Master's Thesis

Universität Bremen

From Simulation to Segmentation: Training Domestic Robots to Understand Liquid Fill Levels Using Deep Learning

Antonius David
5036317
Faculty 04
Production Engineering,
Mechanical Engineering
and Process Engineering

29.05.2025

Tutor:	Examiners:
Arne Hasselbring	Prof. Dr. Udo Frese
	Dr. Rene Weller

Abstract

This thesis presents a framework for analyzing the fill heights of containers like: cups, glasses and bottles. The scene is set in an environment, that is shared with humans. Here a potential robot, that for this thesis is simplified by a camera, analyzes the containers. The framework consists of the necessary synthetic dataset creation pipeline as well as the achitecture and training of *neural networks*. The *neural networks* are setup in stages, where the first stage detects the containers from various angles. The detected area is then the input for the second stage. Here the fill height is predicted.
The *neural networks* were trained with a specifically for this task created *COCOA*-styled dataset. To test the models a test dataset was created with real images. The used labels were: visible and amodal masks of the container and liquid as well as the fill height and the transparency values.
On the validation dataset the average fill height error was: 12 percent. The framework achieved on the test dataset only a average fill height error, that is 3 percent better then a fixed prediction of 0.5. That the frameworks performance does not translate to the test dataset could be caused by reality gap, that is to large between the synthetic dataset and the real dataset.

Contents

List of Figures

List of Tables

List of Abbreviations

CNN	Convolutional Neural Networks
COCO	Common Objects in Context
FOV	Field of View
HDRI	High Dynamic Range Image
IoR	Index of Refraction
IP	Identity Path
ML	Machine Learning
NN	Neural Network
PCA	Principal Component Analysis
RLE	Run Lenght Encoding

1 Introduction

This section provides an overview of this thesis. Included are the introduction of the main problem, as well as a brief recap why this problem is relevant in the current time. Furthermore the scope of this thesis is described and the general structure is elaborated.

1.1 Problem Statement and Objectives

This thesis revolves around topics related to the field of robotics as well as computer vision. The problem is based on domestic robots and their abilities and limitations. Domestic robots are robots that operate in an environment shared and influenced by humans.

For this work, a domestic robot is defined as a robot designed to operate in a environment shared with humans, where it is supposed to perform tasks similar to those a human would carry out. In general, these tasks involve moving rigid objects. For this thesis, the focus is on liquid-filled containers, such as cups, glasses, and bottles, which frequently appear in kitchen environments.

Spilling liquids is not a desirable behavior, in for example, a kitchen environment. Therefore, the main goal is to provide the domestic robot with an understanding of liquid fill levels for the specified containers. The robot should be able to reason about the following aspects:

1. The liquid fill level of a transparent container, expressed as a percentage.
2. The liquid fill level of a non-transparent container, expressed as a percentage.

The liquid can be either transparent or non-transparent.

It is not in the scope of this thesis, that an actual robot is going to move liquid filled containers inside an environment. The robot is for this thesis simplified as a camera.
The focus is directed at in providing the general understanding of fill heights. Further reasoning and drawing action out of that, is not in the scope of this thesis.

For this work, a camera is positioned at a height similar to that of a human head. The camera looks down at a table, on which a container can be placed. Through the camera, the hypothetical robot can spot the container and measure its fill height.

The camera does not use measured depth information, nor does it utilize calculated depth information. Instead, the perception relies solely on the basic RGB image produced by the camera.

To analyze the image captured by the camera, a neural network is used. The creation of the dataset required to train the neural network is part of this thesis. Also a key part of this thesis is the creation and training of the neural network for this specific task.

1.2 Structure of the Thesis

Chapter 1 begins with an overview of the problems and objectives.
In Chapter 2, the necessary background for machine learning is given, providing insights about: *neural nets* and the general training approach. Furthermore, more detailed architectural components are introduced. The chapter also provides an overview of segmentation and the meaning of synthetic data for machine learning. The chapter closes with the introduction of the *COCO* dataset and a survey of related works.

Chapter 3 introduces the software and tools that were used in this thesis.

Chapter 4 generally states what was done in the scope of this thesis. It begins with an explanation of the chosen approach. The creation of the real and synthetic datasets is explained, as well as what types of augmentation were used to further diversify the dataset and prepare it for the training of the *neural nets.* Chapter 4 closes with a comparison of the real and synthetic datasets and an overview of the design of the *neural nets.*

In Chapter 5, the results of the training are stated, and in Chapter 6, they are further discussed. The thesis closes with a conclusion and a proposal for future work.

2 Background and Related Work

After presenting the problem and objectives of this work, as well as the structure of this thesis, the following section will focus on the theoretical background to establish fundamental knowledge about *neural networks* and synthetic data. Furthermore, an overview of related work is provided by presenting examples.

2.1 Overview on Machine Learning

2.1.1 Machine Learning

In machine learning *(ML)*, the objective is to create a model that helps solve a task. A model is generally a function that maps input data to output data. For this thesis, the focus is on supervised machine learning. In supervised machine learning, it is necessary to have labeled data relevant to the task the model is intended to solve [11].

For example, if someone wants to predict the electrical power usage of a building, they would need historical data from that building. Useful data could include information such as the time, the surrounding temperature, and the number of people, as well as the electrical power usage. It can be expected that there is a correlation between these three variables. In this case, the electrical power usage would be the target label the model must predict. A model could learn the relationship between these three variables and the target label, allowing it to predict future power usage. Less useful variables might include the age of the inhabitants or the number of nearby supermarkets.
These variables are shown in the following table:

Time	Temperature	Nb. of People	Power Usage
7	11	5	50
10	14	24	250
13	14	35	350
16	15	24	240
18	14	8	70

Table 1: Example Dataset

In machine learning the columns are called features. The variable that is going to be predicted is called label or target variable. One line of the table is called a sample. The sample consist of features and label. One value from a sample is called a datum. This also relates to image data. The dataset consists of images as a feature. Each image is a sample and each pixel is a datum. The mask that is going to be predicted is the label.

It is also important for the model to be accurate, and therefore, it needs to be able to handle complex relationships. A linear model would not be sufficient for the example above. Following this scenario and adding more and more people into the building, it is very unlikely that energy consumption would simply increase linearly. At some point, every light would be turned on, and every power outlet would be in use. Therefore, it is important for the model to capture non-linear behavior.

ML algorithms generally solve tasks such as regression, classification or segmentation. In regression, the task is similar to the example explained earlier: the model is trained to predict a numerical value. In a classification task, the model should predict the category to which the input belongs. Classification can involve only two classes or multiple classes, as a probability distribution. For example, an image might be classified as 80 percent tiger and 15 percent fish. Segmentation can be seen as a different form of classification. Segmentation works on images and classifies each pixel to a specific label. The chapter: *Segmentation* will put a deeper focus on that topic, since it is very important for this thesis. Other possible tasks of *ML* include transcription, translation, synthesis, and many more [6].

There are a number of algorithms that help build such models to solve tasks by learning from examples. In computer vision, the task is to extract useful information from images and further use it to solve specific problems. The image pixels are the domain from which the information is gathered. Useful information derived from images is often referred to as *features* in the context of computer vision. Such features could include edge shapes, color gradients, or more complex patterns [13].

One method developed in the 1990s is Principal Component Analysis *(PCA). PCA* was successfully applied to the complex problem of face classification. The original publication uses a dataset of face images to learn the mean image and the eigenvectors. Therefore, new images can be decomposed into their eigenvectors. By comparing the eigenvectors, it is possible to classify the image as a face image or not [18].

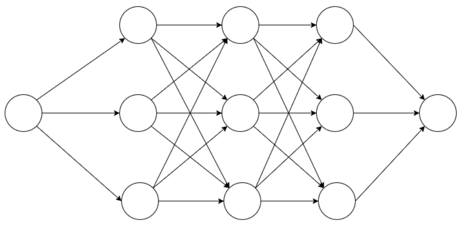
Further advancements in the field of computer vision were made with the development of methods such as: Support Vector Machines, Random Forests and Bayesian Models. Each of these methods had their own way of extracting features, as well as their own advantages and disadvantages. The rise of *neural networks* in the field of computer vision was connected to more capable computational hardware [13].

2.1.2 Neural Nets

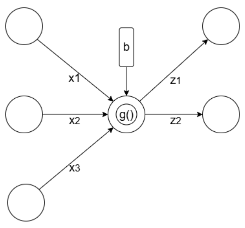
A *neural nets (NN)* consists of many neurons. Each neuron is a linear function with inputs, weights, and a bias, wrapped in a non-linear function to produce its output. The neurons are organized in layered structures and are connected. Neurons in a layer that are connected to all the neurons of the previous and next layer are particularly effective at sharing information. *NN*s with such a connection pattern are called fully connected neural networks. The form of the connections can be designed differently to achieve various results. Information inside the network generally flows from the input side to the output side. The layer structure can be further specialized, especially for larger *NN*s.
The formula for one neuron looks like the following:

$$z = g(\sum_i x_i \cdot w_i + b)$$

Here z is the output vector of the neuron. The input vector x is multiplied with weight vector w. In the end the bias b is added. The linear function is wrapped



(a) Full Connected Network



(b) Full Connected Neuron

Figure 1: Visualization Fully Connected Network

in a non-linear function to increase the models capability. During the training the weights are adjusted. Due to the architecture of the *fully connected* layer, the number of weights can grow fast.

Larger *NN*s can consist of a so-called backbone, a neck, and a head, or even multiple heads. The backbone generally holds the layers that extract general features from the input. For images, these features can range from simple shapes and color gradients to more complex edges and patterns. In practice, the backbone is often pre-trained on other, but similar, data [13].

This yields an advantage, for example, in computer vision, as general image features, such as simple shapes and color gradients, do not need to be learned from scratch again. The neck of a neural network is specifically responsible for learning the features of the specific dataset. The features learned in the neck and head build upon those from the backbone [13].

The head prepares the output of the neural network by shaping the learned features into the appropriate form. As mentioned before, it is also possible to add multiple different heads to the end of the *NN*. This is useful for learning different tasks at the same time, such as regression and classification or different types of regression. A representation of that can be seen in Figure: 2.

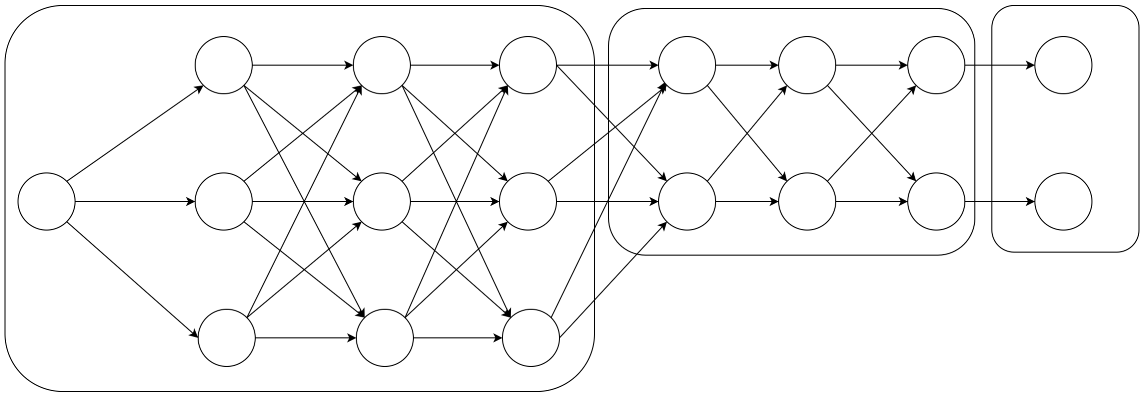
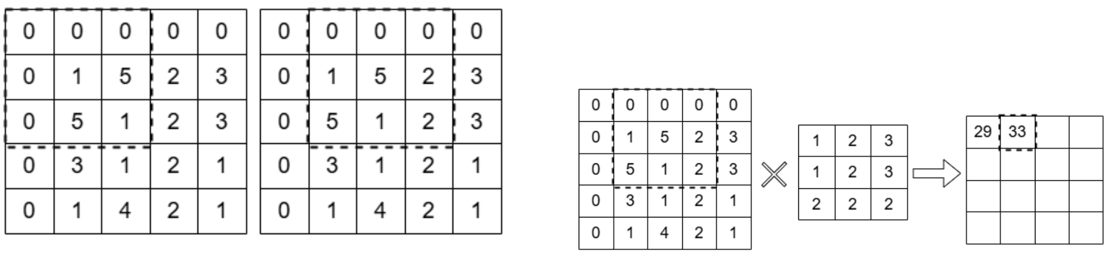


Figure 2: Schematic Network Architecture with Backbone, Neck and Head

2.1.3 Convolutional Neural Nets

This section start with a small introduction into *convolution* layers. First the *depthwise convolution* layer is introduced, followed by the general *convolution* layer.
Convolutional Neural Networks (CNN) use the same network architecture as the one described before. The difference lies in the structure and function of the neurons. The purpose of a *CNN* is to extract useful information from an image. Image data has quite different properties compared to the data from the example before [13].

Image data is, first of all, multidimensional, and the proximity between pixels is relevant. In other words, the spatial position of each pixel inside an image is important. Think of an image of a tree full of leaves—the leaves would form a cluster of green-valued pixels. With a normal fully connected layer, this clustered information would not be passed down to the next layer. However, to identify the tree, that cluster would be a strong hint.
This is the reason why neurons of a *CNN* must have a sense of spatial dimensionality [13].



(a) Depthwise Convolution Structure (b) Depthwise Convolution calculation

Figure 3: Visualization Depthwise Convolution Calculation

Figure: 3a shows the basic functionality of a neuron. Figure 3b shows the calculation that happens while the filter slides over the pixel.
The *depthwise convolution* layer consists of a sliding window (the kernel or filter) that moves over the input image, as can be seen in Figure: 3a. The output is gathered by calculating the weighted sum between the kernel and the values currently beneath the sliding window. To that sum a bias is added and similar to the fully connected neuron, the result is passed through a non-linear function. The result is again multi dimensional. The formula looks like the following:

$$z_{i,j} = \gamma(b + \sum_{l=0}^{k_H-1} \sum_{m=0}^{k_W-1} w_{l,m} \cdot x_{i+l,j+m}) \tag{1}$$

In that equation $z_{i,j}$ is the response. The variables k_H and k_W represent the size of the kernel and $x_{i+l,j+m}$ represents the input value.

Compared to a *fully connected* layer the number of weights of the *convolution* layer only depends in the size of the filter and the number of channles. The equation for the number of weights is $N_w = ((k_H \cdot k_W) - 1)C$. Here C is the number of input channels. For a *depthwise convolution* layer the number of input channels is always the number of output channels.

The strength and the weakness of the *depthwise convolution* layer is, that its not using information across the input channnles (dimensions). The general *convolution* layer does exactly that. The downside on the gained intelligence is the rising number of weights. For a *convolution* layer the number of weights is calculated like this: $N_w = CC'(k_H * k_W) + C'$. Here C stands again for the number of input channels, C' stands for the number of output channels. Here also the sum over the channels is calculated. The formula to calculate the output looks like the following:

$$z_{i,j} = \gamma(b + \sum_{l=0}^{k_H-1} \sum_{m=0}^{k_W-1} \sum_{n=0}^{D-1} w_{l,m,n} \cdot x_{i+l,j+m,n}) \tag{2}$$

Here now the number of output channels is free to choose. Also the sliding kernel window does not have to stop at each position. It can also calculate an output for the n-th next step. This parameter is called *stride*. Another important parameter is the *pad* parameter. When calculating the output with a kernel size bigger then one, the output dimension will reduce. This can be avoided by the *pad* parameter.

2.1.4 Training Procedure

After presenting what *NN* are, this chapter shall shortly introduce important aspects of the training of a *NN*. As described at the beginning of this chapter, *NN* need data to learn from. The data is split in a *training dataset*, *validation dataset* and a *test dataset*. The *training* and *validation* dataset can be separated in even smaller batches, to improve the training. At the beginning of the the training process the *layers weights* of the *NN* are initialized with random values. The objective of the training is to change the *layers weights* so that the prediction of the *NN* is optimal on the training dataset. The training process is a repeating of the following three steps:

1. pass batches of data through the *NN*,
2. observe the outcome of the *NN* and calculate the *loss*,
3. changing the *layers parameters* to minimize the *loss (backpropagation)*.

The *loss* is the difference between the predicted output of the *NN* and the the *ground truth*. It is used to evaluate how good the *layers parameters* already are. If the *loss* is still very high, then the *layers parameters* need to further be improved. One pass of the whole *training dataset* through the *NN* is called *epoch*. To improve the outcome, the loss is propagated backwards to the *layers weights* and they are changed accordingly. For that the gradient of the *loss* is calculated.

The calculated changes for each parameter are often multiplied by a *learning rate*, to influence the learning behavior. A smaller *learning rate* makes sure, that a good minima is found. The downside is, that reaching the minima might take longer. A large *learning rate* learns faster, but might never be able to find a good minima. It is also possible, to change the *learning rate* over the training time. It is common to start with a larger *learning rate* and slowly decay it to a smaller *learning rate.* It is also possible to get stuck on a local minima. For that, the idea of physical momentum can be added to the loss, so that small local minima can be escaped to find a even better minima. Another idea would be, to change for each *layer parameter*, the *learning rate* differently. This is the main idea behind *ADAM* optimizer, that is popular in computer vision [13]. Other optimizers are not considered in this chapter, as *ADAM* can be seen as state of the art.

Other challenges of the training process are *over*- or *under fitting.* A *NN* is *over fitting* when it just learns all the data points from memory, without generalization. This makes the model very good for the training dataset but not for real world applications with new datapoint. If the model is *under fitting* it doesn't really learn all the connections between the data points. It generalizes to much. In between these two extremes lies a good model that understands the data well. *Over fitting* and *under fitting* can be influenced by the number of the training data that is feet through the *NN* as well as by the size of the *NN.* If training data is sparse and the *NN* very large, *over fitting* happens fast.

Over fitting can be seen while training, when the *training loss* is still declining. but the *validation loss* is not following. Here it would be a good idea to stop the training when this happens, to prohibit *over fitting.* Another method to prohibit *over fitting* is regularization.

2.1.5 Regularization

Regularization is the generic term for methods used in machine learning to prevent the model from *over fitting.* This chapter introduces some of them, that are relevant for this thesis. Relevant are: *L1- and L2- Regularization, Batchnormalisation* and *Dropout.*

L1- and L2-Regularization tries to achieve a more stable model, by influencing the growth or the distribution of the weights that are learned during the training process. This is done by adding a regularization term, called *R(P)* to the loss, that is a function of the model weights them selves. *L1- and L2-Regularization* differ in the regularization term they use. *L1-Regularization* uses the absolut sum over all weights, multiplied by a factor, as the regularization term. The multiplication factor is a small value that shrinks the sum to a size, where it does not overwrites the actually loss, returned from the loss function.
The *L1-Regularization* has the effect that the network tends to ignore less important features, like for example data noise.
In contrast, the *L2-Regularization* term corresponds to the sum of the squared weight values, multiplied by a factor. This term punishes larger weights and keeps the weight distribution homogeneously. Homogeneous distributed weighs have positive impact on learning [13].

$$\mathcal{L}(y, y^{\text{true}}) + \lambda R(P) \tag{3}$$

Batchnormalisation (BN) has a different approach in preventing *over fitting*. *BN* affects the current output of the *non-linear* layers. Comparable to the input of a *NN*, that works best when it is normalized, *BN* normalizes each intermediate result. Normalization means to subtract the mean and divide by the standard deviation. This helps to keep the learning process more controlled, because the values of the features, tend to be in a more similar value range. *BN* also includes learnable parameters for scaling and shifting the value distribution. A drawback of *BN* is, that when the model is used in inference, the *BN* algorithm normalizes the intermediate results with the mean and standard deviation, it learned during the training phase.

