DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Photorealistic Rendering of Training Data for Object Detection and Pose Estimation with a Physics Engine

Alexander Epple

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Photorealistic Rendering of Training Data for Object Detection and Pose Estimation with a Physics Engine

Fotorealistisches Rendern von Trainingsdaten für Objekterkennung und Posenabschätzung mit Hilfe einer Physiksimulation

Author:	Alexander Epple
Supervisor:	Prof. Dr. Nassir Navab
Advisor:	Fabian Manhardt, M. Sc.
Submission Date:	15.10.2020

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.10.2020 Alexander Epple

Abstract

Reliable Object Detection and Pose Estimation has consistently been a problem for researchers and developers within fields like autonomous driving and robotic surgery. While robots and Computer Vision are no novelty in environments like factories, where everything is predictable and deterministic, there is still a lot of research to be done when it comes to real, every day scenarios. A more sophisticated technological approach is necessary when dealing with the diversity and unpredictability of the real world.

Pose Estimation uses Deep Learning and specifically Convolutional Neural Networks (CNNs) to handle this complexity. Well-trained networks are capable of transferring their learned abilities into new scenarios, they can *generalize* to the real world. However, a new problem arises when taking into account the immense amount of training data need. With real training data being both difficult to generate and annotate, this problem is even more pronounced.

Synthetically generated data has been explored as a possible solution. Synthetic training data eliminates the challenges and costs of annotation. It can be produced in sufficient quantities while maintaining negligible costs.

While synthetic training data may sound like the obvious solution due to its advantages, there are also problems that need to be solved. Pose Estimation networks that are trained only using synthetic data have a tendency to overfit to those synthetic data sets, thus not performing well in real world scenarios. When real, captured data, such as photographs or 3D scans are used, illumination needs to match the original conditions to generate realistic synthetic images, which is a difficult problem in of itself. Thus, the main challenges are rendering realistic and plausible images, incorporating accurate estimates of the real light conditions, at acceptable computational cost.

In order to address these challenges, we use real photographs and room scans as the basis for our synthetic data set generator. The scans are used both for simulating the rigid-body physics of the virtual objects correctly, as well as to improve rendering quality. The photographs and renders are blended, combining the realism of captured images with the infinite variety simulated objects provide. To keep computational costs low, we render the objects without background. The synthetic parts of the image appear realistic, as the scene is incorporated into rendering. With the images appearing natural, the the risk of overfitting is reduced as well.

Contents

1 Introduction

Since the beginnings of Artificial Intelligence, a lot of research has been done in the field of Computer Vision. There are many applications where accurate Object Detection or Pose Estimation are not only necessary, but required. Deep Learning (DL) algorithms designed to solve these problems only work reliably, if large amounts of diverse, high quality training data is available. Thus, one of the biggest challenges in this problem space of computer vision is the acquisition of training data.

Pose Estimation, which is used to infer positions and rotations of known objects from images, is a DL problem, which requires a lot of high quality, annotated data for training. Since Pose Estimation is often used in real world scenarios, a trained network needs to be able to generalize to unseen, real world scenarios.

Hence, in this thesis we will propose a novel approach to generate synthetic image data sets for Pose Estimation. This approach combines the advantages of synthetic generators, from which data can easily be obtained, and the realism of real world data to fit the data requirements of Pose Estimation.

1.1 Motivation

With the advent of technologies like autonomous driving, robotic surgery and the steadily increasing usage of robots in hazardous industrial applications, the need for reliable object detection and pose estimation has long since arrived. While robots and computer vision are no novelty in predictable, reasonably deterministic environments such as car manufacturing plants, the diversity and unpredictability of real world scenarios requires more sophisticated technology. Self-driving cars, for example, have to be able to detect the pose of obstacles, and most importantly humans, at any time of day and in any weather condition.

Pose Estimation uses DL and Convolutional Neural Networks (CNNs) to handle the complexity of the real world. While well-trained networks are capable of this, a new problem, the requirement of large amounts of training data, arises. Not only is a lot of realistic, annotated data necessary, the training sets also need to be varied enough to be able to generalize to previously unknown inputs. This problem is especially pronounced by the fact that real training data is both hard to generate and to annotate.

