn how to handle messy or incomplete data, like real world 3D point clouds taken from a depth camera.
- **Point Cloud Acquisition:** Validation of the synthetic dataset was conducted by capturing real-world point clouds using an Intel RealSense D435i camera. This camera, equipped with both RGB and depth sensors, provided real-time depth data of physical objects in various configurations. Depth maps were processed

to extract object masks, which were further converted into point clouds.

2.1.2. Results

- The trained model achieved an average accuracy of 88.9% when detecting objects in real-world settings.
- While the model performed well, it encountered challenges with certain types of data. Especially due to use of point clouds. For instance:
 - Reflective or shiny surfaces, common in real-world objects, created issues with depth data capture.
 - The method struggled with intricate object geometries, where fine details were harder to replicate in synthetic data.
 - Real-world object surfaces sometimes had gaps or uneven features that differed from the smoothness of CAD models

2.2. A Fully Automated Method for Object Detection Using CAD Models[14]

In industries such as manufacturing and robotics, the need for precise and reliable object detection is critical. However, creating high-quality datasets tailored to specific objects is often time-consuming and costly, especially when real-world images need to be collected and annotated manually. The research aimed to eliminate these barriers by automating the entire pipeline from CAD models to a trained object detection model, emphasizing industrial applications such as quality control and automated production line monitoring.

2.2.1. Methodology

- **Synthetic Data Generation:** A key aspect of the methodology was leveraging synthetic data to eliminate the need for labor-intensive real-world dataset collection and annotation. Using the 3D modeling software Blender, the authors developed automated scripts to render photorealistic images of CAD models. These scripts introduced variability to ensure that the dataset was diverse and closely mimicked real-world scenarios. These included, randomized backgrounds, lighting conditions, and object textures and materials.
- **Model Training:** The synthetic images and corresponding annotations (such as bounding boxes) were used to train a pre-trained Faster R-CNN[12]. Faster R-CNN was chosen for its balance between detection speed and accuracy, making it a reliable choice for industrial applications. The authors used transfer learning to fine-tune the model, which means the initial layers were initialized with pre-trained weights and only the later layers are trained.
- **Parameter tuning:** To maximize model performance, the authors conducted an extensive study to optimize the parameters used in synthetic data generation and model training such as:
 - Number of images to train on.
 - Lighting and Texture Variability.
 - Model Hyperparameters such as learning rates, batch sizes, and other training parameters were fine-tuned.

2.2.2. Results

- The proposed approach demonstrated strong performance in real-world object detection tasks. By using only synthetic data for training, the model achieved F1-scores exceeding 90%, highlighting its ability to generalize effectively to real-world scenarios.
- The model performed comparably to those trained on real-world datasets, demonstrating that well-designed synthetic images can serve as an effective substitute for real-world data.
- While the pipeline was successful and achieved good automation for dataset creation and good model training results, it had a few shortcomings:
 - **Computational cost:** To generate high quality images, high processing power was required, which is not always available.
 - **Parameter Tuning:** The trial-and-error approach to fine tuning hyper-parameters can be time-consuming and requires a certain level of expertise.

2.3. H3DNet: 3D Object Detection Using Hybrid Geometric Primitives[17]

The study introduces H3DNet, a novel neural network that utilizes hybrid geometric primitives to improve the accuracy of 3D object detection from point clouds. The work addresses limitations in existing methods that struggle with complex object geometries, occlusions, and incomplete data. H3DNet provides a framework that predicts multiple types of geometric primitives, combines them into object proposals, and refines these proposals for robust and accurate 3D object detection.

2.3.1. Methodology

- **Hybrid Geometric Primitives:** H3DNet used three types of geometric features (primitives) to detect objects more accurately:
 - **Bounding Box Centers:** Gave the approximate location of the object.
 - **Bounding Box Face Centers:** Focused on flat surfaces of objects, useful for objects with broad, planar features
 - **Bounding Box Edge Centers:** Captured finer details, which helped detect thin or irregularly shaped objects.

Combining these primitives allowed the model to capture multiple geometric cues, improving detection even in complex or noisy data

- **Object Proposal Generation:** A special distance function was used to create object proposals. This function measured how closely the predicted primitives matched the shape of the object. During training, the network refined these proposals to make them more accurate.
- **Proposal Refinement and Classification:** After generating object proposals, the network classified them to determine which ones contained objects. It also fine-tuned details like the object's size, orientation, and position. To ensure accuracy, the network used information from all the primitives together, combining coarse and detailed features.

2.3.2. Results

- H3DNet achieved good accuracy on two major datasets for 3D object detection:
 - **ScanNet V2**¹: 67.2% mAP (Mean Average Precision), which is 8.5% better than previous methods.
 - **SUN RGB-D**²: 60.1% mAP, a 2.4% improvement over earlier approaches.
- Its use of multiple geometric features (primitives) made the model better at handling noisy or incomplete point cloud data.
- While the above point highlight it's performance on 3D data, especially in noisy and hard to detect objects, it had the following shortcomings:
 - **High Computational Cost**: Processing multiple geometric features and refining proposals made the model slower and required more computing power.
 - **Reliance on Quality Data**: While it could handle some noise, the model still needed reasonably good point cloud data to perform its best.

The papers above prove that significant progress has already been made in the field of synthetic data generation. In this thesis, an attempt to overcome previous challenges and a new method of using styles to reduce the gap between simulation and reality will be made. It builds mostly on this paper[14] which has a lot of the similarities to the methods used in this thesis. It can also be considered as an extension to that paper.

¹<http://www.scan-net.org/>

²<https://rgb-d.cs.princeton.edu/>

3

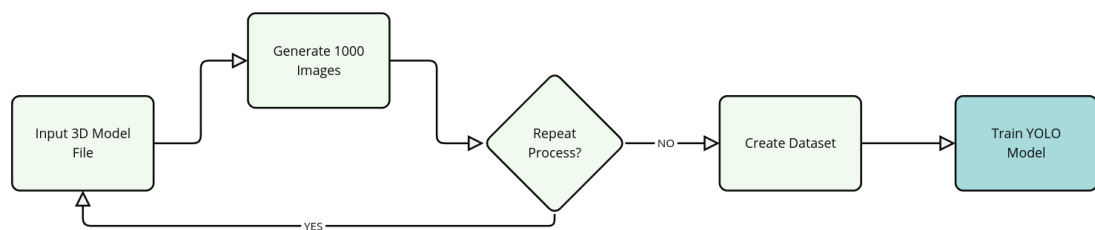
Basics

This chapter serves as the foundation for understanding the core concepts, methodologies, and technologies integral to this thesis. The objective is to provide the necessary context and theoretical background that informs the design, implementation, and evaluation of the proposed approach.

The problem tackled in this thesis revolves around the detection of 3D-printed objects using machine vision. To address this, a pipeline is developed that transforms 3D CAD models into a trained object detection model. This requires knowledge of multiple domains, including computer vision, machine learning, synthetic data generation, and neural networks.

3.1. Pipeline

This is the main section of this chapter, wherein it explains the high level overview of a process which takes in a 3D model and gives out a model capable of detecting that object. Given below is the flowchart diagram of the pipeline: As seen in the image,



miro

Figure 3.1: Pipeline Overview

the process can be explained as follows:

1. First, the user feeds the 3D model file as input

2. Next, 1000 images are generated by the program
3. The above steps can be repeated multiple times for all available objects
4. Next, the dataset is generated by compiling all generated images along with their annotations
5. Finally, a machine vision model is trained on the dataset.

At the end of the pipeline a trained machine vision model is ready to be used to perform object detection tasks in the real world. Further in this chapter more concepts needed for each of the above steps are explained.

3.2. Neural Networks

Neural networks are a key technology in modern AI, especially for computer vision tasks. Inspired by the structure of the human brain, they consist of layers of interconnected nodes (neurons) that process and learn patterns by adjusting the strengths (weights) of their connections [9].

3.2.1. Key Concepts

- **Architecture:** Neural networks have three main types of layers:
 - *Input Layer:* Takes in raw data, such as pixel values from images.
 - *Hidden Layers:* Process the input to extract patterns and features, using functions like ReLU or sigmoid [10].
 - *Output Layer:* Produces the final output, such as object classes or probabilities.
- **Training Process:** Neural networks learn by:
 - Measuring the error between predicted outputs and actual targets.
 - Adjusting weights using optimization methods like gradient descent [13].
- **Evaluation Metrics:** Metrics like accuracy and F1 score assess the performance of the network.

3.2.2. Types of Neural Networks

- **Feedforward Neural Networks (FNN):** Basic networks with one-way data flow, used for tasks like regression and classification [9].
- **Convolutional Neural Networks (CNN):** Specialized for images, CNNs use convolutional filters to extract spatial features for tasks like object detection [7].
- **Variational Autoencoders (VAEs):** Useful for generating synthetic training data by creating diverse variations of CAD models [6].