Dropout works with *fully connected (FC)* layers. The *dropout* can be placed between *FC* layers. Here it randomly deactivates some neurons of the previous layers. The number of deactivated neurons can be specified. This pushes the *neural net* to learn more robust features [13].

2.1.6 Loss Functions

This section introduces loss functions that are relevant for this thesis.
The loss is a function from the predicted value of the *neural net* and the corresponding groundtruth value. The loss is the starting point for the optimizer, where the weights of the layer are optimized. Therefor it is important to use a meaningful loss function. The loss function depends on the task that the *neural net* has to solve.
For example, a *neural net* that tries to solve a regression task, can use the *mean absolut error (mae)* or the *mean squared error (mse)*. Similar to the *L1- and L2 Regularization* the choice of the loss function has an influence on the training result. The *mse* punished worse results more then the *mae*. The *mae* can be directly interpreted as the mean difference in percent.

For segmentation tasks where localization of the labels counts loss functions like *mae* or *mse* are not sufficient. For this task the *Dice-Loss*, among others can be chosen as a loss function. The *Dice-Loss* calculates how well a predicted mask fits to the label mask. The equation looks like the following:

$$Dice(A, B) = \frac{2(A \cap B)}{|A| + |B|} \tag{4}$$

The numerator corresponds to the correctly labeled pixels and the denominator to the total number of predicted and labeled pixels. The *Dice-Loss* is not influenced by class inequalities.

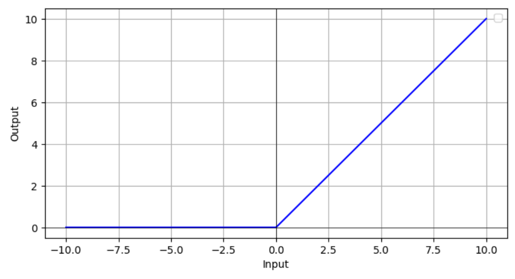
The *Binary Focal Crossentropy Loss (BFC-Loss)* can be used instead of the *Binary Crossentropy Loss (BCL)* when the classes have a high inequality. The equation for the *BFC-Loss* looks like follows:

$$L(p_t) = -\alpha(1 - p_t)^{\gamma} \log(pt), \quad \text{with } pt = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise} \end{cases} \tag{5}$$

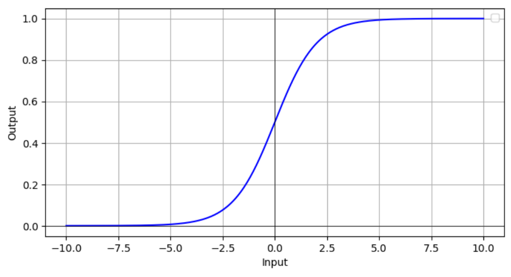
Here p is the predicted binary class value, and pt is a function of y and p. α is a weighing factor for class imbalance, γ is the focussing parameter and y is the binary label value. The *BFC-Loss* focusses on difficult to learn values, that are in a imbalanced segmentation tasks the labeled pixel.

2.1.7 Activation Functions and Initialization

Tis section focusses on activation functions that are relevant for this thesis. Also the weight initialization of the layers will be a topic, whereas they depend on the chosen activation function. The Figure: 4 shows the *Relu* and *Sigmoid* activation function.



(a) Relu Activation Function



(b) Sigmoid Activation Function

Figure 4: Activation Functions

The activation function gives the neuron its non-linearity. The activation function also puts the output values of the *neural net* into the necessary value range. For a binary classification this would be 0 and 1. For a fill height and a transparency, this would also be values from 0 to 1.

Layers that feature a *Sigmoid* activation functions are initialized using the *Glorot Initialization*. Here the initial kernel weights are drawn from a normal distribution with zero mean and a standard deviation of 0.01. These weights are then multiplied by a function of the number of input features. The equation looks like the following:

$$w = w * \cdot \sqrt{\frac{1}{m^{l-1}}} \tag{6}$$

Here $w*$ is the sampled normal distribution and m^{l-1} is the number of input features for the exact layer.
Layer that use the *Relu* activation function should be initialized with *He Initialization*. The equation for that looks like this:

$$w = w * \cdot \sqrt{\frac{2}{m^{l-1}}} \tag{7}$$

Here $w*$ is again a weight distribution, sampled from a normal distribution and m^{l-1} is the number of input features for that layer.

2.1.8 Advanced Architectures

This section gives a small introduction to the *ResNet* and *U-Net* architecture.
The *ResNet (Residual Network)* was introduced in 2016 and its purpose was to be used as a classifier. The motivation to develop the network came from the observation that the performance of a *neural network* did not improve linearly by just stacking more and more layers together.
By naively adding more and more layers, the accuracy could actually decrease again.

Also vanishing gradients are an issue. The *ResNet* network gives a alternative approach to that. It introduces *identity paths (IP)*. Those *identity paths* are build parallel to the *non-linear layers (convolution etc.)* and simply pass through the previous result. The *identity path* and the *non-linear path* are then added together. Sometimes the *IP* has a *convolution layer* to adjust for channel number changes.
This approach makes it in a way learnable for the *neural net* how many layers it needs. Because by adjusting the weights of the *non-linear path* the whole *non-linear path* could be turned of. This would then just forward the *IP* that adds no further transformation to the data. A schematic a *ResNet* layer can be seen in Figure: 5 [8, 13].

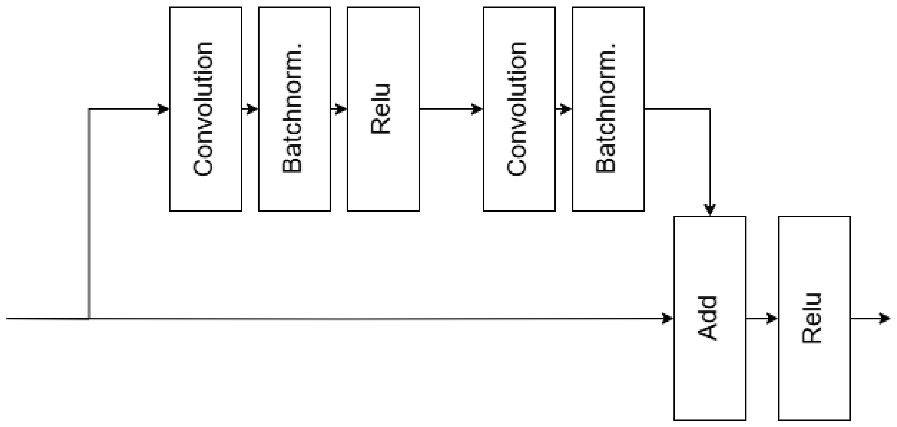


Figure 5: Schematic ResNet Layer

The *U-Net* architecture is used for segmentation tasks and has a symmetrically encoder-decoder structure. It was introduced in 2015. The structure can be seen in Figure: 6.

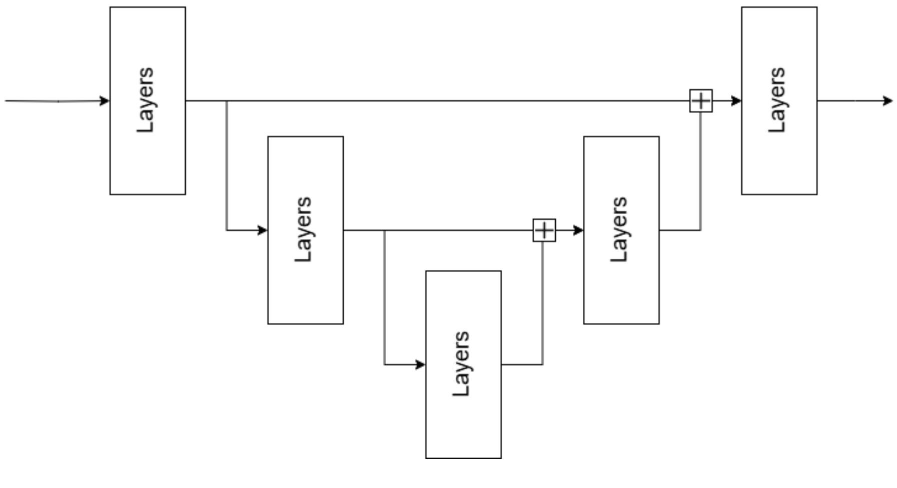


Figure 6: Schematic U-Net Network

The *U-Net* also uses skip connections, but as can be seen, it uses them in a more global way, by connecting encoder stages with decoder stages. This gives the advantage, for a later decoder layer, to be able to access more accurate local information. [13, 15]

2.2 Segmentation

The segmentation task returns a pixel wise classification mask of an image from the *NN*. The mask has often the same shape like the input image. For an image with a resolution of: 1280 times 800 pixel, the resulting mask has also the the shape: 1280 times 800.

It is a more advanced type of detecting, because it delivers locally correct masks for the classes. There are different types of *segmentation*, with different use cases. The most simple form of *segmentation* is *object segmentation.* For example, here the *NN* returns for an image of a parking lot all pixel that seemed to be a car, labeled as cars. This could for example, be quite helpful to estimate how full the carpark is. A more advanced method of segmentation is *instance segmentation.* In this case the *NN* would return a mask where each instance of a car is a separate label. This is a more complex task, whereas the *NN* needs to decide which pixels belong to which instance. With that additional information it would be possible to count the number of cars in the parking. *Instance segmentation* is also important for topics like autonomous driving or in medical imagery. [13]

Two examples for *NN* that were proven to be good at that task are the *YOLO*-architecture and the *MASK-R-CNN.* Both are state of the art. A short introduction is given on how they work. Both rely on a feature extractor as a backbone, plus output layers with different output heads. For *YOLO* and *MASK-R-CNN* it is not possible, to train the segmentation only with the mask as a label. They need to predict helpful other features to be able to build a good segmentation mask. Those extra features that are predicted are for example *bounding boxes* of the interesting objects or *regions of interest.* Another difference between these two is, that *YOLO* analyzes each pixel only once, *MASK-R-CNN* analyzes twice. This has the advantage that *YOLO* is generally a faster *NN*, but less accurate. [13] [7]

For this thesis *amodal segmentation* will be from special importance. The advantages of *amodal segmentation* is that it also detects or estimates where the occluded parts of on objects ends. This is an even harder task to solve, because the *NN* need to somehow have an understanding of how things are positioned in the world and what size they are - without actually seeing everything. The research on *amodal segmentation* is still at its beginning. Where until now only a few publications on that topic exist.

The figures 7a - 7c give an example for each type of segmentation as described before.

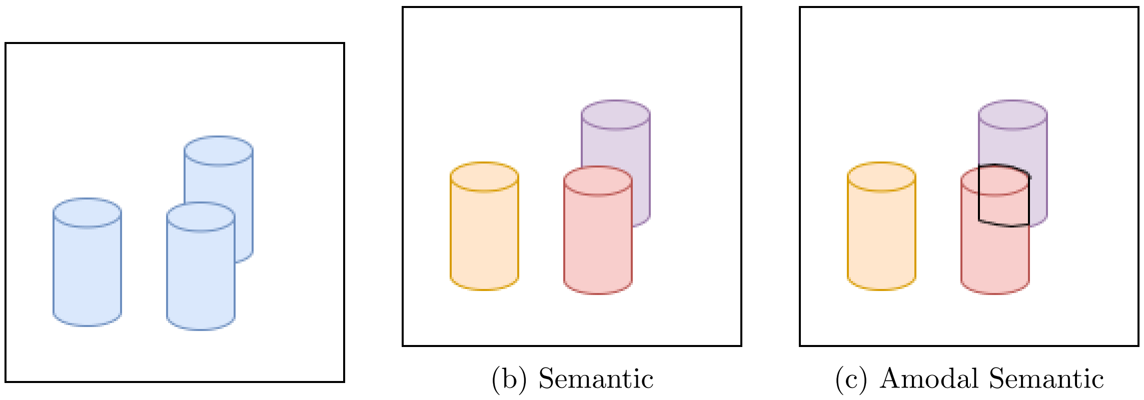

(a) Object Segmentation (b) Semantic Segmentation (c) Amodal Semantic Segmentation

Figure 7: Types of Segmentation

2.3 Synthetic Data and Computer Vision

The need for *syntethic data* arises from the difficulties, that it takes to create a real dataset. Some of these difficulties are, that it is time consuming and prone to errors as well as sometimes just not doable.
For example, in producing industries. To detect defect parts, the company would need a lot of defect parts to train a classifier, what a defect part looks like and what a good one looks like. But it is obvious that the company doesn't want to have many defect parts in the first place.
Here *syntethic data* comes in play. Especially industries often have digital models of there products [13].

Synthetic data is produced from digitals models using a *render engine*. *Digital models* are represented by *meshes* consisting of *faces*, *edges* and *vertices*. *Vertices* are points in *3D coordinates*. *Edges* are two connected *vertices* and a *face* are multiple connected and ultimately closed *edges*. The more *faces* a model has, the more complex it can be, but the more computational power is needed [13].

Placing those digital models as well as a virtual camera and some lighting, will create a basic *render scene*. The *render engine* will then calculate what is visible for the virtual camera using different coordinate frames and transformation matrices.
Rendering a realistic scene is more complex though. For that realistic lighting, and coloring is needed, also including reflection, absorption and other optical effects [13].
It holds generally that the more detail is captured inside a model or a scene by adopting as much properties as possible, the more realistic it will appear.

Until now there are still no perfect *render engines* available. So that there will be a gap in quality between *real images* and *synthetically rendered images*. This gap is commonly known as the *reality gap*. With more accurate *render engines* this gap will shrink.
In the past a technique called *domain randomization* was used to overcome the *reality gap*. The general idea was to include the reality in a wide variety of lighting and texture conditions [4, 13].

2.4 COCO and COCOA Dataset

The *Common Objects in Context COCO* dataset is widely used for training and testing *segmentation* tasks in computer vision. Therefore a small introduction is given to explain the dataset and its underlying structure, to later explain the *amodal COCO* dataset [10].

The *COCO* dataset was introduced in 2015. It was the first large dataset focussing on instance segmentation. Although it contains the information for bounding boxes as well. The images were chosen to reflect objects in contextual relations and from a variety of view angels. It contains 91 object categories with 2.5 million labeled instances on 328 thousand images [10].

A *semantic segmentation label* from the *COCCO* dataset contains an *id* and an *image id* as well as a *category id.* It also contains the *segmentation* area as either a polygon or run-length encoding *(RLE).* The polygon is a list of vertex coordinates that outline the objects boundary. The *RLE* is a compressed representation of consecutive pixels, that represent the binary mask. The *semantic segmentation label* also contains the *area* of the mask, as well as the *bounding box.* The last parameter is, if the object *iscrowd.* For example, for a stadium full of people, not each person would be outlined but the crowed would be. This would be labeled as *iscrowd = 0.* The *category id* points out to which category the *label* belongs. Each category has an *id*, a *name* and a *supercategory* [10].

The *COCOA* dataset follows in a broad sense the structure of the *COCO* dataset. It was first developed in the year 2017 by Y. Zhu et al. The *COCOA* dataset was based on 5000 images of the *COCO* dataset. As explained before *amodal segmentation* also labels the occluded parts of objects. Thats why the label structure changes slightly. A *COCOA* dataset annotation for one image contains a list of *segmentation masks* for each labeled object. It also contains a list of names for each object as well as a list of *area* in pixels for each object. It also contains a list for each object, if it *is stuff.* For each labeled object, also the *occlusion rate* is given. To distinguish between visible and inviable part, for those objects that have a occlusion rate, the *visible mask* and *invisible mask* is given. For not occluded objects a *None* is used as a place holder. There is also a *order* given that maps the names to numbers. These numbers are then used under *depth constraint* to indicate which objects occludes which. The last two features are the *image id* that points to the original image and the *size*, that indicates the max number of labeled objects in the given image [20].

2.5 Related Work

There is a handful of research already done in the field of bottle fill level detection, such as in K. J. Pithadiya et al. [12]. They proposed an algorithm that detects whether a bottle is underfilled or overfilled using edge detection techniques and distance measurement based on a camera and a proximity sensor. This solution was developed for an industrial environment with a fixed test setup utilizing a conveyor belt.

Also partly relying on edge detection is the proposed work from L. Yazdi et al. [19].

Their solution is again developed for an industrial environment with a conveyor belt and a camera. The objective is not only to estimate the fill level but also to classify whether the lid is closed correctly. For this, the edge detection algorithm is paired with neural network techniques. The algorithm is split into a feature extraction part as well as an image classification task.

The work of B. Chatchapol et al. [3] focuses on the image processing part solely using convolutional neural networks. Here, the images are classified pixel-wise using the encoder-decoder method, similar to semantic segmentation. The possible classes are: bottle with a cap, empty spaces in the bottle, unknown areas, areas of water, and background. This solution was again developed for industrial use with a fixed camera position and a conveyor belt.

Another solution was proposed by J. A. Syed et al. [16]. This approach is fully based on convolutional neural networks and again aims to measure the liquid fill height in a liquid container with a fixed camera setup. For the training of the CNN, they created a dataset with around 200 real images. They used the classes: cylinder, liquid surface, and liquid height. They used the YOLOv8 model as the CNN. The model returned the bounding box of each class. To further calculate the fill height, they first corrected the perspective distortion that results from the viewing angle and second used the geometry of the container. They did not use depth information from the camera. This work also focuses more on a fixed camera and liquid location.

G. Qiao used the YOLOv5 CNN as well. In their work, they predicted the water fill level of rivers by detecting the corresponding water gauge and reading the fill level information indirectly from the water gauge. They explained difficulties with complex surroundings and the use of a pre-made training dataset. They used a sequential detection approach by first identifying the water gauge and then finding the characters on the water gauge.

As can be seen, research in this area is plentiful, but it is often limited to a fixed camera position and specific liquid containers, as [3,12] focus on bottles on conveyor belts. J. A. Syed focuses on test tubes from a fixed position. For this thesis, the environment for the measuring task is more diverse, with different containers, surroundings, and camera positions.

3 Software and Tools

For the creation of the models, the resulting renderings of the scenes and the creation of the real dataset, the following software and tools were used.

3.1 Blender

Blender is a free and open-source software developed and maintained by the community and its contributors. It provides a comprehensive 3D pipeline, supporting tasks such as 3D modeling, rendering, rigging, and animation, among others.
Blender offers a realistic rendering approach using *ray tracing*, a technique that simulates the behavior of light by calculating its paths, reflections, and absorption. This method provides a highly accurate representation of real-world lighting [2].
Additionally, *Blender* supports *Python* scripting, enabling users to automate complex processes.

In *Blender*, a mesh consists of *vertices* (points in 3D space), *edges* (connections between two vertices), and *faces* (surfaces formed by multiple connected edges). By manipulating these fundamental building blocks, complex models can be created.
Modeling in *Blender* is primarily achieved through operations such as *extruding*, *scaling*, and *beveling* the building blocks of the mesh. Additionally, more advanced modeling techniques are available for complex shapes and structures.
An alternative approach to modeling in *Blender* is the use of *Geometry Nodes*, a node-based system for procedural geometry creation. This system allows users to build models through interconnected nodes, where each node performs specific operations on the geometry. *Geometry Nodes* are particularly useful for generating procedural patterns and structures.
A similar node-based architecture is also available in *Blender* for creating textures and materials. However, for this thesis, the modeling of meshes is primarily based on the traditional techniques mentioned earlier [17].

The material modeling for this thesis is based on *Material Nodes* in *Blender*. Materials in *Blender* define the appearance of objects in the rendered image and follow a node-based workflow, where different nodes are combined to achieve various material effects.
A commonly used *Material Node* in *Blender* is the *Principled BSDF Node*. This node controls the fundamental properties of a material, such as roughness, base color, transparency, metallic reflection, and emission (if the object emits light).
To create more detailed and realistic surfaces, normal maps can be connected to the normal input socket of the *Principled BSDF Node*. These normal values, often generated from other nodes, add surface detail by simulating small-scale variations in the material without altering the actual geometry [17].

Blender has multiple different *render engines* available. For this thesis, the *Cycles engine* is of interest. This *render engine* provides realistic lighting. For later usage, a small explanation is given here. A typical path for a light ray could look like the following: In *Blender*, the virtual light ray is emitted by the camera. The light ray then travels through space and eventually hits a surface. From that point, child rays—such as from a reflection or transmission—are generated. The virtual light ray completes its path when it hits a light source [17].

With the *Cycles render engine*, the virtual light rays that are emitted from the camera can have different types, depending on the path they take. For example, a ray that is emitted from the camera is a *camera ray*. When it hits a surface, new rays—depending on the surface properties—are created. These rays can be:

1. *reflection rays*, if the surface can produce reflections
2. *transmission rays*, if the object is not opaque
3. and *shadow rays*.

The *reflection* and *transmission rays* can again be further categorized into *diffuse* (when generated by a diffuse reflection or transmission), *glossy* (when generated by a glossy reflection or transmission), or *singular* (when generated by a sharp reflection or transmission) [17].

Blender allows setting numbers for the maximum number of bounces a ray can have through the environment. But it is also possible to set values for each of the named categories. The numbers set, has an influence on the appearance of the rendered image. For example, if the number of maximum bounces is set to zero, the image would only contain direct light [17].

3.2 Blenderproc

BlenderProc and its newer version, *BlenderProc2*, extend the capabilities of *Blender*. *BlenderProc* is primarily used to enhance rendering workflows by providing easier access to semantic segmentation and instance segmentation masks within *Blender*. Additionally, *BlenderProc* enables the output of depth images and normal maps, which are essential for various computer vision and machine learning applications.
BlenderProc2, the successor to *BlenderProc*, introduces a more user-friendly *Python* interface, making it easier to integrate into automated pipelines. Moreover, it allows users to save their work directly in standard dataset formats, such as *COCO* and *BOP*.
The *BlenderProc Python API* is closely integrated with the *Blender Python API*, ensuring seamless interaction between the two. In the following sections, *BlenderProc2* will be referred to as *BlenderProc*.

BlenderProc allows direct manipulation of objects within a scene, including changes to their textures, positions, and rotations. Additionally, *Geometry Nodes* can be configured to modify object structures procedurally. The lighting conditions in the scene can also be adjusted, enabling more dynamic and varied rendering setups.
Beyond basic modifications, *BlenderProc* provides advanced features, such as the ability to generate randomized room layouts automatically.
Camera parameters, which play a crucial role in rendering, can also be directly controlled through the *BlenderProc Python API*, allowing for precise adjustments to focal length, field of view, and positioning [14].

3.3 Tensorflow

TensorFlow is a *Python* library designed for developing *machine learning* solutions. It provides an end-to-end framework that encompasses data preprocessing, model construction, training monitoring, and performance evaluation. Additionally, *TensorFlow* supports the use of pre-trained models.
Neural networks and other *machine learning algorithms* can be implemented layer by layer using the *tensorflow.keras.layers* module, enabling flexible model development [1].

There are two ways to build models in *Tensorflow*. The first uses the *Sequential* function. Here new layers of the *Neural Net* can be just added after each other. This is a convenient method for smaller models. For larger models, with multiple paths this is not sufficient. For that Tensorflow also gives the possibility to use *python* variables for the layers [13].
This could look like the following:

```
layer1 = Conv2D()(input)
layer2 = Conv2D()(layer1)
y1 = Conv2D()(layer2)
y2 = Conv2D()(layer2)
y = y1 + y2
```

3.4 GIMP

GIMP, short for *GNU Image Manipulation Program*, is a free and open-source image editing software. It is available for *Linux*, *macOS* and *Windows* and is widely used by graphic designers, photographers, illustrators, and scientists. *GIMP* offers a variety of powerful tools for image manipulation.
In this thesis, *GIMP* was used to label ground truth images for constructing the test dataset, as further explained in Chapter: 4.2. The software allows for the creation of additional layers on top of loaded images, enabling manual annotations. These annotated layers can then be exported and utilized as ground truth data [5].

3.5 Intel RealSense D435i

The *Intel RealSense D435i* is a camera module developed by Intel, primarily used in the field of robotics. It features a left and right imager, a color sensor, and an infrared projector. This camera is capable of generating depth information and visualizing it as a point cloud. However, in this thesis, only the color sensor is utilized.
The camera's RGB color sensor captures images at a resolution of 1280×720 pixels and has a horizontal *field of view* (FOV) of 69° and a vertical FOV of 42°. It can also take videos with 30 Frames per second. The resolution of the sensor is 2MP [9].

4 Methodology

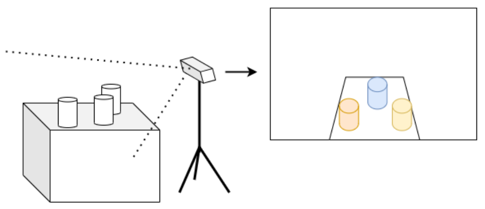
This section deals with explaining the main work that was done in the scope of this thesis. First of all an overview of the approach is given. The next sections deals with creating the necessary models and building the *COCOA*-dataset that functions as the initial dataset. Then a overview is given on how the initial dataset is altered to solve multiple tasks. This section also features statistics of the created datasets. The *neural networks* are presented at the end.

4.1 Overview of the Approach

This section shall explain the general method that was used to solve the task. As mentioned in the intrduction the hypothetical robot, represented by the camera stands in front of a table and looks down at it. On that table stands a container, like a glass, a cup or a bottle filled with liquid.

In the first stage, the camera then detects, enabled by a neural net, where the container is on the image. This corresponds to a segemntation task. In the second stage then, the detected area is send to a second neural net. The part that is send has a height of 300 pixel and a width of 252 pixel.

The second neural net then analyses the image that it got as an input. For this input image two *amodal*- masks are calculated, one for the container and one for the liquid. From that *amodal*-mask then the fill height and the transparency is derived. The main idea behind this approach was that the *neural net* uses the geometrical information that it learned by predicting the masks to improve the fill height prediction. The transparency output for the container and the liquid shall additionally give the *neural net* the understanding of which containers are opaque and which are transparent. In theory this shall help, because with transparent containers the fill height is much more visible then with opaque cups. Also opaque liquids should be better distinguishable then transparent ones. This task corresponds to a segmentation task as well as a regression task. This two staged approach is visualized in Figure: 8.


(a) Stage 1

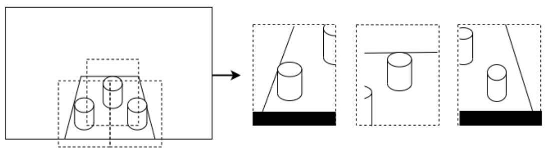

(b) Stage 2

Figure 8: Overview of the Approach

To train the two *neural nets* a *COCOA*-style dataset is created. Because the tasks, that the *neural nets* have to solve are so different, the creation of sub-datasets was necessary. Those sub-datasets are tailored for the specific needs.

The datasets consists mainly of images of scenes from tables. To be able to use a large number of images, most of the images used to train the *neural networks* was automatically created using *Blender* and *Blenderproc*. To be able to compare the solution to real world, also images of real table scenes were taken by hand.

4.2 Real Dataset Creation

As mentioned before, the creation of a real dataset was part of this thesis. For this, a total of 50 images were taken. The images were captured in batches of 10. Each batch of 10 images corresponds to a specific camera position. The viewing angle and camera height were changed in between. Different lighting conditions were also taken into account. Across the dataset, there is direct light from the background as well as from the side. Additionally, hard lighting on the table appears, as well as images taken only with indirect light.

The camera height varied between 1.7 meters and 1.4 meters. The table had a height of around 0.7 meters. At all times, the camera was in close proximity to the table. The table had a white surface, and random objects like boxes and paper were scattered across it to represent a realistic living environment.

To achieve variation in the scene, ten different kinds of cups were used, as well as two plastic bottles. The cups, glasses and bottles can be seen in Figure 9. Cup number one is a plastic cup with printing all around. The numbers 1 to 4 correspond to glasses of different sizes—two of them have a label. Numbers 5,6 and 8 are cups of different sizes, shapes, and colors. The bottle is made of plastic and was used with and without the colored batch.

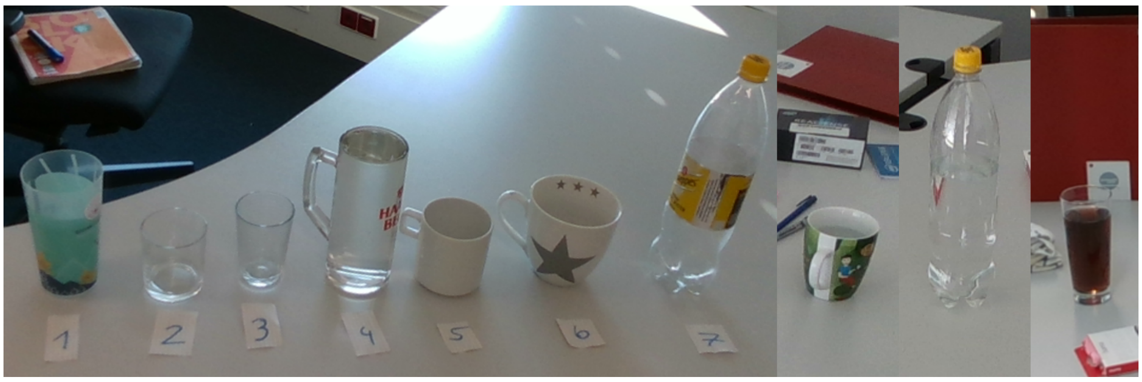
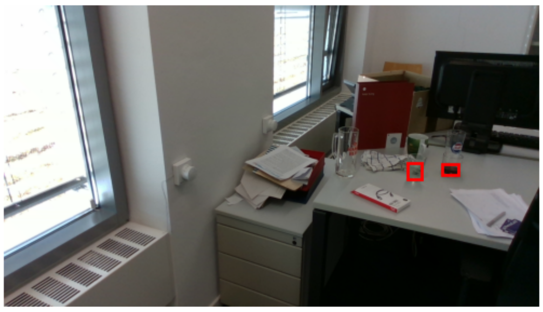


Figure 9: Variation of Containers

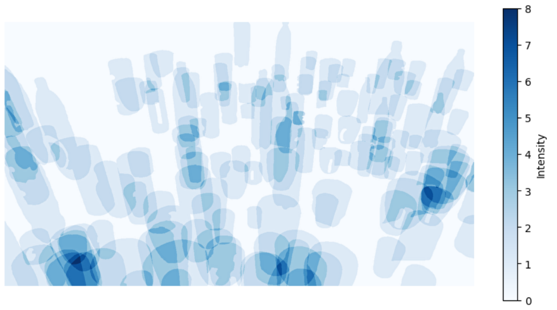
Water was used as the liquid. To achieve variation in the color, the water was dyed. The base colors were red, green, and yellow. By mixing them, it was possible to obtain six different colors: red, green, yellow, shades of turquoise, and brown. Clear water was also used. The color of the liquid changed with every batch of ten images, whereas clear water was present in every batch.

To achieve a homogeneous distribution of the cups and fill heights, each cup and bottle appeared exactly four times per ten-image batch. When it appeared, the fill height was either high, medium, low, or empty, or the liquid level was not visible. This represents all possible states of the cups and bottles that could be encountered.

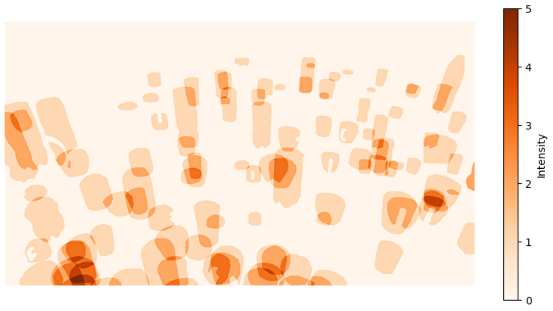
A total of 160 cups and bottles was labeled. With 50 images in total, that makes 3.2 cups and bottles per image on average. The number of labeled liquids is smaller, totaling 114, resulting in an average of 2.28 liquid labels per image. Figure 10 shows an example from the dataset with the bounding boxes marked. The handles were labeled as background because they do not directly indicate the liquid fill height.



(a) Sample from Datatset with Liquid Boundingbox



(b) All Cup Labels



(c) All Liquid Labels

Figure 10: Visualization Real Dataset

Also interesting is the number of pixels that represent the background compared to those that represent labels. For the cups and bottles, only around 2.5 percent of the pixels are labeled, and for the liquid labels, the percentage is even lower, at around 0.9 percent.

As can be seen in Figure 10c and Figure 10b, the labels are quite well distributed across the image.

The labels were created using the program GIMP by drawing by hand with an active pen. This method has the downside of inaccuracy. First, there is the inaccuracy of manually following lines, which can result in lines being much curvier or wiggly instead of straight. Additionally, the color of the object that should be labeled can blend with the background color. As a result, the person labeling must sometimes guess the shape of the object. The same issue applies to clear liquids inside a cup.

Another downside of labeling by hand is particularly evident when performing amodal labeling. The occluded part cannot be labeled correctly for a material that is not see-through.

These problems come in addition to the challenges of labeling real-world data, such as the time-consuming nature of the process and the difficulty of achieving sufficient variation.

4.3 Synthetic Dataset Creation

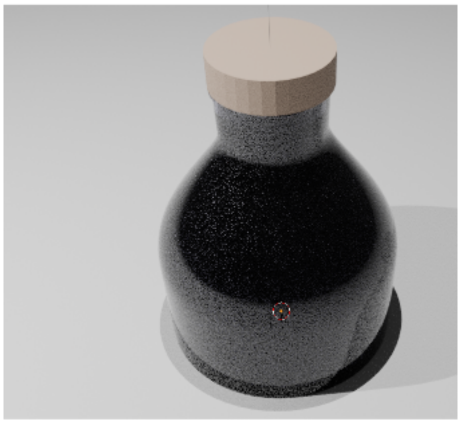
This section explains how the models are created step by step, following the structure of the written code. As explained before, most of the modeling was done using *Blender*'s own Python environment. In the end, an analysis of 150 models will be conducted. The average size, color, and fill height will be analyzed.

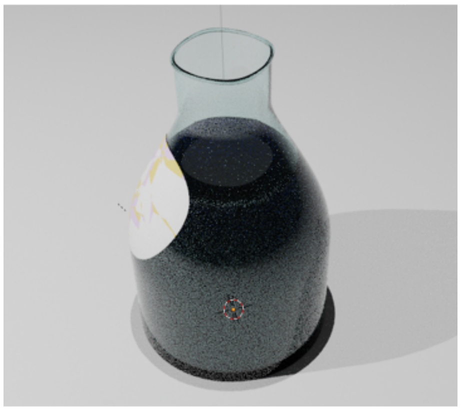
4.3.1 Generated Cups, Glasses and Bottles

With the written program, it is possible to create different types of containers, such as cups, glasses, and bottles. The modeling is done by following a scheme where the object is built from the ground up. This applies to every type of container. Figure 11 shows a few results of the program.
As can be seen, the first row displays bottles, the second row displays cups, and the third row displays glasses. The goal of creating cups, glasses, and bottles was to generate different shapes of regular objects that could be encountered in real life.

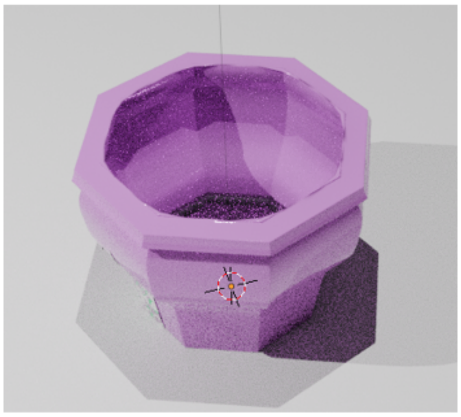
The program provides the option to add handles to cups and glasses, as well as labels to all three container types. Additionally, bottles can have a cap. These three add-ons are created separately and then joined together with the object.

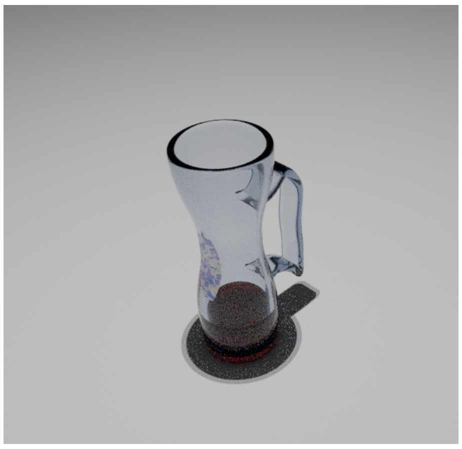
All of these parameters are controlled through a parameter dictionary, which is randomly generated for each container. The parameter dictionary includes various attributes, such as the height of the container, color, width, handle geometry, label creation, and fill height, to name a few.
In total, the parameter dictionary consists of 12 higher-level dictionary entries, including *general parameters*, *cup geometry parameters*, and *handle geometry parameters*, among others. These higher-level entries provide a more structured and readable organization.
Each higher-level dictionary entry contains multiple sub-entries that store the specific values for properties like those mentioned above. The specific values in the parameter dictionary varies for each container type to give them a realistic shape. Some parameter from the dictionary are specifically discussed in the further explanation. A complete example of a parameter dictionary can be found in the Appendix.