3.2.3. How Neural Networks Work

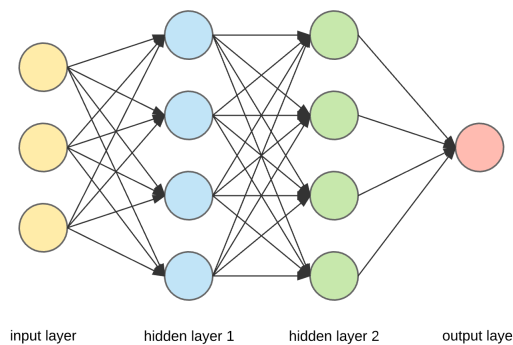


Figure 3.2: Basic illustration of a Neural Network.

As shown in Figure 3.2, neural networks process input data (e.g., image pixel values) through the input layer. The hidden layers initially have random weights and are updated as per a loss. This enables the hidden layers to identify patterns and features, such as edges in an image, and the output layer generates results like classifying detected objects. At the end of training, the model has updated weights which are best fit for the detection of the objects that it has trained for.

3.2.4. Applications in the Thesis

- **Object Detection:** A Convolutional Neural Network is used to detect 3D-printed objects in synthetic and real images.
- **Synthetic Data Augmentation:** Neural networks simulate lighting, textures, and perspectives to create diverse training datasets from CAD models.
- **Real-World Testing:** Trained models are evaluated on real-world images to ensure they generalize well from synthetic data.

3.3. YOLO

YOLO (You Only Look Once) is a powerful real-time object detection algorithm that is both fast and accurate [11]. Instead of using traditional methods like sliding windows or region proposals, YOLO treats object detection as a single regression task. This approach allows it to predict bounding boxes and class probabilities for objects in an image in one go, making it highly efficient.

3.3.1. Key Features of YOLO

- **Speed:** YOLO is designed for real-time object detection, making it a great choice for applications where quick decisions are important.
- **Unified Architecture:** The algorithm splits the input image into a grid and predicts bounding boxes and class probabilities for each grid cell simultaneously.
- **Generalization:** YOLO's unified design makes it good at detecting objects in new and unseen images, reducing the chances of false positives.

3.3.2. YOLOv11-OBB

For this project, the YOLOv11-OBB model is used, an enhanced version of YOLO that supports **oriented bounding boxes (OBB)**. Unlike traditional bounding boxes, which are axis-aligned rectangles, OBBs are rotated rectangles that can align with the

actual orientation of objects in an image. This feature is especially useful for detecting objects that are found in various orientations such as in a manufacturing line. It also helps to identify partially, the pose of the object if need be. For example, in pick and place applications.

The use of OBBs provides several advantages:

- **Improved Accuracy:** By aligning bounding boxes to the object's actual orientation, OBBs reduce background noise and make detection more precise.
- **Better Geometry Representation:** OBBs capture the true shape and pose of objects, which is critical for applications involving complex geometries.



Figure 3.3: Yolo: Default vs OBB

3.3.3. Applications in the Thesis

YOLO plays a key role in detecting 3D-printed objects from both synthetic and real-world images. The addition of OBBs through YOLOv11-OBB further strengthens its utility in this thesis. Specific uses include:

- **Detection with CAD-Generated Data:** YOLOv11-OBB is trained to recognize object features and orientations based on the given data.
- **Real-World Validation:** The trained model is tested on real-world images of 3D-printed objects to evaluate its ability to generalize from synthetic to real data. The inclusion of OBBs improves detection of objects with complex shapes and variable orientations.
- **Integration with Post-Processing:** The oriented bounding boxes predicted by YOLOv11-OBB are used to refine detection results by comparing the detected objects with their CAD model attributes. This step enhances the overall reliability of the detection process.

3.4. Blender

Blender is a powerful open-source 3D modelling software used for modeling, rendering, and animation [5]. Its versatility and accessibility makes it a popular choice for generating synthetic datasets in computer vision research.

3.4.1. Key Features of Blender

- **3D Modeling:** Supports the creation and manipulation of complex geometries, essential for designing 3D objects.
- **Rendering:** Provides realistic lighting, textures, and shadows, which are crucial for generating high-quality synthetic images.
- **Customizability:** Offers Python scripting to automate tasks such as object positioning, rendering, and dataset annotation.

3.4.2. Applications in the Thesis

Blender is utilized in this thesis to create a synthetic dataset from CAD models of 3D-printed objects. Its capabilities are leveraged as follows:

- **Conversion of CAD Models to Rendered Images:** Blender can be used along with python scripts to import CAD models and render them under various lighting conditions and backgrounds.
- **Scene Simulation:** Realistic scenes are created by adding background textures, and clutter to simulate real-world conditions.
- **Dataset Annotation:** We can also generate bounding boxes, segmentation masks, and depth maps to automate the annotations process to train the YOLO models.

Blender's ability to produce photorealistic synthetic data significantly reduces the dependency on real-world datasets, addressing challenges such as limited availability and high annotation costs.

3.5. BlenderProc

BlenderProc is an open-source procedural pipeline built on top of Blender, designed for efficient synthetic data generation and dataset annotation [4]. It automates many of the tedious aspects of rendering and labeling data, making it ideal for creating large-scale datasets.

3.5.1. Key Features of BlenderProc

- **Material Randomization:** Automatically assigns random textures and materials to objects, enhancing model generalization.
- **Lighting Variability:** Introduces variations in lighting conditions, improving robustness to real-world scenarios.
- **Automated Annotation:** Generates bounding boxes, segmentation masks, and depth maps without manual intervention.
- **Physics Simulation:** Supports physically accurate object placement and interactions to create realistic scenes.

3.5.2. Applications in the Thesis

BlenderProc is a crucial tool in this thesis for efficiently generating a diverse synthetic dataset from CAD models. Its contributions include:

- **Scripting framework:** Blenderproc provides a framework and an abstraction layer to blenders default API. This helps us enormously in automating the process of dataset generation.

- **Annotation Pipeline:** Produces annotations such as bounding boxes and segmentation masks required for training YOLO.
- **Scalability:** Enables the generation of large datasets efficiently, making it feasible to train deep learning models with extensive variations.

BlenderProc significantly accelerates the dataset creation process while ensuring diversity and realism, bridging the gap between synthetic and real-world data.

3.6. Style transfer

This is the method that will be used and tested in this thesis. Style transfer means to apply a certain type of style onto an image. Below are a few examples of style transfer.

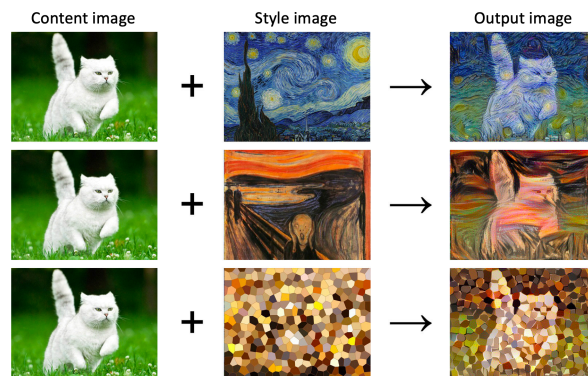


Figure 3.4: Example style transfers

As seen in figure 3.4 the style image or filter in the center is applied onto the cat image so as to obtain a stylized output image. This example is a very evident and extreme case of style transfer, but for the purposes of this thesis, a more simple algorithms of style transfer are used such as the one described in the paper: Learning to generate line drawings[3]. Along with this, much simpler algorithms are also used such as canny and similar edge detection algorithms to test if they can perform better than colored images. These algorithms can be more classified as filters rather than style transfer algorithms.

Two main purposes exist to apply styles and filters to images:

- To reduce the sim-to-real gap. This gap can be described as the differences in various attributes when comparing real life to a simulated environment. This gap exists in many domains and in this thesis the domain of images is looked at. Using these algorithms to change the way images look, the ones that are rendered can look much closer to ones in the real world.
- To enable detection of objects invariant to the color, lighting and other environmental changes that can change the way the image looks.

The following are the methods used to apply a filter/style transfer to the images

3.6.1. Canny Edge Detection

Canny edge detection is one of the most widely used edge detection algorithms due to its ability to detect edges efficiently while suppressing noise [2]. It operates in the following steps:

- **Noise Reduction:** A Gaussian filter is applied to smooth the image, reducing noise that may lead to false edges.
- **Gradient Calculation:** The image's intensity gradients are computed to identify areas with significant changes in intensity.
- **Non-Maximum Suppression:** Thin edges are retained by suppressing all gradient values that are not local maxima.
- **Edge Tracking by Hysteresis:** Thresholding is used to classify strong edges and weak edges, with connectivity ensuring weak edges are retained if they are connected to strong edges.