(a) Glass Bottle with Cap


(b) Glass Bottle without a Cap

(c) Full yellow Cup with a Handle


(d) Empty purple Cup without a Handle


(e) Green Glass with a Label and a Handle

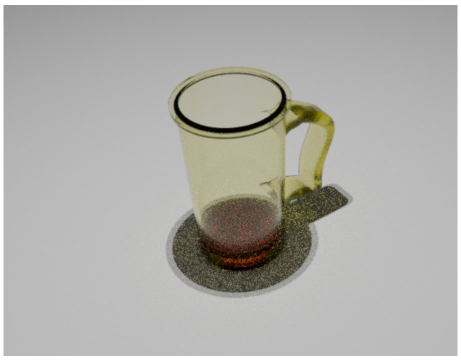

(f) Glass with and a Handle

Figure 11: Types of different Containers

The generation of cups, bottles, and glasses all begins with the creation of a basic mesh. This basic mesh is a primitive cylinder in *Blender*, placed in an empty workspace. Initially, the cylinder does not have closed ends and remains open on both sides.
The initial high of the cylinder is already influenced buy the type of the container. Cups are generally not as tall as glasses and bottles. For this thesis the hight for a cup can vary between 6cm and 12cm. Glasses can vary between 12cm for water glasses and 25cm for example for beer glasses. Bottles can vary around hight of 8 to 33cm.
As well as the hight, the diameter also depends on the type of container. For cups the diameter can vary between 6 and 10cm and for glasses and bottles between 8 and 15cm.

Next, the radius of the lower edge of the cylinder is scaled down to create a more cone-like shape. Following this, the lower edge is extruded inward to form the base of the container. To achieve the typical bottom shape—where the container rests more on the outer edge—the shape is then extruded upwards. Finally, the bottom side of the container is closed.
The steps of this process are illustrated in Figure 12.

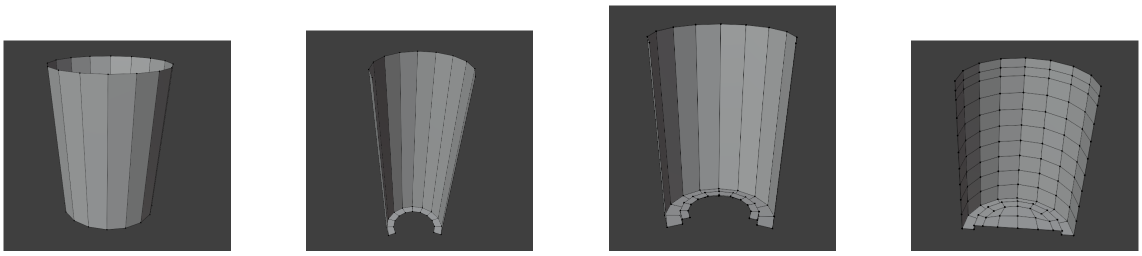

(a) Cone like Shape (b) Beginning of the Bottom (c) Typical Bottom Shape (d) Container is closed

Figure 12: Creation Basic Mesh

The second step in creating the containers is to add a lip to the upper end of the container. Many glasses, cups, and bottles have a slightly thicker rim where the lips or teeth could touch the glass. This feature is also modeled in this process.
To achieve this, the upper edge of the cone is extruded outward and then downward. The extent of this extrusion is randomly controlled by the parameter dictionary. This process is illustrated in Figure 7c.

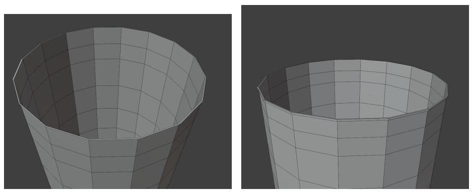
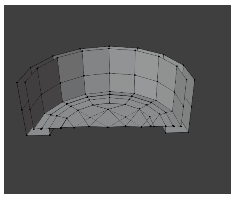

(a) Extruded outwards (b) Extruded down

Figure 13: Creation Lip

The next step in the creation process is to give the container a thickness and adjust

the bottom thickness. The overall thickness is controlled using a *solidifier*, which ensures that the current mesh has a uniform thickness throughout.
Most cups, glasses, and bottles have a thicker bottom compared to their sides. To account for this, the lower inside of the container is accessed. Then, the inner bottom is flattened and moved upwards, creating a realistic thicker base.
Finally, any corner exceeding 30 degrees is automatically smoothed. The steps can be seen in Figure 14.


(a) Inner Bottom is made even

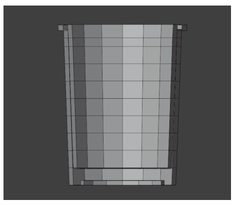
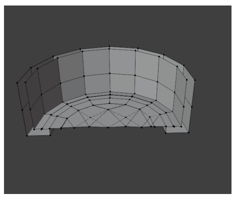

(b) Extruded down

Figure 14: Adjust Thickness

Next, *shader nodes* are assigned to the container, which define its texture. To create a complete texture in *Blender*, the properties of the *shader nodes* are controlled using the parameter dictionary.
This includes setting the base color, metallic value, surface roughness, index of refraction(*IoR*) , and transmission value. The *IoR* is a material property that describes how a light beam bends when it passes through the boundary between two different materials. For glass, a good approximation of the *IoR* is 1.5.

In *Blender*, all these properties can be controlled using the *Principled BSDF shader*. The base color determines the appearance of non-transparent textures, such as those used for cups. Since cups are typically opaque, their wall thickness does not affect their visible color.
However, for transparent objects like glasses and bottles, the perceived color depends on the material's thickness. To account for this, an additional *shader node*, the *Volume Absorption shader* in *Blender*, is used specifically for transparent materials. An example of each case can be seen in Figure 15.


(a) Inner Bottom is made even

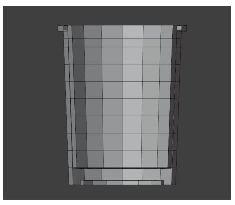

(b) extruded down

Figure 15: basic mesh thickness

This section focuses on describing the label. Each type of container can have a label, which is created by adding a primitive plane to the workspace. The plane has no thickness or additional structure; it is simply a highly subdivided mesh to allow for better wrapping around the container.

The graphics on the label are generated procedurally using only *shader nodes*. The goal was to create diverse patterns that include angular shapes, representing letters, and rounded shapes, simulating logos.

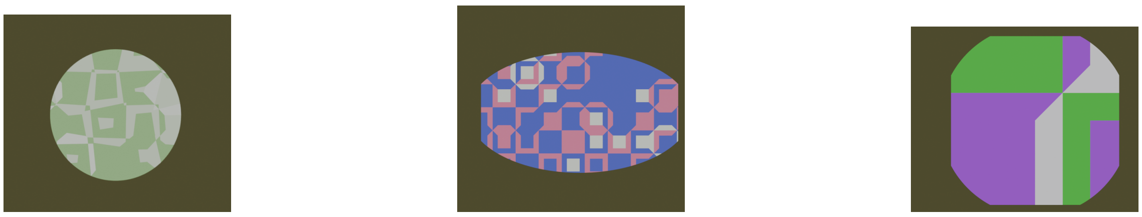
The *shader* architecture consists of two strings that are combined in a *principal BSDF shader*. One branch enters the *base color* input, and the other branch enters through the *alpha* input. The first branch controls the coloring of the label, while the second one creates the shape of the label by setting certain parts to invisible.

To control the coloring of the label, two *shader nodes* are used in the first branch: the *Color Ramp* and the *Voronoi Texture*. The *Color Ramp* maps values to colors and thus controls the label's coloring. The values that enter the *Color Ramp* come from the *Voronoi Texture*'s color output. The *Voronoi Texture* evaluates *Voronoi noise*. *Voronoi noise* was developed around 1996 and is often used to generate procedural textures.

Through the dictionary parameters, the *scale*, *detail*, *roughness*, and *random* parameters of the *Voronoi Texture* can be randomly adjusted. The *scale* parameter controls the size of the generated geometry. The *detail* parameter determines how many shapes are visible. The *roughness* parameter adjusts the transition between a smoother noise pattern and a sharper one. The *random* parameter influences how chaotic the alignment of the geometry is.

The color output of the *Voronoi Texture* is then remapped using the *Color Ramp* to match the color values set in the parameter dictionary.

As mentioned before, the second branch creates the shape of the label. This is necessary because the label mesh itself is a rectangle. Here, a *Color Ramp* is used again, but this time it maps values to alpha values, which are used by the *alpha input* of the *principal BSDF shader*. The input for the *Color Ramp* comes from a *Gradient Texture Node*. This node is set to *spherical* to create round structures. The shape of the round structure can be controlled by a *Mapping Node* and a *Texture Coordinate Node*. Adjusting the *scale* parameter of the *Mapping Node* changes the size of the round structure. If the size is increased to the point where it reaches the boundary of the mesh, the visible shape becomes rectangular. This allows the label to take on round, elliptical, or rectangular shapes.
The following image shows different label creations.



(a) Round Label

(b) Elliptical Label

(c) Rectangular Label

Figure 16: Label Variation

The label is then added to the container structure. This is done using a *Lattice* and a *Lattice Modifier*. The *Lattice* is a special object in *Blender* that can be attached to an existing mesh object. If the *Lattice* is then moved or deformed, the attached object will move or deform in the same way. This functionality is used to stick the label to the container by first attaching the created label to the *Lattice* and then adjusting the *Lattice* to follow the shape of the container. This ensures that the label also conforms to the container's shape.

The next part focuses on how the handle is created. As mentioned before, only glasses and cups have a handle. The creation of the handle starts by adding a primitive *Bezier Curve* to the workspace.

A *Bezier Curve* consists of multiple *control points*. Each *control point* has two *handles*. A curve is formed between two *control points*. By adjusting the number of *control points* and the orientation of the *handles*, it is possible to create different kinds of curves. For the handle, four *control points* are used. The *control points* at each end of the curve later serve as the connection between the handle and the container. Therefore, the *handles* of these *control points* are tilted in such a way that the handle will have the correct connection points. The other two *control points* can be adjusted to alter the shape of the handle. Increasing the length of the *handles* results in more angular handles, while smaller *handles* with potentially twisted directions create curvier handles.

The parameter dictionary controls three additional parameters of the *Bezier Curve*. The first one is the *Bevel Depth* parameter, which gives the curve thickness and essentially creates a tube. The second parameter, *Bevel Resolution*, determines the roundness of the tube. The third parameter is the *Extrude* parameter, which stretches the tube in a specific direction. This is useful because most handles are not perfectly round but more elliptical in shape.
The final steps are to convert the *Bezier Curve* into a mesh and slightly increase the radius of the endpoints. Handles at the connection points are often slightly thicker than the rest.

The following image shows the steps of the creation of a handle.

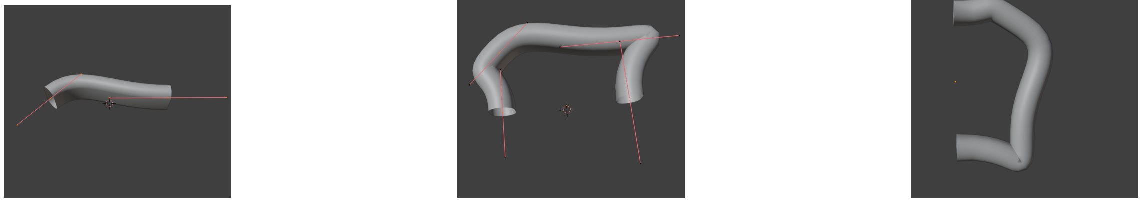
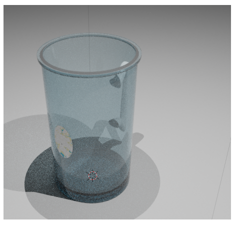

(a) Basic *Bezier Curve*
(b) Shaped *Bezier Curve*
(c) Handle as Mesh

Figure 17: Create Handle

In the following step, the handle also receives a texture. The structure of the handle texture is similar to that of the container texture. It consists of a *Principled BSDF*

node as well as a *Volume Absorption node* to account for glass-like handles, which can be found, for example, on beer glasses.
The handle is then joined to the container structure in the same way the label was attached, using a *Lattice* and a *Lattice Modifier*. The complete container can be seen in Figure 18.


(a) View on Label

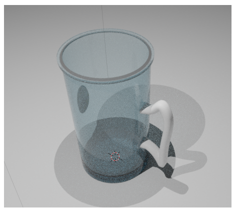
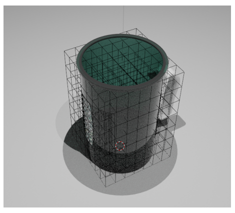
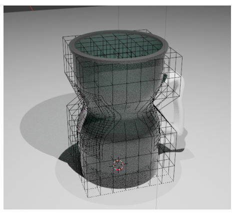

(b) View on Handle

Figure 18: Add Label and Handle

As seen in Figure 18, the shapes are still primarily cone-like. To enable more complex shapes, the program utilizes the *Lattice* object along with the *Lattice Modifier*. The current container, along with the attached label and handle, is placed inside the lattice and connected to it. By modifying specific parts of the *Lattice*, the overall shape of the container can be influenced. These transformations depend on the container type.

In general, some layers of the *Lattice mesh* are scaled either inwards or outwards. Both the selection of layers and the degree of scaling are randomized. However, for each container type, certain assumptions were made regarding which transformations are most appropriate. At this step, the bottleneck is shaped. For example, glasses can be given a narrower waist for easier gripping, while cups may be widened at the top to resemble a mug.
The described steps can be seen in figure: 19


(a) Add *Lattice* to Container


(b) Alter *Lattice*

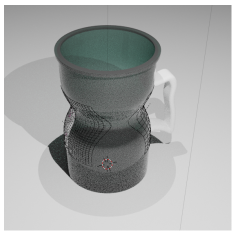

(c) Finish Process

Figure 19: Add Geometrical Complexity

The liquid is the next part added to the container. The fill level for each container is a random value between 0 and 1, where 0 means empty and 1 means full to the brim. To fill each container, the program first determines the lowest point of the inner walls. Next, it finds the highest point of the brim. Using these two values, the fill height is calculated.
For example, when until now a container was created, with a total hight of 11cm a bottom thickness of 1 cm and a fill hight of 50 percent. Then the program calculates

first where those 50 percent would be. Including the offset of 1 cm the 50 percent fill hight of the container would be at 6cm, measured from the ground.
Next, the inner wall of the container, up to the calculated fill height (6 cm in this case), is duplicated and separated from the container mesh. This new mesh serves as the base for the liquid. Then, the upper edge of the liquid is slightly extruded inwards and downwards to simulate surface tension. Finally, the edge is filled to close the liquid mesh.

The texture of the liquid consists of the same *shaders* as the container and the handle. A *Principled BSDF node* controls the *IoR*. For water, this value is approximately 1.33. The *Volume Absorption node* controls the liquid's color and density. Higher absorption values combined with a brownish color, for example, could represent coffee, whereas a lower absorption value with a lighter brown color could resemble apple juice.
Figure 20 illustrates the previously explained steps, as well as two glasses with different fill levels.

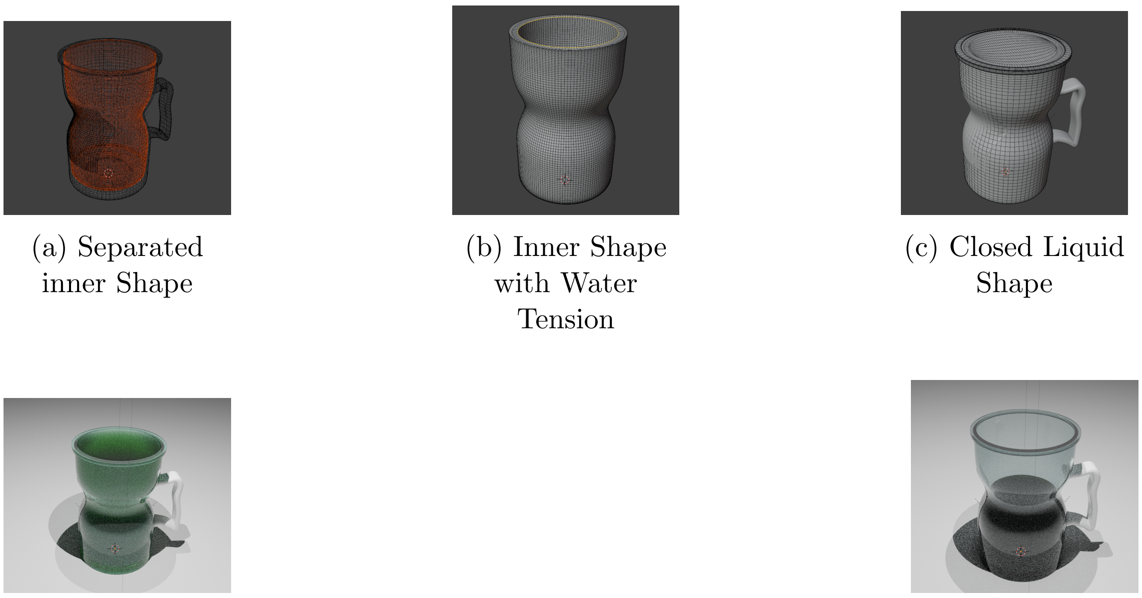
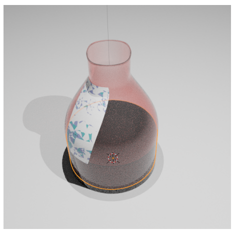

(a) Separated inner Shape
(b) Inner Shape with Water Tension
(c) Closed Liquid Shape
(d) Full Glass
(e) Half Full Glass

Figure 20: Creating the Liquid

For cups and glasses, this marks the end of the creation process. The *Blender* file is then saved, using the current timestamp and container type as the filename. This naming convention facilitates easier evaluation of the generated models later.

The final step for bottles is the creation of a bottle cap. To achieve this, the top of the model is accessed, and the bounding box of the bottleneck is used to determine the cap's radius. Once the radius is known, a new primitive cylinder is added to the workspace with the corresponding diameter. The cap is then scaled down to a realistic size.
Finally, the cap's texture is applied using a *Principled BSDF node*, where the color is adjusted accordingly. An example can be seen in Figure 21.


(a) Bottle without a Cap

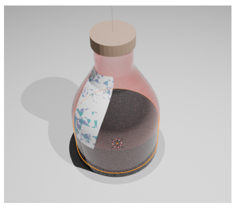
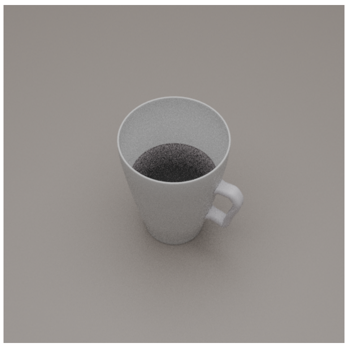
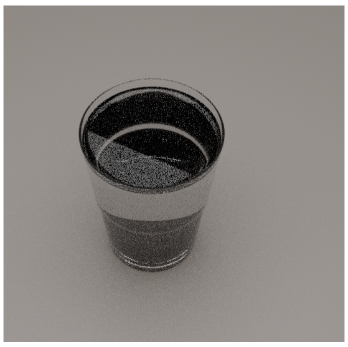

(b) Bottle with a Cap

Figure 21: Bottle without a Cap and with a Cap

4.3.2 Handmade Cups, Glasses and Bottles

Three models were handmade in *Blender*. Two were glasses and one is a cup. The models can be seen in Figure 22.


(a) Handmade Cup


(b) Handmade Glass

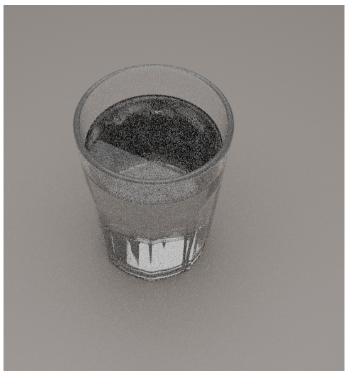
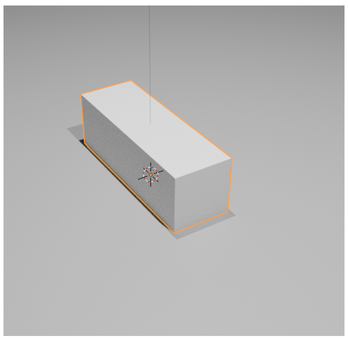

(c) Handmade Glass

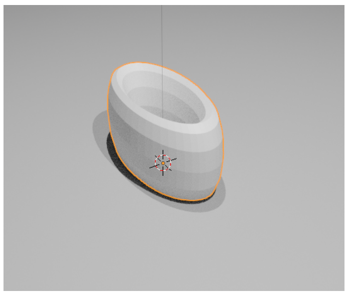
Figure 22: Handmade Models

Each model was made with three different fill hights. One for a more empty container, one medium filled container and one fuller container.

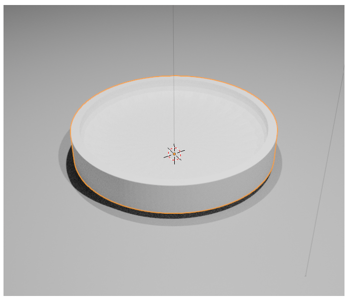
4.3.3 Distractors

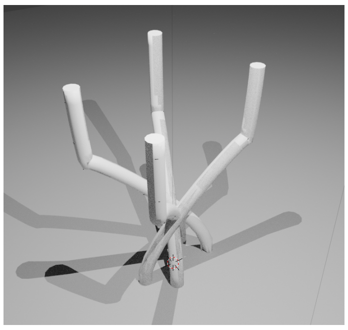
The distractor objects were created to enhance the realism of the table scene. They represent various objects commonly found on a regular table. Some have precise shapes, such as a laptop, a table, or a smartphone, while others are more abstract and resemble general shapes that might naturally appear.
The purpose of these distractors is to partially obscure the glasses or cast shadows on the table and the containers themselves. In total, there are ten distractor objects. Initially, they do not have specific textures; textures are applied later when the objects are placed within the table scene.
The textures vary from different wood textures, to plastics and stone as well as metal.
Figure 23 displays the ten distractor objects.

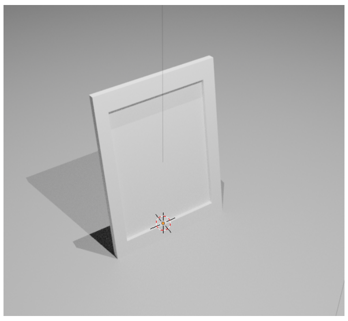

(a) Distractor 01 - Block

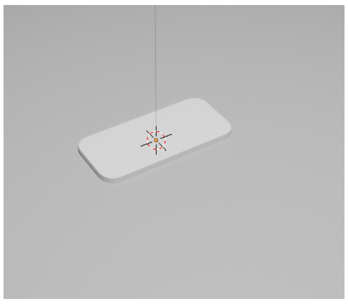

(b) Distractor 02 - Torus

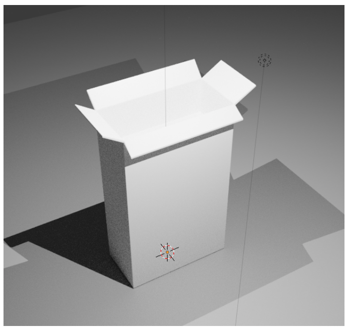
(c) Distractor 03 - Cone

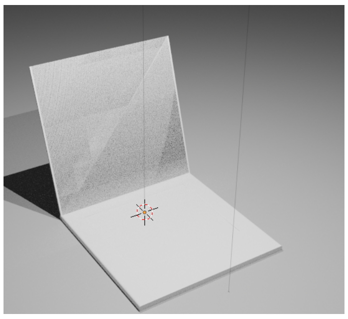

(d) Distractor 04 - Plate


(e) Distractor 05 - Candlestick


(f) Distractor 06 - Frame


(g) Distractor 07 - Smartphone


(h) Distractor 08 - Cornflakes


(i) Distractor 09 - Laptop

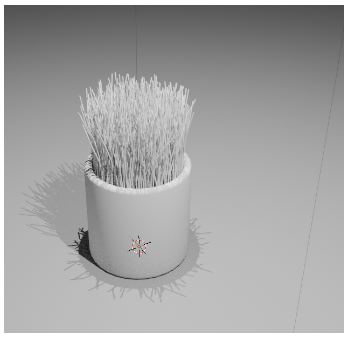

(j) Distractor 10 - Flower Pod

Figure 23: Types of Distractors

4.3.4 Environment

The environment is predefined and consists of a room with four walls, a floor, and a ceiling. When loaded into the scene, the room is initially empty. To furnish the room, various furniture objects are added, including a couch, a bed, an office chair, a desk, and several bookshelves. These objects are always positioned in the same locations and maintain the same orientation. Additionally, their textures remain unchanged.
The furniture models are sourced from the *IKEA* dataset, which is part of the *Pix3D* dataset. To simplify the loading process, each object was rescaled, its origin was set to the lowest point of the model, and its axis alignment was adjusted. Each object was then saved in a separate *.blend* file for easy access.
The placed furniture objects can be seen in Figure 24.

The room itself changes over time, with the color of the walls varying across different scenes. Additionally, the texture of the floor is modified from scene to scene by applying different materials.
The room features three transparent windows, allowing light to enter from the surrounding environment. This environment is simulated using a *High Dynamic Range Image* (*HDRI*) file.

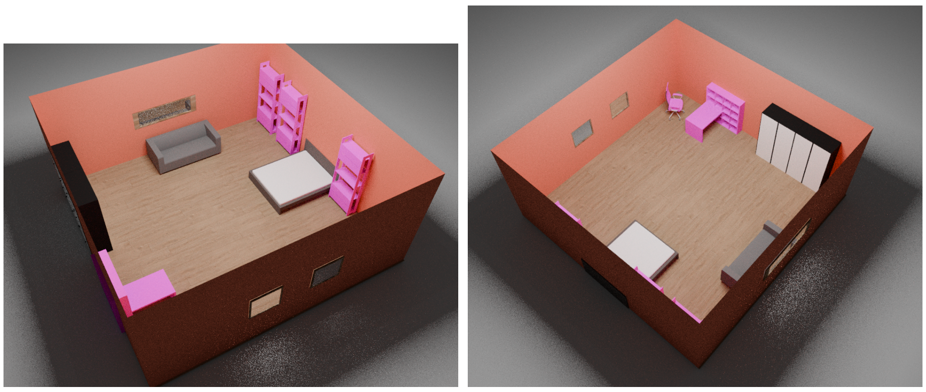

(a) Room Side A (b) Room Side B

Figure 24: Room with Objects

The tables, where the containers and distractors are later placed, were also altered in the same way as the furniture objects to simplify the loading process.

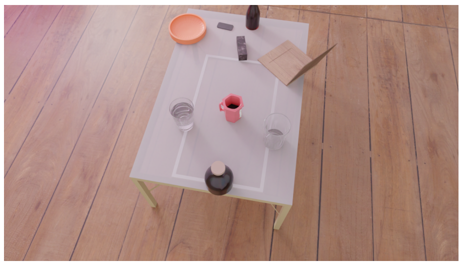
4.3.5 Building the Scene and Rendering the Image

To create the rendered scenes, *Blenderproc* is used. The rendering pipeline follows a specific scheme. First, all necessary models are loaded into Blender using the *Blender Python API* via *Blenderproc.* For each scene, a random selection of premade cup, glass, or bottle models is chosen and loaded. Additionally, some clutter objects are included to enhance realism.

An empty room is initialized along with the required furniture models. Textures for the floor and the window frames are applied, and the walls are assigned a random color to increase variability. The furniture is placed in its predefined positions. A table from the *IKEA* dataset is also loaded. If the selected table is made of wood, a corresponding wood material is applied to ensure a realistic appearance. Different types of wood textures are used to enhance the variety in the rendered scenes.

Next, the loaded cup, glass, and bottle models, along with the clutter objects, are placed on the table. The placement is done in such a way that the objects are randomly distributed across the table surface, while ensuring that they do not intersect or collide with each other.

To achieve this, the largest object is placed first, and its bounding box is measured. Subsequent objects are then placed one by one, ensuring they are positioned outside the bounding boxes of the already placed objects. This process continues until all objects are placed. The goal is to create a realistic and cluttered table scene. Two example results of such scenes are shown in Figure 25.


(a) Table Scene 01

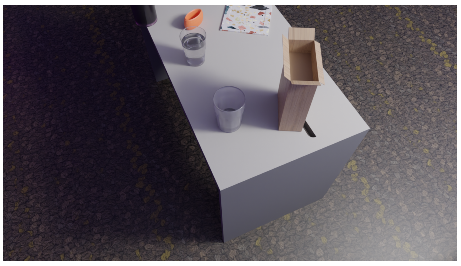

(b) Table Scene 02

Figure 25: Table Ccene

The next step in creating the scenes is setting the lighting. To achieve diverse lighting conditions, three *Blender Point Lights* are randomly placed within a bounding box measuring 6 m × 6 m × 2 m, centered around the table. This setup allows for a wide variety of illumination scenarios. Occasionally, lights may even be placed outside the room boundaries, resulting in intentionally darker scenes.

However, it is ensured that all light sources are positioned above the table, as the 2-meter vertical range of the light box starts at the height of the tabletop.

The next part is crucial for the segmentation task and requires a brief explanation of the concept of *custom properties* in *Blender*. In *Blender*, each object can be assigned multiple *custom properties*, which serve as metadata—essentially key-value pairs that can be used to uniquely identify and categorize objects. This functionality is particularly useful for tasks such as semantic segmentation, where each object must be assigned a unique identifier so that *Blenderproc* can generate semantic segmentation masks by mapping each pixel in the image to its corresponding object ID.
In this thesis, however, the focus is not on semantic segmentation masks but rather on generating visible, invisible, and amodal segmentation masks. This mapping was achieved through the use of *custom properties.* For each container and liquid object in the scene, masks were rendered individually by toggling visibility via custom properties. Specifically, for a given object, a custom property was created and set to `1`, while all other objects were assigned a value of `0` (background class). This

selective masking process was repeated for every container and liquid object in the scene to generate the appropriate segmentation outputs.
Special consideration was given to labels the containers. These labels were assigned the same *custom property* as their corresponding container, since they are considered part of the same object and in reality often obscure the inside of the glass. In contrast, handles were not treated the same way. They belong to the background class.

The invisible mask was created by subtracting the visible mask from the amodal mask. Figure 26 and Figure 27 give an example of the created masks. These masks represent the state of the water glass seen in Figure 25b, which stands next to the cereal box.

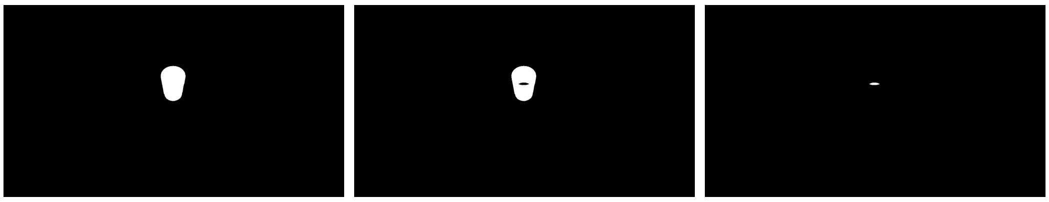

(a) Container amodal Mask (b) Container visible Mask (c) Container invisible Mask

Figure 26: Container Masks

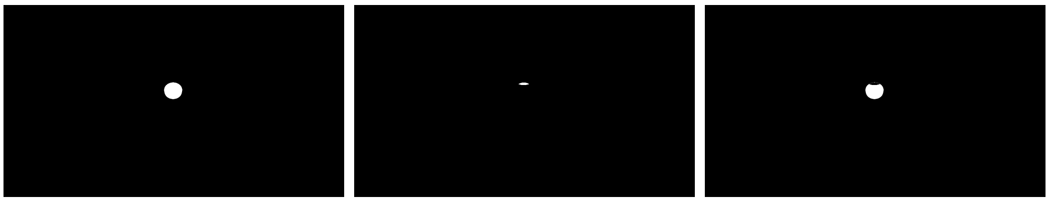

(a) Liquid amodal Mask (b) Liquid visible Mask (c) Liquid invisible Mask

Figure 27: Liquid Masks

The render settings for the *Blender Render Engine* are the following. The render mode that was used to create the *RGB* images, like in Figure 25, was *Cycles*. This is the *Render Engine* that models the light path based on physical assumptions. The settings for that are the following:

- maximum amount of *samples*: 516 (up to 1032 and 2064 for the last 200 scenes)
- maximum amount of *diffuse bounces*: 64
- maximum amount of *glossy bounces*: 64
- maximum amount of *transmission bounces*: 64
- maximum amount of *transparent bounces*: 64
- maximum amount of *volume bounces*: 64
- maximum amount of *bounces* per ray: 124
- noise threshold: 0.01

The virtual camera has the same properties as the camera that was used to create the training dataset. The camera resolution was set to 1280×720 pixels, and the

Horizontal FOV is 69°. In *Blenderproc*, it is possible to implement lens distortion, but for the used camera, it was not possible to find the correct values.

4.3.6 Model Statistics

With the described method of creating different models for cups, glasses and bottles a total of 150 different model as been created. That means for each container type there are 50 different models. This includes the same container model but with different fill heights. In general there are 24 different cup as well as 16 different glasses and 25 different bottles. So for each container there are around two versions of it with different fill heights. Each version of the container, also have different color.
As expected from a linear distribution, from which the fill height was sampled, the average is close to 0.5 percent. The distribution is not uniform though. There are more fill hights in the 30 percent area, then in the 60 percent area. This can be seen in Figure 28. The Figure shows the distribution of the fill height. Here each label contains the counts for fill heights in the area of ±2.5 percent.

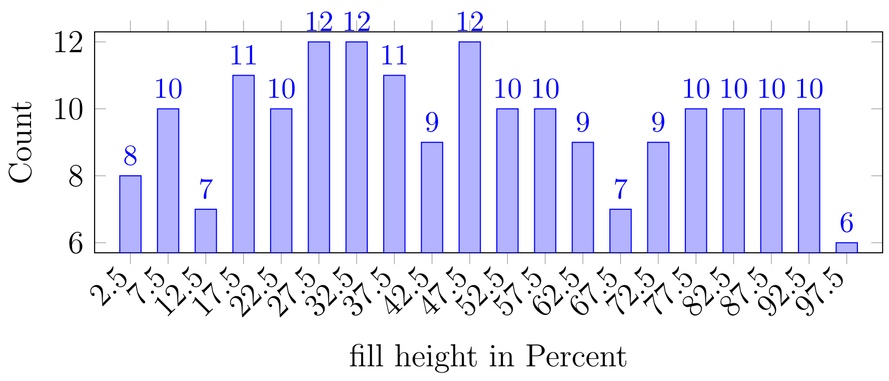
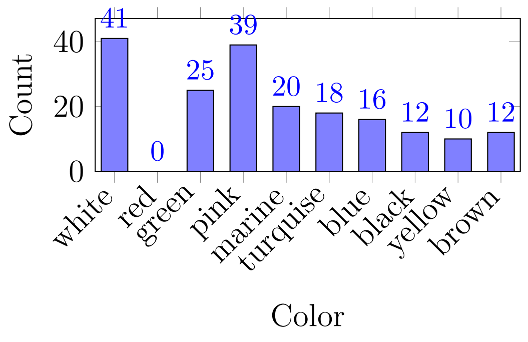


Figure 28: Histogram of Fill Height Distribution

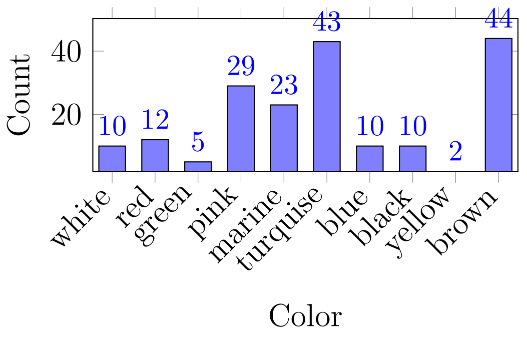
The containers could have the following shades of colors:

- white (clear for transparent container)
- red
- green
- pink
- marine
- turquoise
- blue
- black
- yellow

The distribution is presented in Figure 29: The data shows a small overhang to white and pink of the containers. For the Liquids brown and turquoise are most frequent.



(a) Container Color Distribution



(b) Liquid Color Distribution

Figure 29: Comparison of Container vs. Liquid Color Counts

From all the containers 125 have labels, and 94 have handles.

4.3.7 Image and Masks Statistics

In total an amount of 500 scenes is rendered. The rendering took several hours. As explained each rendered scene contains multiple segmentation maps, that need time to render as well. The images were rendered using two graphics cards from type *NVIDIA TITAN V* as well as the *CPU core* from type *Intel Core i7-7700* with *3.6 GH*. The rendering of one scene on the named machine took at average 12 minutes. One scene consists of two frames, where each frame has one colored image as well as one semantic segmentation image, where each object is a new label.

Each frame also consists of five masks of amodal masks, visible masks and invisible masks for the container and for the liquid. So that in total each scene has 64 images. Each masks of an object for both frames ware rendered simultaneously. The time it took for rendering one of the objects masks for both frames took around 60 seconds. The rendering of the color image, for both frames, took around 90 seconds. It seamed that *blenderproc* did not used the *GPU's* to render the segmentation masks. Thats why the segmentation masks took compared to the much more complex colored image quite a long time.

For 500 scenes with each two frames and two times 10 objects masks a total of 20.000 masks is created. The invisible masks is calculated by subtracting the visible form the amodal mask.