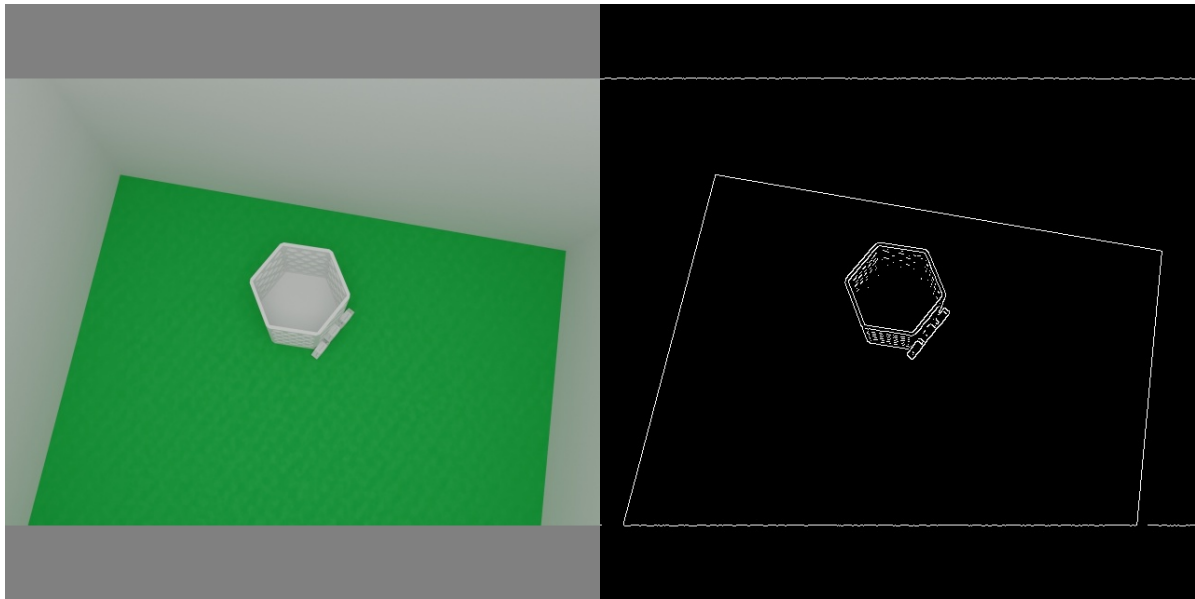


Figure 3.5: Color vs Canny

3.6.2. Active Canny Edge Detection

Active Canny Edge Detection is an enhanced version of the traditional Canny edge detection algorithm, incorporating active contour models to refine edge detection. Unlike the standard Canny algorithm, which relies solely on intensity gradients, Active Canny integrates shape priors and geometric constraints, making it more effective in detecting edges for complex objects with predefined structures [16].

- **Integration with Active Contours:** Combines Canny edge maps with active contour models (snakes) to iteratively refine edge boundaries.
- **Shape Awareness:** Uses prior knowledge about object shapes to improve edge localization, particularly useful for detecting objects with consistent geometries, such as 3D-printed parts.
- **Robustness to Noise:** Enhanced noise reduction by incorporating edge tracking mechanisms that align with shape constraints.

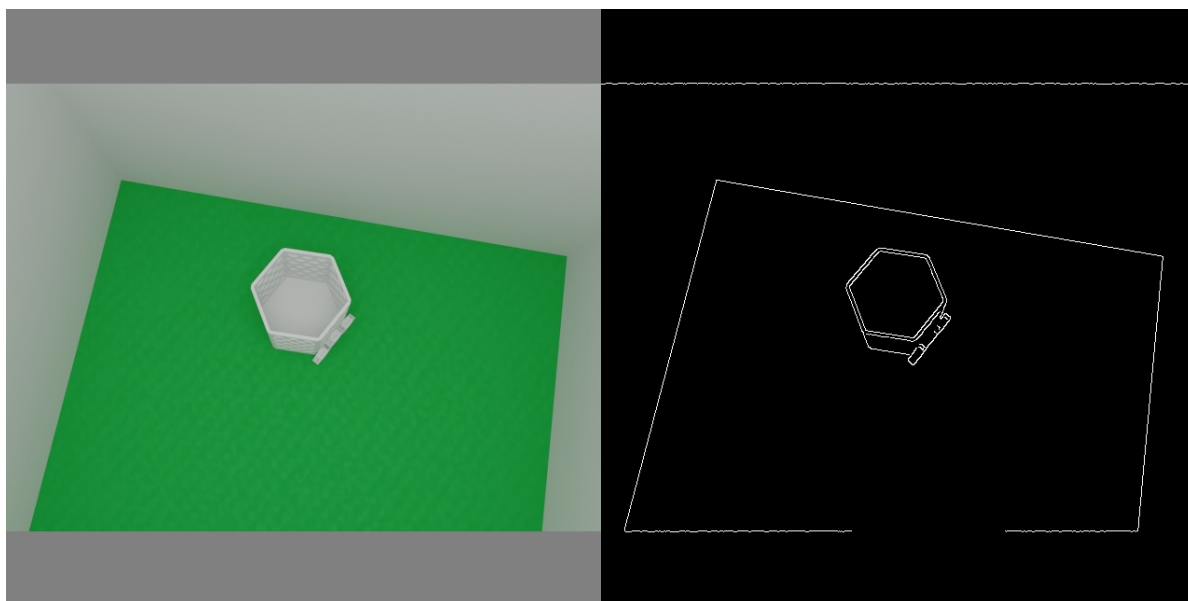


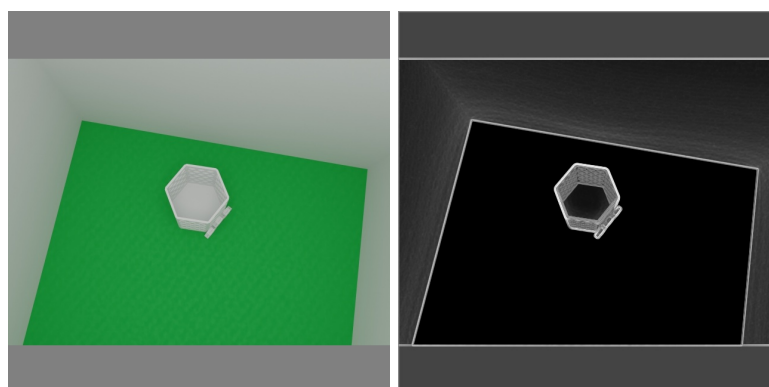
Figure 3.6: Color vs Active Canny

3.6.3. Holistically-Nested Edge Detection (HED)

Holistically-Nested Edge Detection is a deep learning-based edge detection algorithm that predicts edges at multiple scales [15]. Unlike traditional methods, HED leverages a CNN to learn edge features in a data-driven manner.

- **Multi-Scale Learning:** HED uses a Fully Convolutional Network (FCN) to extract features at different levels of abstraction, enabling it to detect both fine and coarse edges.
- **Holistic Training:** The network is trained holistically, ensuring all scales contribute to the final edge prediction.
- **End-to-End Framework:** The model produces a single unified edge map, simplifying the post-processing pipeline.

Figure 3.7 shows the color image progressively generated through each layer.



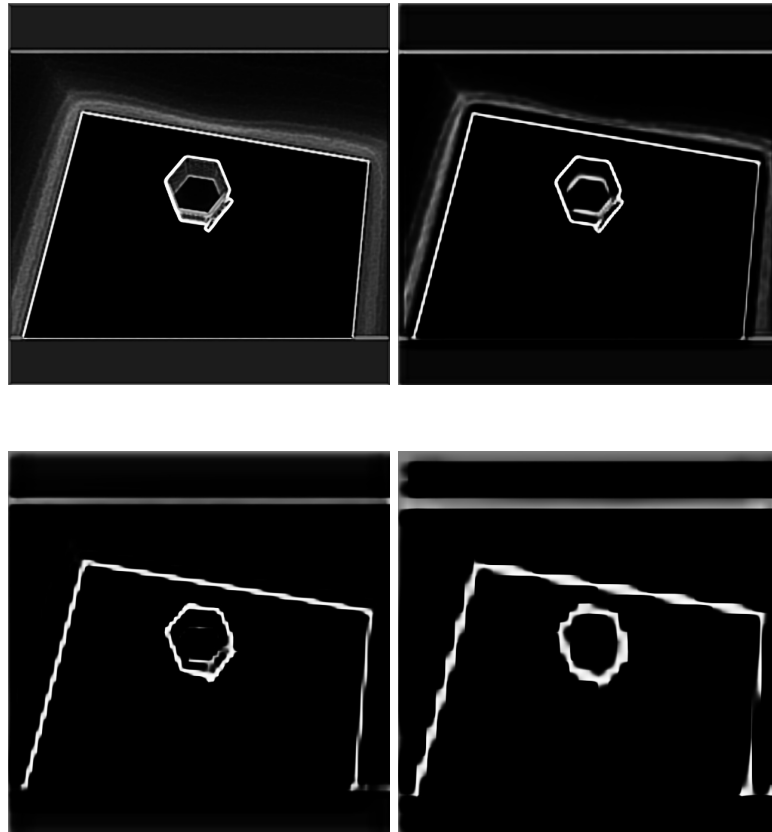


Figure 3.7: Color vs HED layers

3.6.4. Informative Drawings

This is a new method to generate sketch like images by using an image translation model to generate line drawings suing objectives based on these observations [3].

- **Variety:** This model has been trained on 3 different styles, namely, anime style, contour style, and opensketch style.
- **Performance:** This method outperforms all state of the art methods at creating line drawings that keep the most important edges and features while applying the style.

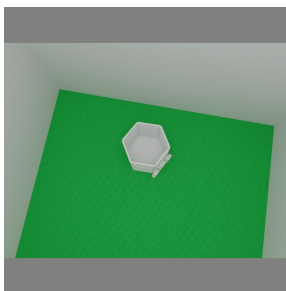


Figure 3.8: Colored image

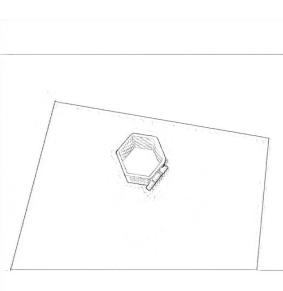


Figure 3.9: Anime style

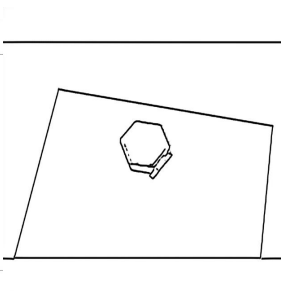


Figure 3.10: Contour style

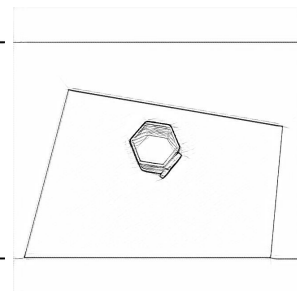


Figure 3.11: Opensketch style

Figure 3.12: Color vs Line drawings

4

Design

Designing a robust and modular data generation pipeline is critical to ensuring the success of the project. The design decisions outlined in this chapter reflect the requirements and constraints that guide the implementation process.

4.1. 3D Rendering Software

A modular and configurable rendering software is fundamental for generating synthetic datasets. This environment must allow for:

- Manipulating objects, including position, orientation, and scale.
- Modifying the background and lighting to introduce variety into the dataset.
- Applying diverse textures and colors to the objects to simulate realism.

To fulfill these requirements, Blender and Unity were considered as suitable options. Blender was chosen due to better documentation and user base. It also provides better API's to programatically interact with the scene and rendering environment. Refer to Section [3.4](#) for details about Blender.

4.2. Rendering Environment

With Blender now chosen as the software to generate the images, there needs to be a scene within which the 3D model of the object can be placed and manipulated while varying the environmental conditions to generate diverse images. A scene, in this context, can be defined as a 3D environment that contains all the necessary components, such as lighting, camera setup, and background elements, to facilitate high-quality rendering of the object.

The design of the rendering environment includes the following considerations:

- **Controlled Lighting:** To ensure the scene is well-lit and shadows are present to depict depth, two adjustable lights are placed near the ceiling of the rendering box. These lights can change their angles to vary the shadows and lighting intensity for each render.
- **Greenscreen Floor:** One wall of the rendering box is painted green to simulate a greenscreen, making it easier to segment the object from the background. This

is done as a control variable, but this can be varied with different textures to also add robustness to the model.

- **Camera Placement:** A single camera is fixed inside the scene and is aligned to focus on the object, ensuring that the object is always fully captured and centered in the frame. The render of the scene happens through the view of the camera. Its settings are fine-tuned for optimal field-of-view and rendering quality.
- **Object Positioning:** The 3D model of the object is placed randomly within the box while ensuring that it is fully visible and not clipped by the frame. This introduces variability in the object's position, orientation, and perspective in the dataset.

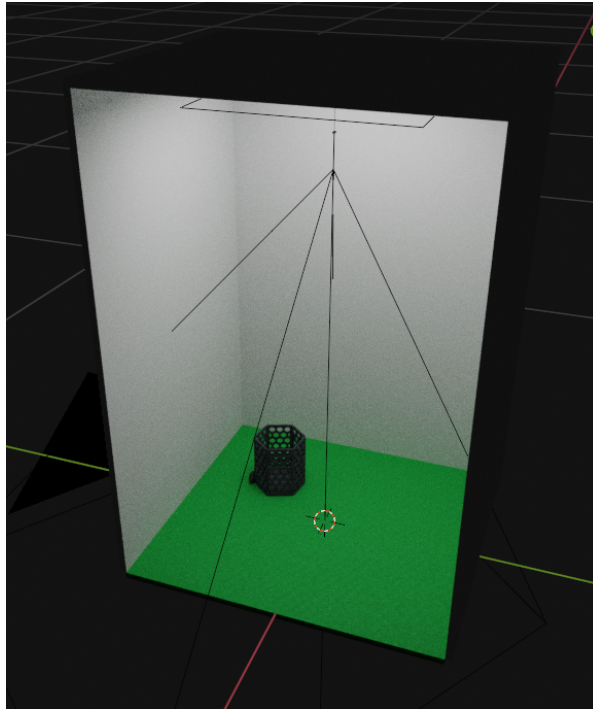


Figure 4.1: Blender scene for rendering

4.3. API for Automation

The interface between the rendering software and the automation scripts must enable full control over the 3D environment. This includes automating:

- Object placement and configuration.
- Scene manipulation, including lighting, textures, and backgrounds.
- Rendering high-quality images efficiently.
- Generating annotations and masks for the rendered data.

To meet these requirements, **BlenderProc** and **Blender API (bpy)** were selected. BlenderProc extends Blender's capabilities, making it easier to automate complex rendering workflows, while bpy provides low-level control over Blender's features. The combination of these tools ensures seamless automation of the rendering and annotation processes. Details on these tools are provided in [Section 3.5](#).

4.4. Style Transfer Algorithms

Bridging the simulation-to-reality (sim-to-real) gap requires innovative methods to process and enhance synthetic images. To this end, multiple style transfer algorithms are incorporated into the design to test their impact on object detection accuracy. The chosen styles include:

- **Canny Edge Detection:** A classical edge detection algorithm used to retain essential contours. Refer to subsection (3.6.1).
- **Active Canny Edge Detection:** An advanced version that enhances weak or missing edges for greater detail. Refer to subsection (3.6.2).
- **Holistically-Nested Edge Detection (HED):** A deep learning approach to edge detection, combining multi-layered edge predictions. Refer to subsection (3.6.3).
- **Informative Drawings:** A state-of-the-art technique generating line drawings with enhanced 3D shape and semantic details. It includes:
 - Anime Style
 - Opensketch Style
 - Contour Style

Refer to subsection (3.6.4) for more detail.

These algorithms were chosen for their ability to retain meaningful features while removing distractions like lighting and color.

5

Implementation

Now that the tools to create a pipeline are ready, it needs to be put together to create a successful data generation pipeline.

Referring to figure 3.1, the pipeline can be broken down into 3 main sections. Image generation, dataset creation and model training. Each step can be broken down into smaller parts to easier understand it:

5.1. Image generation

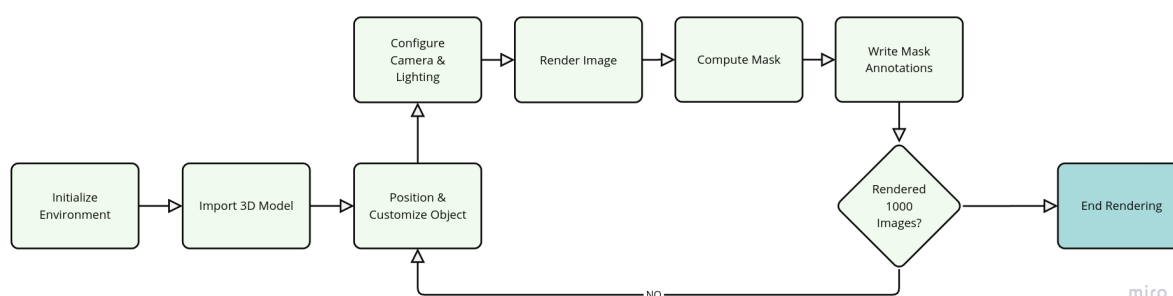


Figure 5.1: Pipeline: Image Generation

Image generation is the most important step in the pipeline. If the images that are generated, look realistic with lot of variety in the dataset, then the sim to real gap is already reduced by a great margin. The rendering and modification of the object is handled by a single python script¹. The process is as follows:

- **Inputs:** The input for the script is the path to the 3D model of the object to train on.
- **Initialize the environment:** The script then sets up a BlenderProc environment and loads a predefined Blender scene file. It customizes rendering parameters, including those for the Cycles render engine², lighting configurations, and camera settings. All these changes and settings are done to enhance rendering quality.

¹https://github.com/nggyu69/Bachelor_Thesis/blob/master/Generate.py

²<https://www.cycles-renderer.org/>

- **Importing and Configuring the 3D Model:** A 3D model in STL format is imported from a user-defined file path. The script calculates the model's centroid to center its geometry, applies scaling, and assigns it a category based on a category mapping. This model is then designated as the primary "train object" and integrated into the scene by calculating its position to place it onto the greenscreen.
- **Object positioning and Customization:** The model is positioned randomly within the scene on top of the greenscreen. It also ensures that the complete object is within the box and nothing is cut off. Along with this the model color is also changed by a user proved hex code.
- **Camera and Lighting:** Next, the script configures camera and lighting to optimize the scene. The camera is oriented such that it is pointing to the center of the object. The lighting angle is also angled such as to bring in more shadows to reveal the depth of the object.
- **Rendering and writing masks:** The final step in this script it to render the image. The rendering is set to high quality in order to bring out the realism of the scene. Along with the rendered image, the segmented mask annotations are also written out. This is possible due to the fact that the position of the selected object is know by default and the mask for the object relative to the scene can be computed. This mask is then used in the next steps to annotate and label the images.