4.3.8 Complete COCOA Dataset

The complete *COCOA*-style dataset, that was created in the scope of this thesis, is structured in folders. Each folder resembles a scene and contains subfolders for the frames. Each subfolder contains the color image, the visible, invisible and amodal mask as well as a semantic segmentation mask. The scene folders are named like *scene_0, scene_1, ...* and subfolder are named like *frame_0, frame_1, ...*. The color images are named after the scene and the frame. So for the color image, in the folder *scene_0* and *frame_0* the name is *color_00_.png*. The masks are named after the objects of the mask. So if the object is named *basic_mesh2025_03_20_17_27_07*, the corresponding amodal masks name would be *amodal_basic_mesh2025_03_20_17_27_07_segmaps.png.* This works equivalent for the visible and invisible mask.

Each frame also has a *cocoa_per_frame.json* file. This file contains the following information:

- image_id
- file_name
- height
- width
- date_captured
- image
- annotations
- name
- label
- area
- bbox
- not_grouped
- occlude_rate
- fill_hight
- transparency
- image_id
- visible_mask
- invisible_mask
- amodal_mask
- size

The *image_id* is the number of the color image. *File_name* is the complete color image name. *Height* and *width* are the height and width of the image in pixel. *date_captured* is the date when the *cocoa_per_frame*-file is created. *Image* is a complete path to the image.

Under *annotations* more information to the content on the image is found. *Name* is a alphabetic list of all container and liquid objects on the image. *Label* contains a list for each object, if it is a container or a liquid. *Area* is the area in pixel per object and per mask. *Bbox* is the bounding box for each object and mask. *Not_grouped* is a boolean value that is *true* when the object is not hidden by other objects. If the object is hidden by more then fifty percent by another container then it would be *false*. *Fill_height* is the fill height of each container and liquid. *Transparency* is the value from *Blender* that describes how transparent an object is. For liquids and transparent containers it is the *volume absorption* parameter from *Blender*. *Visible_mask, invisible_mask* and *amodal_mask* are the local paths to the masks files. *Size* is the number of objects. At the creation of the *cococa_per_frame* file it is also checked if an object is hidden completely or outside of the camera field. In this case it is not mentioned in the file.

4.4 Data Preprocessing and Augmentation

To solve both tasks, two subdatasets are build from the *COCOA*-style dataset. The following sections explain what data is used to finally train the two *neural nets.*

4.4.1 Dataset One

This dataset is used to train the *neural net* of the first stage for the detection of containers on the table. One sample consists of an image from a table scene and the corresponding mask. Both are rescaled to a height of 360 and a width of 640 pixels. Here, only the visible masks of the *COCOA*-styled dataset were used. Each sample was also augmented. To double the length of the dataset, each sample was flipped left to right, and the contrast and brightness were adjusted. Additionally, normally distributed noise was added. Considering the 500 scenes with two frames each and doubling the original dataset length through augmentation, a total of 2000 samples are available for training. Again, 20 percent of these are used as the validation dataset, leaving 1600 samples for training.

On average there are about 8,800 annotated pixels per image, making around 14 million annotated pixels for the entire training dataset. A sample can be seen in Figure 30.

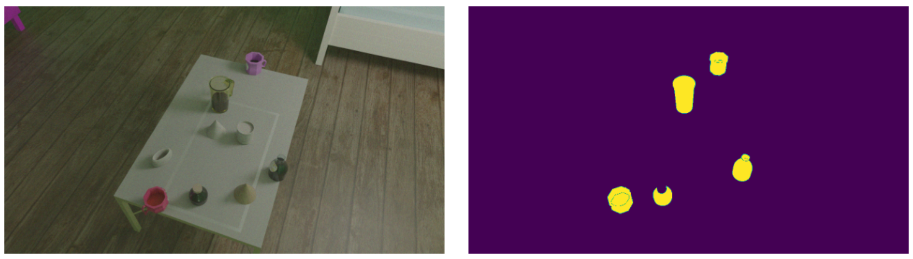

(a) Sample (b) Visible Mask

Figure 30: Sample Dataset One

4.4.2 Dataset Two

This section describes how the final dataset, used to train the *neural net* for predicting liquid fill height, is obtained. Smaller image snippets were cut from the original color images using the *cocoa_per_frame.json* file, which contained the bounding boxes of each container and liquid object. To extract separate mask snippets for the container and the liquid, only the container bounding box was used. The center of the container bounding box was then calculated and randomly shifted by ±15 pixels. An image snippet with a height of 300 pixels and a width of 252 pixels was cut from the image, along with the amodal masks for the container and the liquid. This size normally fits the complete container. Because of this, each rendered image produced multiple snippets. Each sample also includes the fill height as a decimal number. From 500 rendered scenes with two frames each and approximately five objects per frame, this results in around 5,000 snippets used for training. Containers, that are not visible on the image, are left out.

The newly created image snippets were also augmented by flipping them left to right and randomly adjusting the brightness. The contrast was also changed, and normally distributed noise was added. The final dataset is therefore twice the length of the unaugmented dataset, resulting in a total of 9,632 samples. With an 80/20 split for training and validation datasets, approximately 7,700 samples are available for training.

On average, 10 percent of each snippet is labeled in the container masks, and 6 percent is labeled in the liquid masks. With an image size of 75,600 pixels, this corresponds to an average of 7,560 and 4,536 labeled pixels per image, respectively. Across all 7,700 training samples, this amounts to around 58 million container-mask labels and 35 million liquid-mask labels. The complete label set for the second dataset includes: the masks for the container and the liquid, the fill height, and the transparency of both the object and the liquid. Transparency values range from 0 (opaque) to 1 (transparent).
The training and validation datasets also contain 20 percent of the test dataset, which consists only of real image samples. This should slightly improve the predictions. A sample for the dataset that trains the second-stage *neural net* can be seen in Figure: 31. The fill height of the shown glass is 20 percent, and the transparency is 0.9 for the container and 0.2 for the liquid.

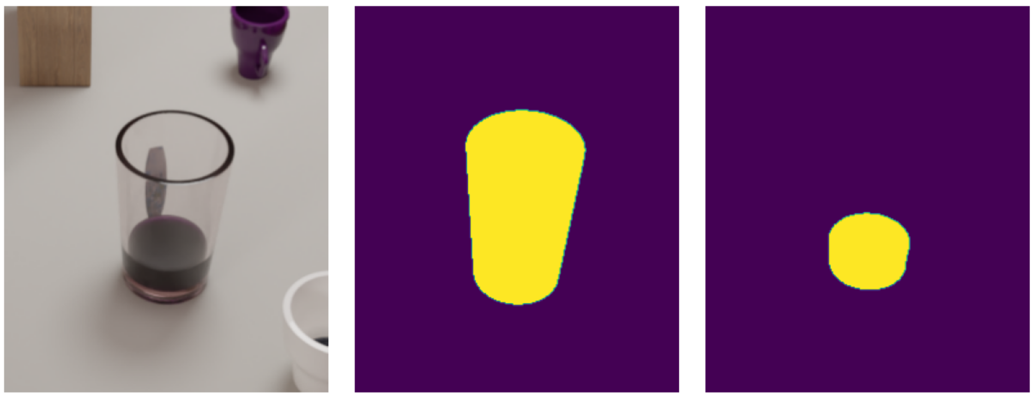

(a) Sample (b) Amodal Container Mast (c) Amodal Liquid Mask

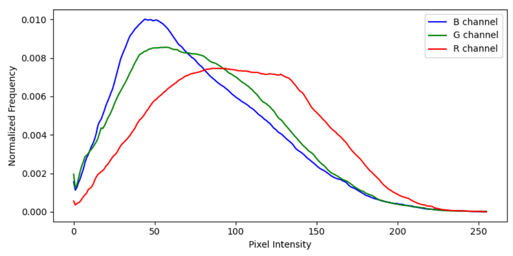
Figure 31: Sample Dataset Two

4.5 Synthetic Dataset vs Real Dataset

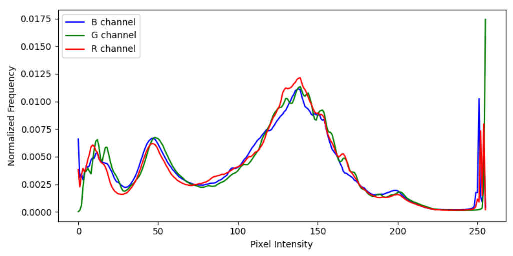
In this section, a comparison is made between the rendered portion of the dataset and the real images. As mentioned before, the setup should work in a real-world application. To simulate that, the test dataset was created. For both datasets, used to train the stages, the color histograms of the rendered and real datasets are compared. The color histogram provides an overview of how often each red, green, and blue color value appears in the images. The x-axis of the histogram shows the intensity of the color, from dark to bright. The y-axis represents the distribution of the colors. With the help of the color histogram, fundamental shifts in the color space can be observed. For example, if the histogram is skewed to the left, the images are darker, and vice versa. Secondly, a *PCA* is performed and the first three *Principal Components* are compared.

4.5.1 Dataset One

Figure: 32 shows, on the left, the color histogram for the training dataset and on the right, the color histogram for the test dataset.



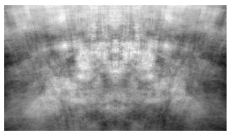
(a) Average Color Histogram Train Dataset



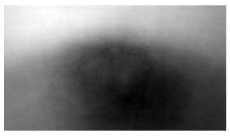
(b) Average Color Histogram Test Dataset

Figure 32: Color Histograms Dataset One

It is evident that the color histogram of the training dataset is more left-centered. The appearance of colors is not as balanced as in the test dataset. It seems that the rendered images contain many more regions of a single color. This may be because, for example, the walls in the modeled room have a uniform random color. Additionally, there are many overexposed white pixels in the real images due to the reflectiveness of the white table and wall, as well as sunlight in the images. The test dataset does not exhibit much color variance. Overall, the two color histograms do not align very well.

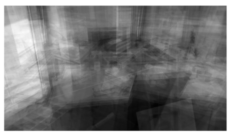


(a) PC 1 Synthetic Image

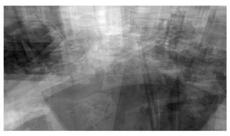


(b) PC 2 Synthetic Image

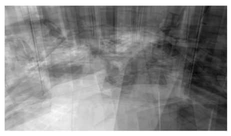
(c) PC 3 Synthetic Image



(d) PC 1 Real Image



(e) PC 2 Real Image



(f) PC 3 Real Image

Figure 33: Comparison PCA of first three Principal Components

The first row shows the first three *Principal Components* of the synthetic images, and the second row shows the first three *Principal Components* of the real images. Figures: 33a–33c are smoother and do not show as many edges. The reason for this is that the training dataset is much larger than the test dataset. The first *Principal Component* seems to focus on the structure of the room as well as individual objects, where many corners are still visible. The second *Principal Component* appears to focus on the placement of the table, whereas the third addresses light and shadows from one side to another.

Except for the third *Principal Component* of the test dataset, which also focuses on light and shadow, the *Principal Components* are not easily comparable. The table in the test dataset appears larger in the images than the tables in the training dataset. Most of the time, it spans from one side of the image to the other. In the training dataset, the table's boundaries are clearly visible. This may explain why a second *Principal Component* does not emerge in the test dataset, as it does in the training dataset.

This observation is supported by the *dot product* between the *Principal Components* of the two datasets. The *dot product* can be used as a measure of vector similarity. The result of the *dot product* is shown in Table: 2.

PC's	PC1	PC2	PC3
PC1	0.08	0.13	0.02
PC2	0.62	0.43	0.07
PC3	0.21	0.02	0.87

Table 2: PC Train Dataset vs. PC Test Dataset

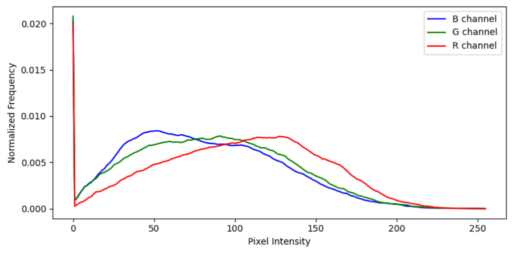
The matrix shows that the third *Principal Component* for both dataset have quite the similarity. Apparently also The first *Principal Component* of the train dataset and the second *Principal Component* of the test dataset have similarity's.

Considering both, the color histogram and the *PCA* it can be said, that the rendered dataset captures some characteristics of the test dataset.

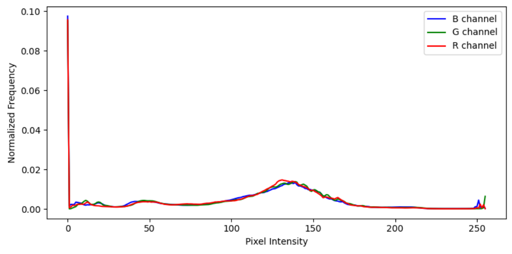
4.5.2 Dataset Two

This section compares the synthetic image snippets, to the image snippets from the test dataset, that come from the camera. Here first the color histogram is compared and secondly the *PCA* of the first three *Principal Components* is evaluated.

Figure: 34 shows the color histograms of the training dataset on the left and the test dataset on the right. Both histograms exhibit a peak at the black value, which results from cropping the images and inserting black pixels where the image would otherwise be too small. In general, the training data histogram is more spread out and more centered, although it may still tend slightly toward darker values. The test dataset histogram shows a bump at around 140, which is not present in the training dataset. Additionally, the test dataset contains more bright white colors.



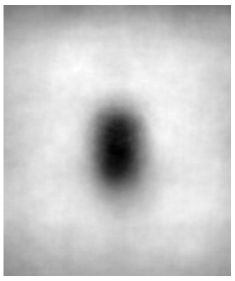
(a) Average Color Histogram Train Dataset



(b) Average Color Histogram Test Dataset

Figure 34: Color Histograms Dataset Two

The results of the *PCA* are shown in Figure 35. This time, more similarities are visible. *Principal Component* 1 of the training dataset clearly shows a container object, which corresponds to the second *Principal Component* of the test dataset. This also displays a gray shape in the center, again with more visible corners because the test dataset is smaller. The representation of the container is also larger in the test dataset.
The second *Principal Component* of the training dataset highlights the tabletop as well as a central object, with the background appearing dark. This is similar to *Principal Component* 1 of the test dataset. Again, the third *Principal Components* align between the two datasets, representing lighting conditions from left and right.

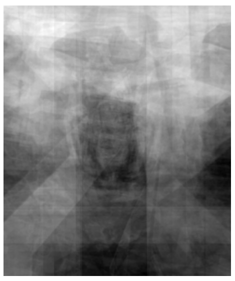


(a) PC 1
Synthetic Image

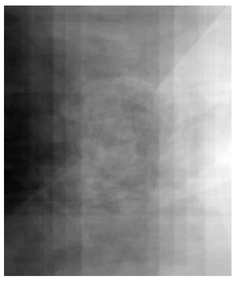
(b) PC 2
Synthetic Image

(c) PC 3
Synthetic Image

(d) PC 1 Real
Image



(e) PC 2 Real
Image



(f) PC 3 Real
Image

Figure 35: Comparison of Ground-Truth Snippets and corresponding Feature-Vector Maps

This observation between similarities can again be seen when calculating the *dot product* between the *Principal Components* of the two datasets. The results are shown in table: 3

PC's	PC1	PC2	PC3
PC1	0.26	0.94	0.05
PC2	0.93	0.27	0.01
PC3	0.02	0.05	0.95

Table 3: PC Train Dataset vs. PC Test Dataset

4.6 Neural Net

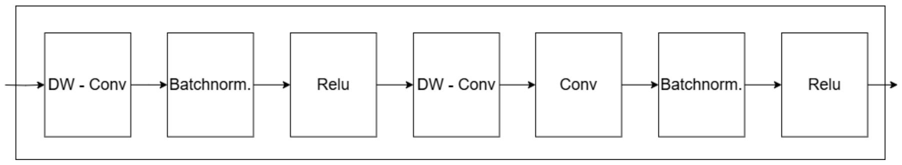
This section explains the architecture of the *neural nets* that were developed and used, as well as why certain parameters were chosen. It also details the total number of parameters in each *neural net* and compares these numbers to those suggested by the *one-in-ten rule*. Additionally, the receptive field is calculated for both networks.

4.6.1 Architekture Stage One and Training Procedure

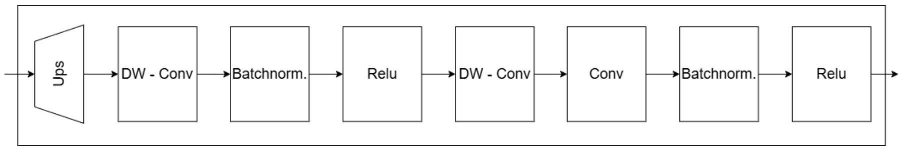
The purpose of this *neural net* is to detect the liquid containers in the scene image. The *neural net* is based on the *encoder-decoder architecture*. For that it takes the *RGB*-image with a size of: 360 times 640 times 3 as an input. The *neural net* uses a pretrained *ResNet50* backbone. The *ResNet50* networks layers are organized in blocks. This backbone uses the pretrained layers from the input to the third block of the second convolution block. The backbone is split in three parts. The first part includes the layers from the input op to the *conv2_block1_out-Layer*. The second block goes from the *conv2_block1_out-Layer* to the *conv2_block2_out-layer* and the last block goes from *conv2_block2_out-Layer* to *conv2_block3_out-Layer* The *feature vector* has here a dimension of: 90 times 160 times 256. The backbone reduces the size by a factor of 4. After the backbone the size is again halved. So the maximum reduction is by a factor of 6. This is important for the *One-in-ten-rule*.

The training dataset has around 14 million annotated pixels, that are not background. By dividing the 14 million pixels by the factor of 6, around 2.3 million pixels remain in the lowest resolution. The *One-in-ten-Rule* says that for each trainable parameter, there should be around 10 datum points. So following this rule the network should have roughly 230.000 trainable parameters. While keeping the backbone as non-trainable, the trainable parameters on the *neural net*, as it is shown in Figure: 37, counts to 134.937. Having the end of the backbone set to trainable, the number of trainable parameters increase to around 200.000. Both numbers are below the reference value of the *One-in-ten-Rule*.

The *receptive field* for this *neural net* is calculate backward from the last layer, before the upsampling begins. Until the backbone begins, the *receptive field* grows to a size of 118 pixel. With the backbone, the *receptive field* grows to 276 pixels. The head of the *neural net* consists of two types of custom layers. The *DwCNN_Cnn_Bn_Relu* custom layer consists of two *depthwise convolution* layer as well as one *convolution* layer. Also *batchnormalisation* is used as a method of regularization. The custom layer uses the *Relu* activation function. The *kernels* of the *convolution* layers are initilizied with the *He Normal* methode. A visualization of the *DwCNN_Cnn_Bn_Relu* custom layer can be seen in Figure:36. The *Ups_Dwconv_Cnn_Bn_Relu* custom layer has the same strukture like the *DwCNN_Cnn_Bn_Relu* custom layer. It just features a *upsampling* layer infront of it. This layer is used to increase the spatial dimensionality of the feature vector back again to the input size. All *convolution* layers, that are outside a custom layer, use *L2-Regulaization* with a factor of 0.00001.