The whole process above is to render a single image. This is repeated 1000 times per object. The above script¹ is controlled by a bash script³.

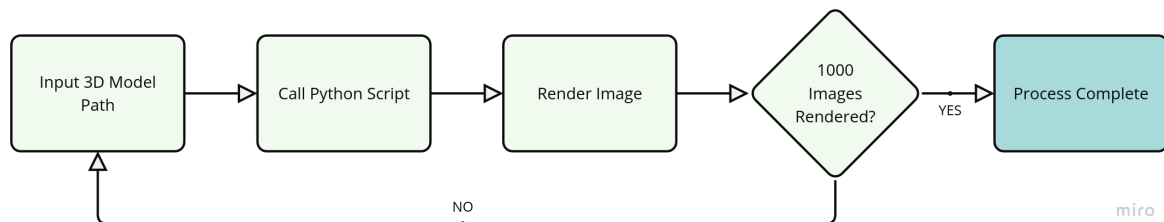


Figure 5.2: Pipeline: Image Generation Control

The functionality of the script can be described as follows:

- It accepts a 3D model path as input by the user. It can also be a path which consists of multiple 3D models and it will repeat the process for each object.
- It runs the python script by passing the 3D model path of one object and a color as arguments to the python script¹. It then waits for the rendering to be complete and proceeds.
- Once the rendering is complete, it checks if 1000 images are rendered already. If it is, it either continues onto the next 3D model if more than one was given by the user or ends the program.

This script is present to help do all the rendering in the background and also randomly provide the color parameters. It also ensures that the generation script does not stop before it has generated all the 1000 images. Both the scripts help automated the entire image generation process without any intervention.

³https://github.com/nggyu69/Bachelor_Thesis/blob/master/run_gen.sh

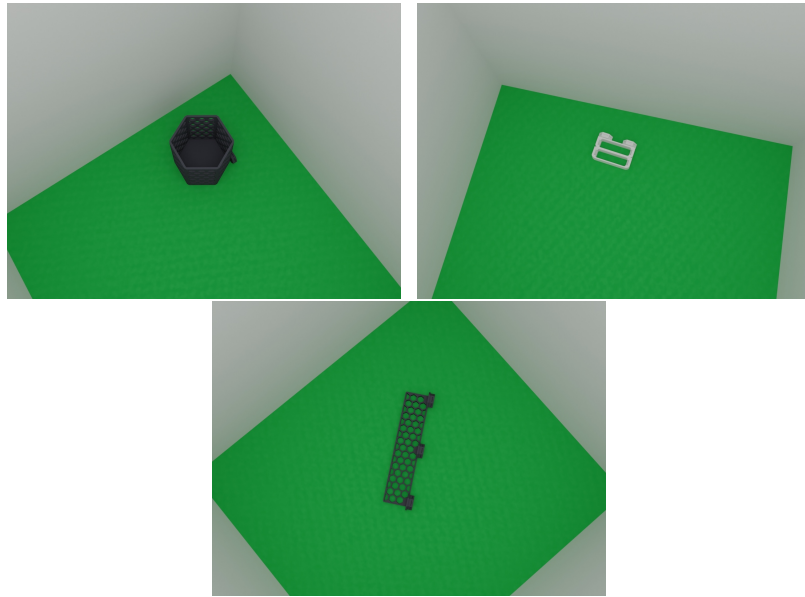


Figure 5.3: Generated images

5.2. Dataset Preparation

This is the step that gives this pipeline an edge and an efficiency boost when compared to manual dataset generation. Since the object was placed and manipulated in the scene by the program itself and the geometric CAD data is available to the program, the script can determine where in the scene it is located along with the mask up to the exact pixel in the image. This allows the pipeline to automatically label these images to their respective classes in a matter of minutes. This step is also handled by a python script⁴.

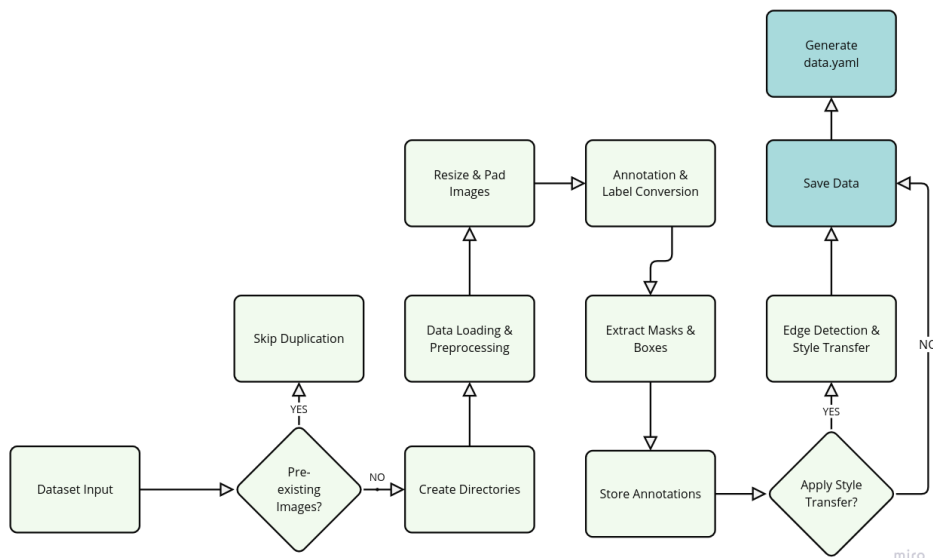


Figure 5.4: Pipeline: Dataset Preparation

⁴https://github.com/nggyu69/Bachelor_Thesis/blob/master/Annotate.py

The process of annotating and dataset generation is explained in the following steps:

- **Dataset Organization and Structure Configuration:** The script establishes a systematic directory framework for the storage of eight different dataset for each style(control, canny, anime-style, contour-style, etc.) designated for training, testing, and validation purposes. It verifies the existence of subdirectories for both images and labels within these formats. Furthermore, it creates directories specifically for annotated and masked images. The script then scans the directories to identify any pre-existing images, thereby preventing duplication and thereby additional processing.
- **Data Loading and Preprocessing:** This process involves handling a collection of directories that contain image and annotation data formatted according to COCO standards. These directories are the output of the previous step wherein the images were generated. Each image undergoes resizing and padding to conform to a standardized dimension of 640x640 pixels, while preserving the original aspect ratio. The annotations, which encompass segmentation masks and bounding boxes, are modified in accordance with the resizing and padding adjustments made to the images.
- **Annotation and Label Conversion:** The script is designed to handle COCO-format annotations, facilitating the extraction of segmentation masks, bounding boxes, and object categories. The bounding boxes are resized and modified to align with the new image dimensions, and the resulting annotations are stored in text files that are compatible with the yolo obb standard. Each annotation is stored into the dataset according to its designated split (training, testing, validation).
- **Style Transfer:** Style transfer is the integral part of this thesis where different styles are tested to see which one gives the most accurate results in a variety of environments. These style transfer/edge detection algorithms are meant to make the real and rendered images look more similar to each other by removing various factors like lighting, color and keeping the most important features in the end image. For each image, a range of edge detection techniques (such as Canny, active Canny, and HED) and style transformations (including anime-style, contour-style, and opensketch-style) are implemented. The processed images are then saved in their respective directories, adhering to a uniform naming convention. Refer to section (3.6) for a description on each style used.
- **Saving data:** The images, annotations, and masks that have been processed are stored in designated directories. In order to test and compare different styles, each style has a separate dataset. Additionally, the script produces a data.yaml configuration file that encompasses metadata for the dataset. This includes the paths to the training, testing, and validation images along with their associated labels and class names. This configuration file is a prerequisite for the next training of the Yolov11-obb model.

5.3. Model training

The final step, after generating all the datasets, is to train the computer vision model to detect objects. In this thesis, the YOLOv11-OBB model⁵ is used. This model employs the concept of Oriented Bounding Boxes (OBB), which enables the detection of objects in any orientation by adapting the bounding box to the object's orientation during both training and inference. Refer to subsection (3.3.2).

⁵<https://docs.ultralytics.com/tasks/obb/#models>

The training process leverages transfer learning, where a pre-trained YOLOv11 model serves as the foundation. This approach utilizes the knowledge already encoded in the model from training on large, generic datasets, significantly reducing the time and computational resources required. For this thesis, only the final layers of the model are trained, and their weights updated. These layers are responsible for learning dataset-specific features, such as the characteristics of the objects being detected in the generated synthetic datasets. By freezing the earlier layers of the network, the model retains its generalized feature extraction capabilities, while the fine-tuned final layers adapt to the new domain.

Similar to the image generation step, where a bash script controls the python script which does the rendering, there is also a bash script⁶ which controls the python training script⁷. This is also done for the same reasons as above, which is to prevent the script from exiting before the training is complete and to let the training run in the background while also logging the output. A brief overview of the steps are given below:

- **The bash script:** This script runs the training script for each different style and passes the dataset path for the particular style. This is possible because the annotation script has already made eight different datasets, one for each style, split them into train, test and validation sub folders and ready to be trained on. The script also monitors the training process and alerts the user if there are any errors during training.
- **The training script:** This accepts the style name and dataset path as arguments from the bash script and reads the data.yaml file from that specific directory which further contains paths inside the dataset regarding the images and labels corresponding to them. Next, it starts the training for that dataset and stores it in a folder names trains under the name of the style. This is not only done for all the eight different styles, but also for the 5 different sizes of the models. This is to test the inference time vs accuracy of the model, across different sizes of models. As shown in table 5.1, sizes differ by how many parameter can be tuned for the model. The smallest model would have much lesser parameters to calculate and hence training of the model can be done much quicker as compared to the largest model. Even the inference time as shown in the table, is much faster for the smaller models when compared to the larger models. But as it is evident, in the default pretrained configuration for the yolov11-obbb model which was pre-trained on the DOTAv1⁸ dataset, the mean average precision(mAP) increases as the size of the model increases. Hence it is important to test which model has the best balance of speed vs accuracy.

Model	mAP _{test} (%)	Speed (CPU ONNX) (ms)	Params (M)
YOLO11n-obbb	78.4	117.6 ± 0.8	2.7
YOLO11s-obbb	79.5	219.4 ± 4.0	9.7
YOLO11m-obbb	80.9	562.8 ± 2.9	20.9
YOLO11l-obbb	81.0	712.5 ± 5.0	26.2
YOLO11x-obbb	81.3	1408.6 ± 7.7	58.8

Table 5.1: Performance Metrics of YOLOv11-OBb Models

⁶https://github.com/nggyu69/Bachelor_Thesis/blob/master/run_train.sh

⁷https://github.com/nggyu69/Bachelor_Thesis/blob/master/train.py

⁸<https://github.com/ultralytics/ultralytics/blob/main/ultralytics/cfg/datasets/DOTAv1.yaml>

6

Evaluation

For evaluation, many models were trained on all the different styles and size to test which ones would perform the best. There are eight 3D printed objects used as the test subjects. In total 40 models were trained across eight styles/filters and five sizes of the YOLO model.

Now that all the models are trained and organized into their different weights in order of the styles, it is very easy to test them in different scenarios, lighting and variety of the object. The models are evaluated by the following metrics.

6.1. Validation Metrics

In the training of each model, there is a validation phase which gives some output about how the model is performing in with respect to images it has not seen before. These metrics include box loss, mean average precision with 0.5 IoU threshold(mAP50) which checks for the intersection over union of a predicted bounding box over the ground truth, mean average precision with 0.5-0.9 IoU and recall: which is a measure of how good the model is at finding all the objects it is supposed to detect. The ones focused on in this thesis is:

- **Box loss:** Box loss measures how well the predicted bounding boxes match the actual (ground truth) boxes around objects. It measurement is a combination of center loss(how close the centers of predicted and actual boxes are), size loss(size difference between the boxes), IoU loss(Measures how much the predicted box overlaps with the actual box).
- **mAP50-90:** mAP50-90 refers to the mean Average Precision (mAP) of the YOLO model, calculated across a range of Intersection over Union (IoU) thresholds, from 50% to 90%. It gives a comprehensive measure of how well the model performs at detecting objects with varying levels of precision.

Style Name	Box Loss	mAP50-90
HED2	0.13	0.99
Contour_Style	0.15	0.99
Opensketch_Style	0.16	0.99
HED1	0.16	0.99
Anime_Style	0.16	0.99
Canny	0.21	0.98
Color	0.22	0.98
Active_Canny	0.22	0.98

Table 6.1: Small Model Metrics

Style Name	Box Loss	mAP50-90
HED2	0.15	0.99
HED1	0.16	0.99
Opensketch_Style	0.16	0.99
Contour_Style	0.17	0.99
Anime_Style	0.19	0.99
Color	0.22	0.98
Canny	0.23	0.98
Active_Canny	0.23	0.98

Table 6.2: Nano Model Metrics

Style Name	Box Loss	mAP50-90
HED2	0.14	0.99
Contour_Style	0.14	0.99
Opensketch_Style	0.15	0.99
Anime_Style	0.15	0.99
HED1	0.20	0.99
Color	0.22	0.99
Canny	0.25	0.98
Active_Canny	0.27	0.97

Table 6.3: Medium Model Metrics

Style Name	Box Loss	mAP50-90
HED2	0.13	0.99
Opensketch_Style	0.14	0.99
Anime_Style	0.15	0.99
Contour_Style	0.15	0.99
Color	0.19	0.99
HED1	0.21	0.98
Canny	0.23	0.98
Active_Canny	0.24	0.98

Table 6.4: Large Model Metrics

Style Name	Box Loss	mAP50-90
HED2	0.13	0.99
HED1	0.14	0.99
Anime_Style	0.15	0.99
Contour_Style	0.15	0.99
Opensketch_Style	0.15	0.99
Color	0.21	0.99
Active_Canny	0.23	0.98
Canny	0.25	0.98

Table 6.5: Extra-Large Model Metrics

From these metrics the following observations can be made about box loss and mean average precision:

- Smaller models (Small, Nano) exhibit higher box loss compared to larger models (Large, Extra Large). This suggests that smaller models are less precise in their bounding box predictions.
- The box loss for the colored model(control) and edge detections filters such as canny and active canny tend to perform worse than models that use neural style transfers.
- The mAP50-90 remains consistently high (around 0.99) across all YOLO model sizes, indicating that all models are able to achieve excellent detection performance, irrespective of the size.

6.2. Inference time

In this thesis, the inference time refers not only to the time that takes to model to process an input image and produce the detections(bounding boxes) as the output, but also the preprocessing time which includes resizing and padding the image and to apply a style to the image. This is tested on 2 different systems: one with a powerful GPU, the Nvidia A100, which is meant for AI tasks and one with a weaker GPU, the NVIDIA 1650. A short differentiation is done in table 6.6 between the 2 GPUs. The inference time is tested for a total of all 40 models(8 styles x 5 model sizes) and the inference time is averaged out for each style under each size.

Specification	NVIDIA A100	GTX 1650
VRAM (GB)	40/80	4
Tensor Cores	432 (3rd Gen)	No
Ray Tracing Cores	No	No
CUDA Cores	6912	1024

Table 6.6: NVIDIA A100 vs GTX 1650

Table 6.6 shows the difference between the A100 and 1650. The A100 is better than the 1650 in every aspect and also costs a hefty bit more. The A100 is suited better for AI tasks which requires more tensor cores and CUDA cores. It is also better suited for 3D rendering applications as it has more Ray tracing cores which can compute

Style	A100	GTX 1650
Color	16.60	15.43
Canny	33.80	35.32
Active_Canny	47.98	47.94
anime	125.57	309.66
opensketch	123.07	318.71
contour	123.63	313.44
HED2	2474.85	2521.13
HED1	2532.26	2445.48

Table 6.7: Inference Time: Nano model(ms)

Style	A100	GTX 1650
Color	17.57	20.14
Canny	38.74	41.56
Active_Canny	50.23	54.42
anime	124.31	316.31
opensketch	123.91	326.67
contour	122.90	317.94
HED2	2628.25	2662.89
HED1	2889.56	2654.13

Table 6.8: Inference Time: Small model(ms)

Style	A100	GTX 1650
Color	16.22	33.24
Canny	35.87	52.85
Active_Canny	48.75	66.62
anime	131.36	338.72
opensketch	126.37	337.73
contour	126.01	341.88
HED2	2630.05	2553.59
HED1	2774.77	2574.19

Table 6.9: Inference Time: Medium model(ms)

Style	A100	GTX 1650
Color	19.62	40.28
Canny	42.27	61.40
Active_Canny	74.29	73.86
anime	128.95	338.58
opensketch	128.38	345.19
contour	156.59	340.35
HED2	2810.33	2783.36
HED1	2852.65	2681.06

Table 6.10: Inference Time: Large model(ms)

Style	A100	GTX 1650
Color	20.24	80.68
Canny	38.01	104.08
Active_Canny	54.82	115.01
anime	127.04	386.48
opensketch	126.15	394.81
contour	129.23	392.73
HED2	2549.12	2640.72
HED1	2562.47	2807.04

Table 6.11: Inference Time: Extra Large model(ms)

There following conclusions are drawn from the comparison above:

- The inference time naturally increases as the model size increases, no matter the GPU. It is more evident in the weaker GTX 1650 than in the A100. The A100 so capable at this task that model size does not affect inference time by much at all.
- Within each model size, there 2 different type of styles, neural style transfer(see sections 3.6.4 and 3.6.3 and basic edge detection algorithms(see sections 3.6.1 and 3.6.2. Between these two types, the pre processing time for the edge detections algorithms are much lesser that the time taken to apply a style transfer. This can be seen in the HED models especially as they take significantly longer to infer, which is due to the time taken detect the edges using the HED network.

6.3. Confidence

This metric measures the average confidence of each detected object in a particular image. This helps to understand how confident the model is about a detected object and if there are chances it can have false positives or negatives.