(a) DwCNN_Cnn_Bn_Relu Layer



(b) Ups_Dwconv_Cnn_Bn_Relu Layer

Figure 36: Custom Layers

The *neural net* of the first stage can be seen in Figure: 37. The figures show the layers as boxes. The abbreviation inside the boxes correspond to the kernelsize *(k)*. The *neural net* has also short cut conenctions, similar to the *neural net: U-Net*. As discribed in chapter: 2.1.8, this brings local information from the backbone to the decoder.

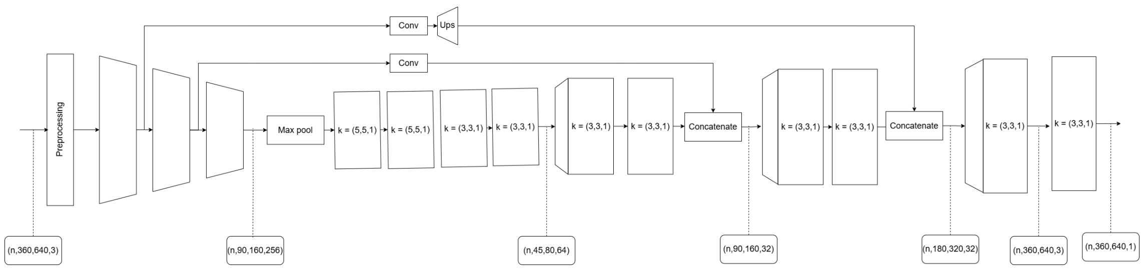


Figure 37: Neural Net One

The training spans over 40 epochs. At epoch 15 the last backbone block and at epoch 25 the second last block are unfreezed. The learing rate starts at 0.01 and decreases by a factor of 0.5 atomatically every third epoch, if the validation loss is not decresing. As a optimizer the *AdamW* is used. The loss that is used, is a combination of the *Dice-Loss* and the *BFC-Loss*.

4.6.2 Architekture Stage Two and training Procedure

The second *neural net* that was designed, has a similar backbone to the first one. The strukture can be seen in Figure: 38. It also features the same custom layers. This is now a *neural net* with three heads. The first one predicts the mask (*segmentation head*), the second on perdicts the fill height and the transparency (*regression head*).

The training procedure is a bit more difficilt here. In genral the *neural net* is trained for 80 epochs. The first 45 epochs, just the first head is trained, that predicts the masks. In this 45 epochs the *neural net* allready learns a good representation of

how the masks look like. This is needed, because the *regression head* learns from the geometry of the learned mask.
In these first 45 epochs the *regression head* loss is set to zero. After the 45 epoch this methode switches. Now the *regression head* is learnign and the segmentation head is not. After epoch 55 this changes again and now the complete *neural net* exept for the first two backbone block are set to trianable.
The loss function for the *segmentation head* is again a combination between the *Dice-Loss* and the *Binary Focal Crossentropy Loss*. For the *regression head* the *Mean Absolut Error* was used. The architecture also uses short cut conenctions to bring high level features from the backbone to the encoder. Again, all *convolution* layers outside a custom layer use the *L2-Regularization*. The *FC* layer in the *regression head* uses *Dropout* as the *Regularization* methode.

As can be seen in Figure: 38 the *neural net* has one downsampling step less, compared to the first stage *neural net*. The reduction is here only by a factor of 4. The training dataset for the second stage is larger then the training dataset of the first stage. It has arround 58 million labeld pixels, that are not background. When devided by a factor of 4 this still accounts for 14.5 million pixels. Considering the *One-in-ten-Rule*, that for each parameter there should be 10 datum values, the *neural net* could have around 1.45 million parameters.

The abbreviations, shown in Figure: 38 stand for the following.

1. *k* kernelsize
2. *f* number of filters

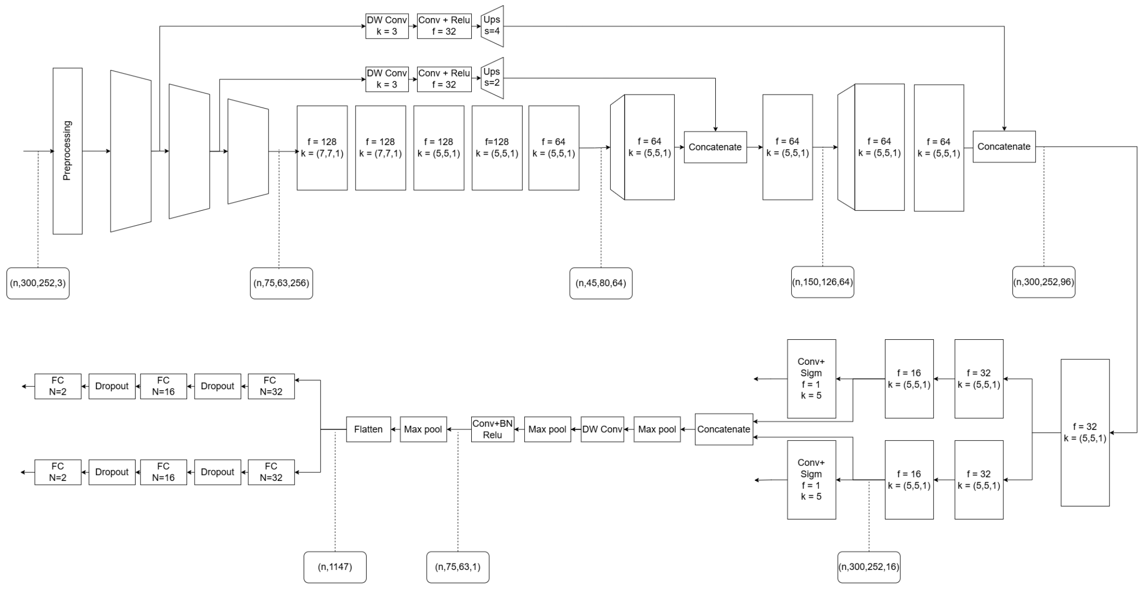


Figure 38: Neural Net Two

The *neural net*, that was designed has roughly 200.000 parameters, when the last backbone layer is not trainable. With a trainable last backbone block, the parameter count would go up to 271.000 parameters. This is below the reference of the

One-in-ten-Rule. The *receptive field* from the last layer (before the first *upsampling* custom layer), to the input spans over 132 pixel.

The regression head, that is going to predict the fill height and transparency, does not use the final masks. Instead it uses the *feature vector* of the second last layers. The idea was, that the feature vector still holds more usefull information, for learning the fill height and transparency.

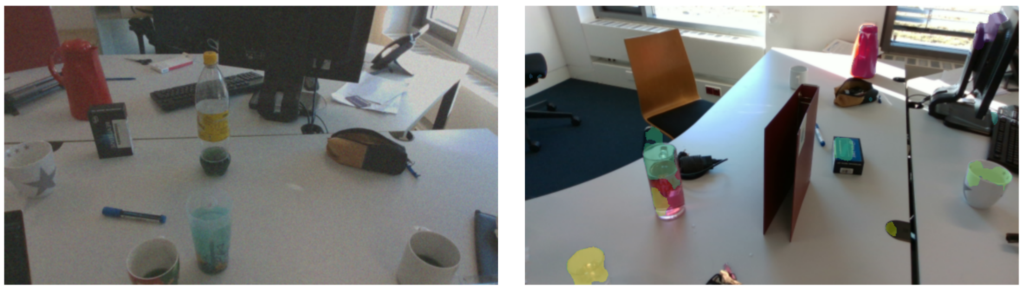
5 Experiments and Results

This section shall give an overview over the achieved results for each stage. As well as the results of a combination of, stage one and stage two. This should represent the real world application. For this section a *Dice-Metric* is defined as *Dice-Metric*=1-*Dice-Loss*. With the *Dice-Metric*, a perfect prediction would be the value: 0 and a total miss would be the value: 1.

5.1 Results on Stage One

When evaluating the trained model with the test dataset, it produces a *Dice-Metric* from 0.6588 and a *Binary Focal Crossentropy Loss* loss from 0.2877. This is a rather disappointing result, because the optimal *Dice-Metric* and *BFC-Loss* would be at 0. When this result is compared to the result of the validation dataset (*Dice-Metric*=0.1087 and *BFC-Loss*=0.0438), it gets clear that this model does not transfer very well between synthetic dataset and real dataset.

When analyzed the predictions the following patterns are visible. Figure: 39 visualizes them.



(a) Issue with Augmented Images (b) Other Patterns

Figure 39: Error Patterns of Visualization

Figure: 39a shows that augmented images does not get any prediction on them at all. This is not a substantial issue, because they do not relate to the original camera in use. It is still unfortunate.
More pressing are the issues on the un-augmented images. Here the typical issues of *false prediction* (labeling something as a container although it is something else) and missed classifications (does not labeling something as a container although it is) dominate. For bigger containers like bottles and big cups it happens also frequently, that the label is not continuous but broken into two labels. Also incomplete labels appear. Incomplete labels are still okay for the two staged approach that is later discussed, because the area that gets cut is big enough to fit the container in it.
Some labels also are well and sufficiently predict a containers location. For that the *Dice-Metric* between each detection and ground truth mask is calculated. Over the complete test dataset there are 200 labels distributed. In total the *neural net* made 266 predictions. The Table: 4 shows, the number of labels that were smaller then a specific *Dice-Metric* threshold and the delta, to the threshold below.
The table shows that if a container was detected it was detected quite well. The values around 0.6 correspond to in-continuous labels, that were split in two. The

Dice-Threshold	Number of Labels	δ
< 0.1	26	26
< 0.2	43	17
< 0.3	50	7
< 0.4	56	6
< 0.5	59	3
< 0.6	66	7
< 0.7	67	1
< 0.8	70	3
< 0.9	77	7
< 1	266	189

Table 4: Dice Threshold

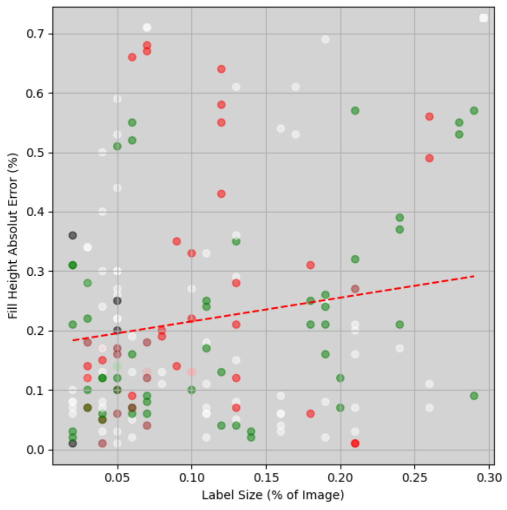
biggest number are false predictions with 189. 145 labels were smaller then 400 pixel. If the labels were filtered, for labels bigger then 400 Pixel, only 56 Labels stay as *false predict*.

5.2 Results on Stage Two

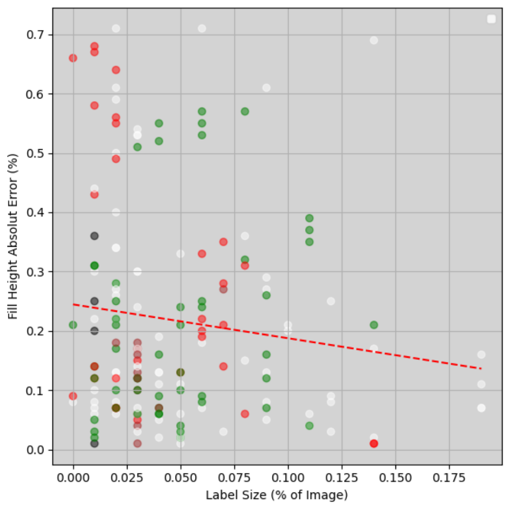
First the classic results of the loss functions are discussed. Then there are set in context with the fill height and the label size. Furthermore the performance for different colored liquids and cups is analyzed as well as for different transparencies.

The average loss of the fill height over all batches of the test dataset is 0.227. Because the fill height loss is the *Mean Absolut Error* this value can directly be interpreted as a percentage. So in other words on average the fill height detection has a uncertainty of around 23 percent. The prediction of the transparency achieved a 0.3204. The loss function for the transparency was also the *mae*. So an average there is an uncertainty of 32 percent on the test dataset.

The *Dice-Loss* over all batches of the test dataset is 0.212. The *BFC-Loss* is 0.067. As described earlier the *Dice-Loss* measures how good shapes, here the groundtruth mask and the predicted mask fit together. The *Dice-Loss* is here calculated to $1 - DiceLoss$. So a value closer to 0 is better. If the mask would be perfectly overlap the value would be 0 here.
The first property that is compared, is the label size to the absolute fill height error. This should give an understanding where the containers are placed best in the frame. Smaller labels correspond to further away containers and bigger labels to closer ones. This can be seen in Figure: 40. The colors of the dots stand for the the color of the liquid that was inside the container. Red dots correspond to a red liquid, white correspond to a clear liquid, green correspond to a green liquid and gray to a grayisch color. Different transparencies were not taken into account.



(a) Label Size Container vs. abs. Error

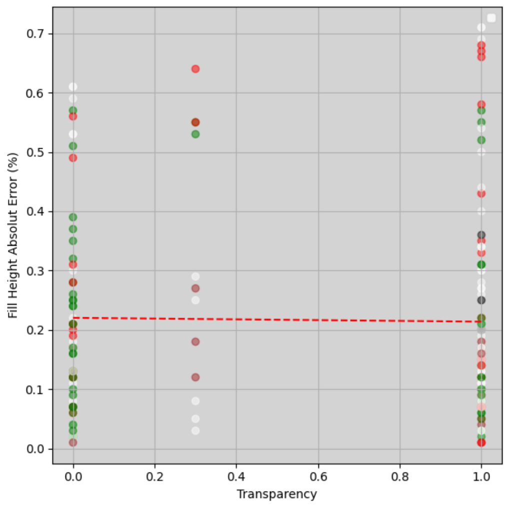


(b) Label Size Liquid vs. abs. Error

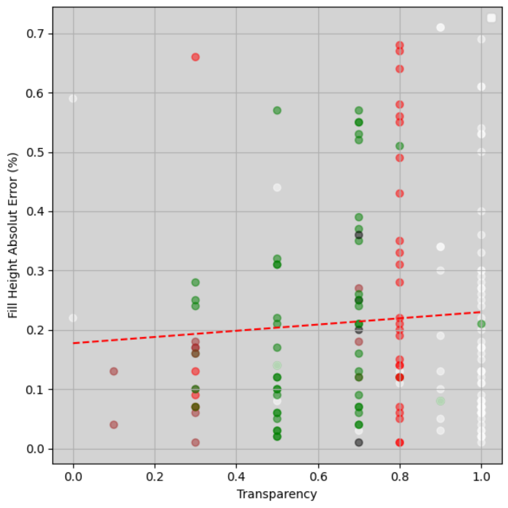
Figure 40: Label Size vs. absolut Error

Figure: 40a shows that the error for the prediction of the container masks, rises with growing label sizes, that the neural net should predict. This is probably a result from to little samples with big label sizes in the dataset. The *PCA* analysis of the train and test dataset also allowed for the result that the cups are systematically different sized.
A different observation can be done for the size of the liquid label in Figure: 40b. Here the error shrinks for larger labels. Intuitively this makes sense, because smaller amounts of liquid are harder to detect.

The transparency for the test dataset was not measured, instead it was estimated with the constraints that 0 is not transparent, so opaque and 1 is transparent. The behavior of the transparency against the absolute error can be seen in Figure: 41. The color of the dots corresponds again to the color of the liquid.



(a) Transparency Container vs. abs. Error



(b) Transparency Liquid vs. abs. Error

Figure 41: Transparency vs. absolut Error

From the Figure: 41a it is visible that the transparency of the container does not have a direct influence on the absolute error. This is against the expectation, that the visibility of a liquid inside a transparent container has a influence of the prediction. But maybe the *neural net* understands the shapes of containers so well that it makes no difference if the liquid is not visible.
With splitting the transparency in high and low transparency at 0.5 for the containers, the average error for a high transparency has $\mu = 0.21$ with a $\sigma = 0.19$ (98 samples). Compared to $\mu = 0.22$ and $\sigma = 0.18$ (76 samples) for a low transparency.
The transparency of the liquid does matter to the absolute error. Here Figure: 41b shows a small increase in the error to more transparent liquids. The white dots here represent clear water. This also should be the hardest to detect case.
With the same split, the liquids with a high transparency have a $\mu = 0.23$ and $\sigma = 0.19$ (128 samples), For the more opaque liquids the average and standard deviation is like: $\mu = 0.17$ and $\sigma = 0.15$ (46 samples). The dataset size varies here quite a lot, so the *Welch Test* was choosen to keep the results comparable. The result is a p-value of 0.018. This result is significant.

The Figures above also shows the color distribution for the comparison. To have a more concrete understanding also the average and standard deviation for the absolute error was calculated per color. Here the sample sizes vary quite much as well. To compensate for that a *Welch Test* is computed again, to keep the results comparable.
The results for each possible color, for the containers and the liquids can be seen in the following tables.

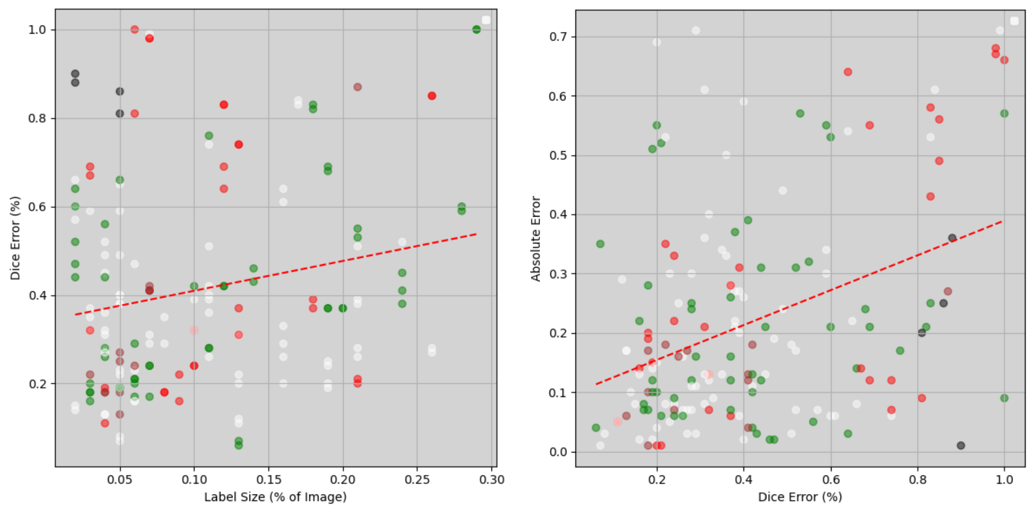
Color	Average	Sdt.	Nb. of Samples
green	0.24	0.17	26
white	0.21	0.19	144

Table 5: Statistics about Container Color and absolute Error

Color	Average	Sdt.	Nb. of Samples
red	0.28	0.22	30
green	0.21	0.16	52
white	0.21	0.19	75
brown	0.13	0.07	13

Table 6: Statistics about Liquid Color and absolute Error

Unfortunately inside the test dataset there are only to colored containers. Once a shades of white and the others are more greenish. Here it is visible that the color of the containers does not have a influence on the absolute error.
This does not translate to the liquid color. Here the color brown is generating the best results by far. With the *Welch Test* it can be said that this is not a coincided dew to a smaller dataset. With a p-value of 0.01 between the brown liquids and the white liquids. This is significant.
The average *Dice-Loss* for the containers over the test dataset is at 0.12 with a standard deviation of 0.13. For the liquids the average *Dice-Loss* value increases, up to a value of 0.41. With a standard deviation of 0.24. The standard deviation of the *Dice-Loss* is large. A reason for that could be that larger labels were predicted poorer, as can be seen in Figure: 42a.