Style	Average Confidence (%)
Opensketch	95.0
Contour	94.0
HED1	92.0
Color	92.0
Anime	91.0
HED2	90.0
Active Canny	81.0
Canny	76.0

Table 6.12: Average Confidence: Nano size

Style	Average Confidence (%)
Contour	95.0
Opensketch	95.0
Anime	94.0
Color	92.0
HED1	88.0
HED2	85.0
Active Canny	82.0
Canny	71.0

Table 6.13: Average Confidence: Small size

Style	Average Confidence (%)
Anime	95.0
Contour	95.0
Opensketch	95.0
Color	91.0
HED1	87.0
Canny	80.0
HED2	79.0
Active Canny	60.0

Table 6.14: Average Confidence: Medium size

Style	Average Confidence (%)
Opensketch	96.0
Anime	95.0
Contour	94.0
HED2	93.0
Color	92.0
HED1	88.0
Canny	81.0
Active Canny	60.0

Table 6.15: Average Confidence: Large size

Style	Average Confidence (%)
Contour	95.0
Opensketch	95.0
Anime	94.0
HED1	92.0
Color	90.0
HED2	89.0
Canny	78.0
Active Canny	64.0

Table 6.16: Average Confidence: Extra-Large size

From the above tables the following are observed:

- The anime, contour and opensketch style perform better across all model style when it comes to the confidence with which it detects object. This could say that the models are confident at detecting the true positives in the image.
- The two canny algorithms are the worst performers across different sizes.

But for the above observations, one must also take into account how many object were there in the image. Without taking that into account, an image having five objects but only one detection with 95 percent confidence can mislead one to believe that this model performs better than the other. Hence taking the number of objects to be detected into account, there can be another metric that is introduced that takes into account the percentage of objects detected, and then calculating a weighted confidence.

6.4. Weighted Confidence

Style	Avg. Confidence(%)
Contour Style	93.09
Opensketch	89.64
Anime Style	88.26
Color	77.85
HED2	60.96
HED1	56.99
Active Canny	53.20
Canny	50.46

Table 6.17: Weighted Confidence: Small Model

Style	Avg. Confidence(%)
Opensketch	92.71
Contour Style	92.44
Anime Style	88.76
Color	72.52
HED1	71.33
HED2	57.69
Active Canny	53.85
Canny	51.58

Table 6.18: Weighted Confidence: Nano Model

Style	Avg. Confidence(%)
Opensketch	95.06
Anime Style	94.93
Contour Style	92.35
Color	74.98
HED1	62.72
Canny	55.16
HED2	43.93
Active Canny	36.26

Table 6.19: Weighted Confidence: Medium Model

Style	Avg. Confidence(%)
Opensketch	93.97
Anime Style	91.34
Contour Style	88.98
Color	82.99
Canny	68.25
HED2	62.53
HED1	62.75
Active Canny	40.46

Table 6.20: Weighted Confidence: Large Model

Style	Avg. Confidence(%)
Anime Style	94.20
Opensketch	91.81
Contour Style	85.53
HED1	79.98
Color	79.87
HED2	61.67
Canny	55.18
Active Canny	46.04

Table 6.21: Weighted Confidence: Extra Large Model

The tables above hold the truth as to which models perform best in the real world. Here are a few observations:

- Firstly, the models trained using images with edge detection algorithms(canny, active canny and the HED models) did poorly as compared to the style transfer methods. The model trained with color images detect the objects rather averagely as well. They are better than the edge detection algorithms, but not as good as the style transferred image models.
- In the previous section, the HED models had good average confidence, leading to believe that the model is good for detecting the objects. It can be said that this is partly true, but the reality is, while the model does have good confidence with which objects are detected, it is not capable of detecting all the objects in the scene.

Figures 6.1-6.7 show the prediction of the large model on real images using different algorithms. As clearly seen, the anime style and opensketch style transferred images and models detect the most object in the scene with a very high confidence level. This is also backed by table 6.21.

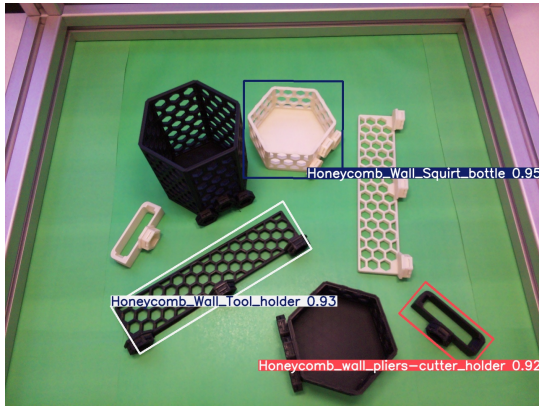


Figure 6.1: Prediction: Control Image

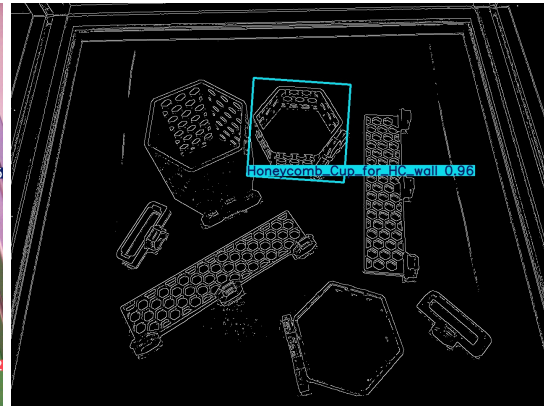


Figure 6.2: Prediction: Canny Image

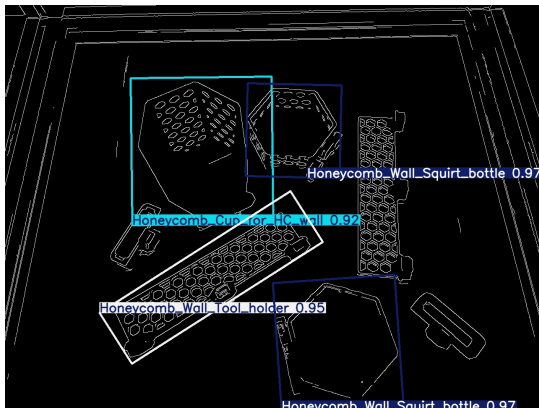


Figure 6.3: Prediction: Active Canny Image

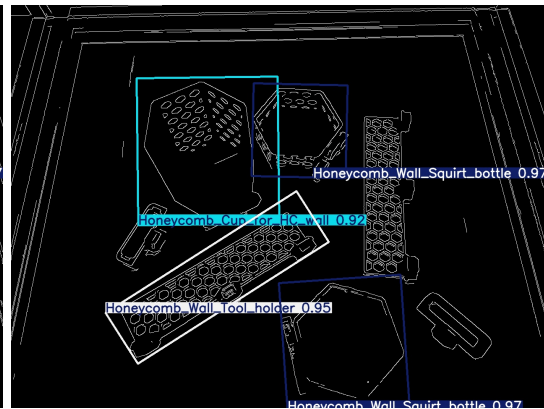


Figure 6.4: Prediction: HED1 Image

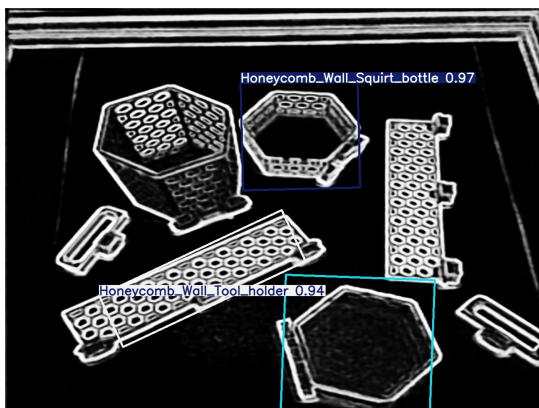


Figure 6.5: Prediction: HED2 Image

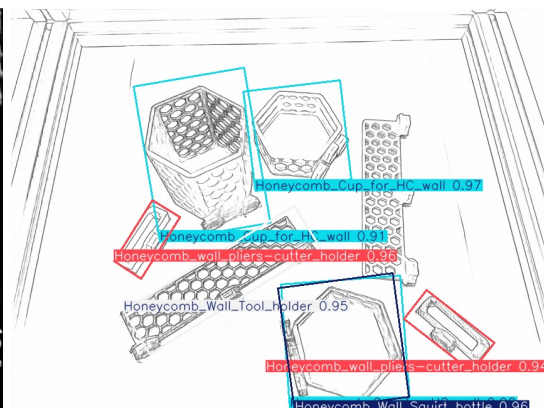


Figure 6.6: Prediction: Anime Style Image

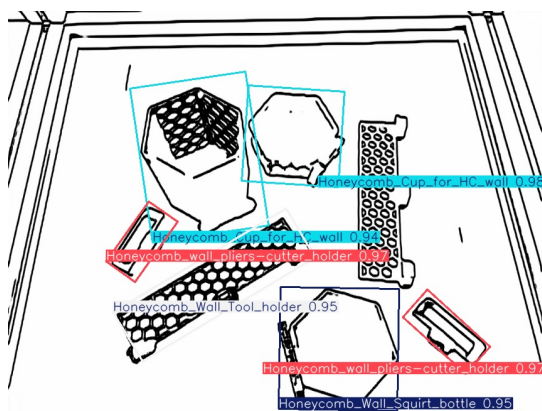


Figure 6.6: Prediction: Contour Style Image

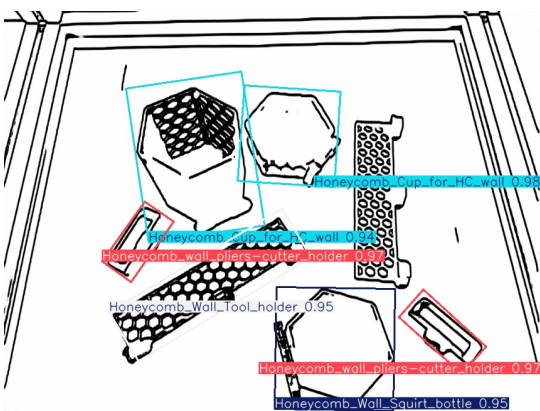


Figure 6.7: Prediction: Opensketch Style Image

7

Conclusion

This thesis implemented a pipeline for the detection of 3D-printed components using CAD models and different style transfer techniques to bridge the sim-to-real gap. By leveraging synthetic datasets generated through Blender and BlenderProc, combined with various style transfer methods, the research achieved significant improvements in accuracy, efficiency, and robustness over existing methodologies.