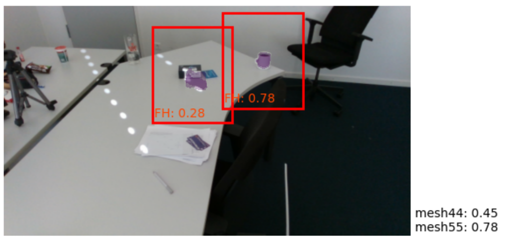
(a) Dice Error vs. Label Size (b) Dice Error vs. abs. Error

Figure 42: Analyses Dice Error Liquid

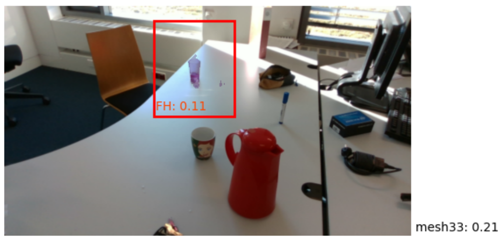
Figure 42b shows the relationship between the liquids *Dice Error* and the absolute fill height error. Here it is quite clear that the fill height error rises with a worse liquid fill mask. The colored dots, represent again the color of the liquid.

5.3 Results on Combined Stages

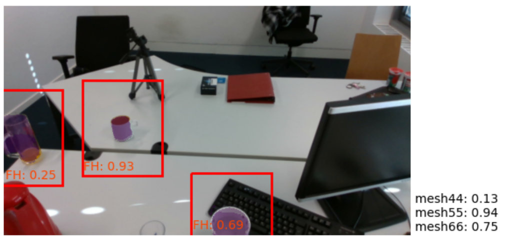
The results for the combined stages, were gathered by using the test dataset of the stage one and pass it through both stages end to end. For that the predictions that the first stage does, are filtered. All labels below a pixel size then 400 pixel are ignored. But due to the weak detection results, that the second stage inherits form the first stage, the overall two staged prediction results are weak as well.



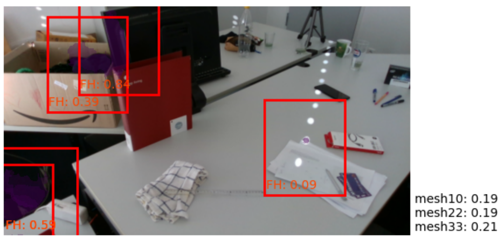
(a) Sample 1



(b) Sample 2



(c) Sample 3



(d) Sample 5

Figure 43: Combined Predicted form Test Dataset

The first stage created predictions for 64 from 78 samples. On those 64 samples the first stage made *119 false predictions* and missed *47* predictions, using the 400 pixel filter. Only considering the predictions that actually contained a container the *mae* is 0.2395. This translates to around 24 percents.
This result is very close to the result the test dataset made on the second stage only.

6 Discussion of Results

This section discusses the findings that were made, when presenting the results of the models. At first, the first stage weaknesses are elaborated as well as possible reasons and solutions are given.
After that the results of the second stage are summarized.

Now having all results together, it is quite clear that the weakness of this solution is the first stage of detecting the containers. The first model continuously makes *false prediction* or misses predictions. The average *Dice-Metric* for the test dataset is at 0.6588. As already described in chapter: 4.5 the training dataset differs to much from the test dataset. Also the test dataset contains some wrong labels or the same image multiple times. All this does not increase the performance of the first stage.

The *neural net* that was designed as the first stage, used well tested components, like a pre-trained backbone, short cuts and *convolution + batchnormalisation + relu* based custom layers. Through switching the trained model with a different architecture, like the *Yolo neural network* the weakness could get eliminated. The *Yolo neural network* is still state of the art in semantic segmentation and could be used as the backbone, instead of the *ResNet* backbone. Also a larger training dataset, that shares more similarities with the test dataset, should improve the results.

When the first stage returns an image snippet that contains a container, the second stage produces reliably results for the masks. The second model produces results that differ on average 22 percent from the ground truth value. Although the standard variation is also 0.18. This is around 10 percent better then a random prediction could have done. When sampling two random values between 0 and 1 and calculating the *mae* between those values the mean absolute error would be 33 percent. The results appears different, if it is compared to the case, that the prediction would always be 0.5. If then enough samples of fill heights would be collected, the *mae* between the constant predicted fill height and the varying samples would lie automatically at 0.25. Compared to this result the performance of the prediction looks very weak as well.

In chapter: 4.5 it was discussed, that the synthetic dataset had similarities compared to the test dataset using color histograms and *PCA*. The second model also creates similar results when directly feed with the test dataset or the output of the first stage. This shows that the second model is more robust to small fluctuations.

Also the influence of color, transparency and fill height on the absolute fill height error were given. As expected less transparent liquids and larger amounts of liquids are less error prawn. What was surprising is that the transparency of the container made little to no influence on the absolute error. What a clear downside on the second model is, is that it predicts also fill heights for *false predictions*, that a possible first stage feed him. This makes it a generally more unreliable. The validation loss for the fill height was: 0.12. This is much better then the result on the test dataset. This must lead to the conclusion, that the framework can learn to analyze the fill height. As it is now, it fails though, to transfer that understanding to the real dataset.

All though a great amount of the architectural components are shared between the first and the second stage, the second stage produces more reliable mask predictions. This could be a further hint, that the synthetic dataset for the first stage, was not well designed to produce reliable results on the real test dataset.
The results of the *Dice Loss* also makes it visible, that a better mask prediction for the liquid and container results in a better fill height prediction. The inheritance of errors from a stages process can also be named as a weakness.

7 Conclusion and Future Work

It was the goal of this thesis to train a hypothetical robot to understand liquid fill hights in a kitchen like environment. As interesting containers: cups, glasses and bottles of different size, color and structure were chosen. The idea was to use a cameras RGB images, as the input for a model, that can derive the necessary information. The main focus here, was lying on creating a synthetic dataset as well as a real dataset and training a two staged model to first detect the container on a table scene and then analyze the detected container to predict its fill height.

The dataset that was created, using *Blender* and *Blenderproc*. For *Blender* a script was developed that can produce *Blender models* of the above mentioned containers. The created models can vary in shape, hight, width, fill height as well as color and more to represent a variety of possible containers.
Overall 55 *Blender models* were created using this program as well as three additional handmade models. Each with a different fill heights. All those models were used in *Blenderproc* to create a meaningful synthetic dataset. To again achieve a large variety, the container models as well as clutter objects were distributed randomly on different table models and placed inside a room to create different render images. With the help of *Blenderproc* not only the RGB images were render, but also the masks necessary to train a *convolution neural network* to solve the task. A *COCOA*-style dataset was created that contained the visible, amodal and invisible mask for each container and liquid model. The *COCOA*-styled dataset also contained information about the bounding boxes, fill height, transparency and more. The *COCOA*-styled dataset consists of 1000 scenes with maximum five containers per scene.
The real dataset was created by taking images with a real camera and labeling the images by hand. Doing this a independent test dataset, with 80 scenes was created, with whom the reliability of the created model was going to be tested.

To solve the specific tasks of detecting the containers on a cluttered table scene and then analyzing its fill heigh, the *COCOA*-styled dataset had to be further adjusted. The dataset for the detection stage (first stage) contained the whole scene as well as the visibility mask of the container. The second stage, where the fill hight was predicted, contained only image snippets of the container, the compatible amodal mask for the container and the liquid as well as the transparency value of the container and liquid and the fill height. Both datasets were also augmented to increase the robustness. Also 20 percent of the test dataset are moved from the test dataset to the train and validation dataset.

Both *neural nets* are based on well known architectural components like: pre trained backbones, short cuts and an encoder-decoder structure. The second stages *neural net* features a multi-head-architecture, which predicts the amodal mask, transparency and fill hight at the same time. Both *neural nets* stay below the parameter guideline of the *One-in-ten-Rule*.

The prediction results on the test dataset of the first stage are weak (the average *1-Dice-Loss*=0.66), probably due to a to big difference between the synthetic train

dataset and the test dataset. The test dataset features almost exclusively bright scenes, with a white table as well es walls. The synthetic dataset has a more variant color scheme. Also compared with the second dataset, the first dataset is ten times smaller, although the scene is more complex.
The second stage, that actually predicts the fill height value, produces reliable values, with a average error of 22 percent on the test dataset. That is around 10 percent better then random predictions but only 3 percent better, then a prediction, that is always 0.5. The fill height error highly correlate to the *Dice-Loss* of the container and the Liquid. This inheritance of errors is a weakness of the model.

The results still show that synthetic datasets are a good source for data hungry *neural networks*. When designed carefully the abstraction between the two domains can overcome the reality gap. But the result also shows that creating helpful datasets is a challenging task.

As stated in chapter: 2.5, similar research were done multiple times. But often the researched task, handled a fixed camera setup, with far less possible variability in the scene. Those proposed solutions, does not need to deal with different view angels and different container shapes. To better account for those additional challenges, some future improvements could be done. For future work, the training could be repeated with a larger dataset, to increase the robustness against diverse view angels. Also a modified train dataset could be created, that features more white color tones and setups with larger tables. But also a improved test dataset could be created, that features more similarities to the train dataset. Additionally a confidence value could be added, that expresses the models confidence, in how correct the predicted fill height value is. What could also be a future improvement, is that the last predicted value, also enters the second stage again, to build up confidence over time.

References

[1] Martin Abadi, Ashish Agarwal, and Paul Barham et. al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Inc Blender. The freedom to create, 2025. `https://www.blender.org/about/` [Accessed: (03.01.2025)].

[3] B. Chatchapol, W. Autanan, and W. Anan. Liquid filling quality control system for beverage industry using image processing (CNN). In *2024 24th International Conference on Control, Automation and Systems (ICCAS)*, pages 1434–1439, 2024.

[4] Maximilian Denninger, Martin Sundermeyer, Dominik Winkelbauer, Dmitry Olefir, Tomas Hodan, Youssef Zidan, Mohamad Elbadrawy, Markus Knauer, Harinandan Katam, and Ahsan Lodhi. Blenderproc: Reducing the reality gap with photorealistic rendering. In *16th Robotics: Science and Systems, RSS 2020, Workshops*, Juli 2020. Video presentation: https://www.youtube.com/watch?v=tQ59iGVnJWM.

[5] GIMP. The free and open source image editor, 2025. `https://www.gimp.org/` [Accessed: (03.01.2025)].

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[9] Inc Intel. Intel realsensetm product family d400 series. Technical Report 337029-017, Intel, 2022. Accessed: 2025-01-03.

[10] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[11] Mohssen Mohammed, Muhammad Badruddin Khan, and Eihab Bashier Mohammed Bashier. *Machine Learning: Algorithms and Applications.* CRC Press, Boca Raton, FL, 2016.

[12] Kunal Pithadiya, Chintan Modi, and Jayesh Chauhan. Machine vision based liquid level inspection system using isef edge detection technique. pages 601–605, 02 2010.

[13] Benjamin Planche and Eliot Andres. *Hands-On Computer Vision with TensorFlow 2: Leverage deep learning to create powerful image processing apps with TensorFlow 2.0 and Keras.* 05 2019.

[14] DLR RMC. Blenderproc2, 2024. https://dlr-rm.github.io/BlenderProc/ [Accessed: (03.01.2025)].

[15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[16] J. A. Syed, A. A. Takriti, and R. R. Damindarov. Computer vision and deep learning enabled real-time liquid level detection and measurement in transparent containers. In *2024 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)*, pages 711–716, 2024.

[17] Blender Developer Documentation Team. Blender 4.2 reference manual, 2025. https://docs.blender.org/manual/en/4.2/ [Accessed: (27.05.2025)].

[18] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[19] Leila Yazdi, Anton Satria Prabuwono, and Ehsan Golkar. Feature extraction algorithm for fill level and cap inspection in bottling machine. In *2011 International Conference on Pattern Analysis and Intelligence Robotics*, volume 1, pages 47–52, 2011.

[20] Yan Zhu, Yuandong Tian, Dimitris N. Metaxas, and Piotr Dollár. Semantic amodal segmentation, 2015.

Universität
Bremen

Nachname	**David**	Matrikelnr.	**6216287**
Vorname	**Antonius**		

Hinweise zu den offiziellen Erklärungen

1. Die folgende Seite mit den offiziellen Erklärungen

   A) Eigenständigkeitserklärung

   B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten

   C) Einverständniserklärung über die Bereitstellung und Nutzung der Bachelorarbeit / Masterarbeit in elektronischer Form zur Überprüfung durch eine Plagiatssoftware

   ist entweder direkt in jedes Exemplar der Bachelor- oder Masterarbeit fest mit einzubinden oder unverändert im Wortlaut in jedes Exemplar der Bachelor- oder Masterarbeit zu übernehmen.

   Bitte achten Sie darauf, jede Erklärung in allen drei Exemplaren der Arbeit zu unterschreiben.

2. In der digitalen Fassung kann auf die Unterschrift verzichtet werden. Die Angaben und Entscheidungen müssen jedoch enthalten sein.

Zu B)

Die Einwilligung kann jederzeit durch Erklärung gegenüber der Universität Bremen, mit Wirkung für die Zukunft, widerrufen werden.

Zu C)

Das Einverständnis der dauerhaften Speicherung des Textes ist freiwillig.

Die Einwilligung kann jederzeit durch Erklärung gegenüber der Universität Bremen, mit Wirkung für die Zukunft, widerrufen werden.

Weitere Informationen zur Überprüfung von schriftlichen Arbeiten durch die Plagiatsoftware sind im Nutzungs- und Datenschutzkonzept enthalten. Diese finden Sie auf der Internetseite der Universität Bremen.

Universität Bremen

Nachname	**David**	Matrikelnr.	**6216287**
Vorname	**Antonius**		

A) Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet, dazu zählen auch KI-basierte Anwendungen oder Werkzeuge. Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht. Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

☐ Ich habe KI-basierte Anwendungen und/oder Werkzeuge genutzt und diese im Anhang "Nutzung KI basierte Anwendungen" dokumentiert.

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.

☒ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☒ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

C) Einverständniserklärung zur Überprüfung der elektronischen Fassung der Bachelorarbeit / Masterarbeit durch Plagiatssoftware

Eingereichte Arbeiten können nach § 18 des Allgemeinen Teil der Bachelor- bzw. der Masterprüfungsordnungen der Universität Bremen mit qualifizierter Software auf Plagiatsvorwürfe untersucht werden.
Zum Zweck der Überprüfung auf Plagiate erfolgt das Hochladen auf den Server der von der Universität Bremen aktuell genutzten Plagiatssoftware.

☑ Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum oben genannten Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

☑ Ich bin nicht damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

Mit meiner Unterschrift versichere ich, dass ich die obenstehenden Erklärungen gelesen und verstanden habe und bestätige die Richtigkeit der gemachten Angaben.

26.05.2025 [illegible]

Unterschrift

Appendix

```
{"general_para":{"handle":random.choice([True, False]),"label":random.choice([
    True, False]),"variation":random.choice([True, False]), "liquid": random.
    choice([True]),"bottle_cap":random.choice([True, False])},
"cup_geom_para":{"hight":random.uniform(0.08, 0.25), "diameter": random.uniform
    (0.08, 0.15),"pitch":random.uniform(0.5, 0.9),"num_verts_vertical":16,
             "num_cuts_horizontal":16,"bottom_scale_in":random.uniform(0.5,
                 0.9),"bottom_scale_up":random.uniform(0.2, 0.4),
             "lip_scale_out":1.01,"lip_extrude":-0.001,"cup_thickness":random.
                 uniform(0.001, 0.008),
             "bottom_thickness":random.uniform(0.004, 0.01),"
                 cleaning_mesh_threshold":0.0001},
"cup_texture_para":{"bsdf_IOR":1.5,"bsdf_metallic":0,"bsdf_transmission":0,"
    bsdf_RGBA":(1, 1, 1, 1),
                 "bsdf_roughness":0,"va_RGBA":(0.396185, 0.286895, 0.214343, 1),
                 "va_density":5},
"handle_geom_para":{"extrude":random.uniform(0.05, 0.25),"bevel_depth":random.
    uniform(0.1, 0.2),"bevel_res":4,"resize_bez_x":random.uniform(0.5, 1.5),
                 "resize_bez_y":random.uniform(0.5, 1.5),"resize_open_handle_y"
                     :1.3,"resize_open_handle_z":1.3,
                 "handle_scale_para":0.2},
"handle_texture_para":{"bsdf_IOR":1.5,"bsdf_transmission":1,
                     "bsdf_RGBA":(0.820774, 0.800408, 0.208576, 1),"
                         bsdf_roughness":0,
                     "va_RGBA":(0.396185, 0.286895, 0.214343, 1),"va_density"
                         :0},
"label_texture_para":{"backround_color_RGBA":(random.uniform(0, 1), random.
    uniform(0, 1), random.uniform(0, 1), 1),
                 "highlight_0_color_RGBA":(random.uniform(0, 1), random.uniform
                     (0, 1), random.uniform(0, 1), 1),
                 "highlight_I_color_RGBA":(random.uniform(0, 1), random.uniform
                     (0, 1), random.uniform(0, 1), 1),
                 "struct_scale":random.uniform(1, 10), "struct_detail":random.
                     uniform(1, 1.5),"struct_roughness":random.uniform(1, 3),
                 "struct_random":random.uniform(0, 1),"mapping_x":random.uniform
                     (0.6,2.6),"mapping_y":random.uniform(-1,1),"mapping_z":-1},
"lattice_geom_para":{"variation_low_bound":self.lattice_para[0][0],"
    variation_high_bound":self.lattice_para[0][1],
             "variation_scale":self.lattice_para[1]},
"liquid_geom_para":{"fill_hight":random.uniform(0, 1)},
"liquid_texture_para":{"bsdf_IOR":1.33,"bsdf_transmission":1, "bsdf_RGBA"
    :(1,1,1,1),
                     "va_RGBA":liquid_color,"va_density":random.uniform(100,
                         1000)},
"cap_geom_texture_para":{"bsdf_RGBA":(0.396185, 0.286895, 0.214343, 1)},
"random_seed":0,
"cup_type": self.cup_type
}
```