The proposed approach demonstrated:

- **High Accuracy:** The YOLOv11-OBB model trained on datasets augmented with style-transferred images (e.g., anime, contour, and opensketch styles) consistently achieved superior mAP (Mean Average Precision) scores across all tested scenarios. The anime style showed a remarkable balance between precision, confidence and inference time, outperforming both traditional edge-detection filters (e.g., Canny) and real-world image datasets.
- **Reduced Sim-to-Real Gap:** Style transfer, when applied on both training images and images taken in real life while performing object detection, made them look significantly similar, enabling better generalization of the model to real-world conditions.
- **Improved Confidence:** Models trained using style-transferred datasets exhibited higher detection confidence and reduced false positives compared to those trained with colored or edge-detection images.
- **Invariant to color:** Even though the models were trained only with black or white colors for the objects, when new colors are to be tested on the object, it performs without any problem. This is due to the style transfer. An example can be seen in Fig. 7.1

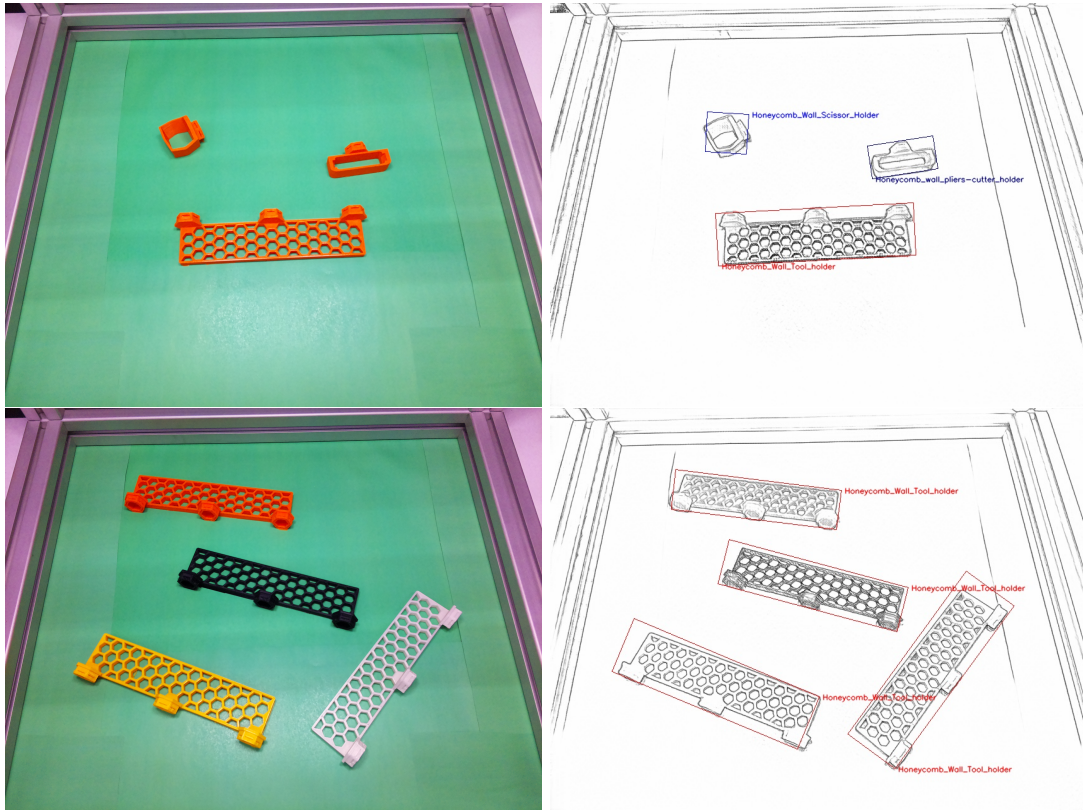


Figure 7.1: Prediction: New colors

When compared to previous work, this method provides the following advantages:

- Unlike the point-cloud-based approach of Lee et al.(2021)[8], which struggled with reflective surfaces and intricate geometries, this thesis employed image-based techniques that bypassed these limitations while maintaining high detection accuracy. It also reduced the complexity and processing power needed to handle 3D data.
- Compared to H3DNet by Zhang et al. (2020)[17], which focused on hybrid geometric primitives for 3D object detection, this thesis showed that image-based methods could achieve comparable, if not better, results for 3D-printed objects with lower computational power and complexity.
- Compared to all previous methods, style transfer methods outperformed normal colored images during detection, in terms of number of accurate detections, precision, as well as the confidence with which it detects the objects.

Even though there were positive results from the thesis, the following areas can possibly be improved upon, which can produce better results and more accurate detections:

- After all the development was complete, it could be noticed that there was lesser variation between images, in terms of the angles with which the photos were taken. It could be possible to better the models' accuracy if more angles were captured into the dataset.
- Having more images with more variety could improve the model's accuracy and confident detection of objects.
- In this approach the model has to be retrained on the whole dataset, every time it

has to detect a new object. This could be improved by using an approach which can retrain the model only with the new dataset.

This approach advances the field of object detection by addressing key challenges in synthetic dataset generation, sim-to-real transfer, and model generalization. Future research can build on these findings by exploring additional style transfer methods, optimizing computational workflows, and expanding the approach to other domains.

References

- [1] Yizhak Ben-Shabat, Michael Lindenbaum, and Anath Fischer. “3D Point Cloud Classification and Segmentation using 3D Modified Fisher Vector Representation for Convolutional Neural Networks”. In: *CoRR* abs/1711.08241 (2017). arXiv: 1711.08241. URL: <http://arxiv.org/abs/1711.08241>.
- [2] John Canny. “A computational approach to edge detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8.6 (1986), pp. 679–698.
- [3] Caroline Chan, Fredo Durand, and Phillip Isola. *Learning to generate line drawings that convey geometry and semantics*. 2022. arXiv: 2203.12691 [cs.CV]. URL: <https://arxiv.org/abs/2203.12691>.
- [4] Maximilian Denninger et al. “BlenderProc: A procedural pipeline for photorealistic rendering of large datasets using Blender”. In: *arXiv preprint arXiv:1911.01911* (2019). URL: <https://arxiv.org/abs/1911.01911>.
- [5] Blender Foundation. “Blender - a 3D modeling and rendering suite”. In: (2024). <https://www.blender.org>.
- [6] Diederik P Kingma and Max Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013). URL: <https://arxiv.org/abs/1312.6114>.
- [7] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. URL: <https://ieeexplore.ieee.org/document/726791>.
- [8] Cheng-Hsuan Lee and Chia-Hung Huang. “Generating Datasets from 3D CAD Models for Object Detection”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2021. URL: <https://ieeexplore.ieee.org/document/9576247>.
- [9] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133. URL: <https://link.springer.com/article/10.1007/BF02478259>.
- [10] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML)* (2010). URL: <https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf>.
- [11] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 779–788. URL: <https://arxiv.org/abs/1506.02640>.
- [12] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: <http://arxiv.org/abs/1506.01497>.
- [13] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536. URL: <https://doi.org/10.1038/323533a0>.

- [14] Igor Garcia Ballhausen Sampaio et al. “A Novel Method for Object Detection Using Deep Learning and CAD Models”. In: *arXiv preprint arXiv:2102.06729* (2021). URL: <https://arxiv.org/abs/2102.06729>.
- [15] Saining Xie and Zhuowen Tu. “Holistically-nested edge detection”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1395–1403.
- [16] Yaping Zhang et al. “Active contour-based Canny edge detection framework for medical image segmentation”. In: *Computers in Biology and Medicine* 62 (2015), pp. 80–89. URL: <https://doi.org/10.1016/j.combiomed.2015.04.019>.
- [17] Zaiwei Zhang et al. “H3DNet: 3D Object Detection Using Hybrid Geometric Primitives”. In: *arXiv preprint arXiv:2006.05682* (2020). URL: <https://arxiv.org/abs/2006.05682>.