Considering those downsides, synthetically generated data has been explored as a possible solution. Synthetic training data can be produced in sufficient quantities at negligible costs and eliminates the challenges of annotation. The generated data has enough variation to train a model that can generalize to the real world. With those benefits in mind, we propose a generator pipeline that can produce photo-realistic, physically accurate training data fully synthetically.

1.2 Challenges

Synthetic training data has a lot of advantages, but there are many challenges and problems to solve here as well. CNNs that are trained using synthetic data only can be over-fitted and may not perform well in real world scenarios. Overcoming the obstacle that is the domain gap between real world and synthetically generated images is therefore one of the main focus points of the proposed approach. If training data is generated entirely synthetically, the computational cost of both rendering and simulating is very high and a limiting factor.

When real, captured data such as photographs or 3D scans are used, illumination needs to match the original conditions to render accurate and realistic synthetic images. Illumination and lighting generally is integrated, however, which makes a full reconstruction impossible. The main challenges are, consequently, rendering realistic and plausible images and incorporating accurate estimates of the real lighting conditions, at a low computational cost.

1.3 Approach

To tackle those challenges, we use real photographs that were used to create 3D room scans as well as those 3D scans as the base of the generation pipeline. The scans are used both for simulating the rigid-body physics of the synthetic objects correctly as well as to improve the ray-traced rendering of those objects. The photographs and renders are then blended, combining the realism of captured images with the almost infinite variety that simulated objects enable.

Rendering only part of the image by not rendering the scene itself keeps the computational cost low without a reduction in rendering quality. By integrating the 3D scan into rendering and simulation, the synthetic parts of the image appear natural, which reduces the risk of over-fitting. Finally, to evaluate the effectiveness and performance of training data generated using the proposed approach, Mask R-CNN [13] is trained with only our synthetic images and compared to a network which uses an OpenGL baseline data set.

2 Background

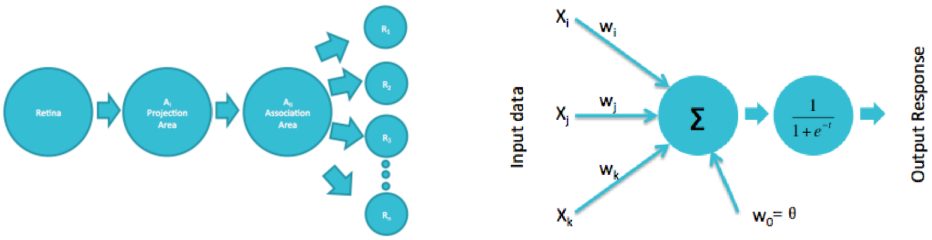
The field of machine learning is steadily growing in both importance and size. In order to understand the importance of high quality training data, a brief history and summary of the concepts of deep learning is given in 2.1. An overview of pose estimation and its training data requirements follows in 2.2, as it is the foundation of this thesis.

2.1 Deep Learning

Donald O. Hebb's famous rule, "Cells that fire together, wire together" [1], also known as the *Hebbian Learning Rule*, is often seen as laying the foundation of neural networks. Simply put, the more often two nodes interact, the stronger their connection should get [29]. Here, the nodes are analogue to biological neurons and the connections between them can be compared to axons and dendrites in biological neural networks[6].

2.1.1 Building Blocks

The Hebbian Learning Rule lead to the development of the perceptron by Frank Rosenblatt in 1958. This machine learning algorithm tries to find a separation between different classes based on input features. The original concept uses a step function instead of probabilities; each neuron either fires or stays inactive [6].



(a) Original perceptron model

(b) A modern perceptron

Figure 2.1: The original perceptron model and an example of how it is used today [29]

[1] *The Organization of Behaviour*, Donald O. Hebb, 1949

A perceptron is roughly modeled after biological neurons and split into four units as depicted in figure 2.1a. Stimuli arriving on the retina are transmitted to the projection area, where impulse strength is based on the stimulus intensity. The impulses are passed on to the association unit through random connections, which, in return fires if the sum of all impulses reach a certain threshold. This result is then consumed by the response units, which function similarly.

Even though perceptrons today differ from the original design, the fundamental concept has not changed and perceptrons are still used in modern neural networks today. The key differences are the introduction of weighted sums in the association unit, the omission of the projection area and the use of non-linear activation functions [29]. Figure 2.1b shows an example of a modern perceptron with a sigmoid activation function.

2.1.2 Activation Functions

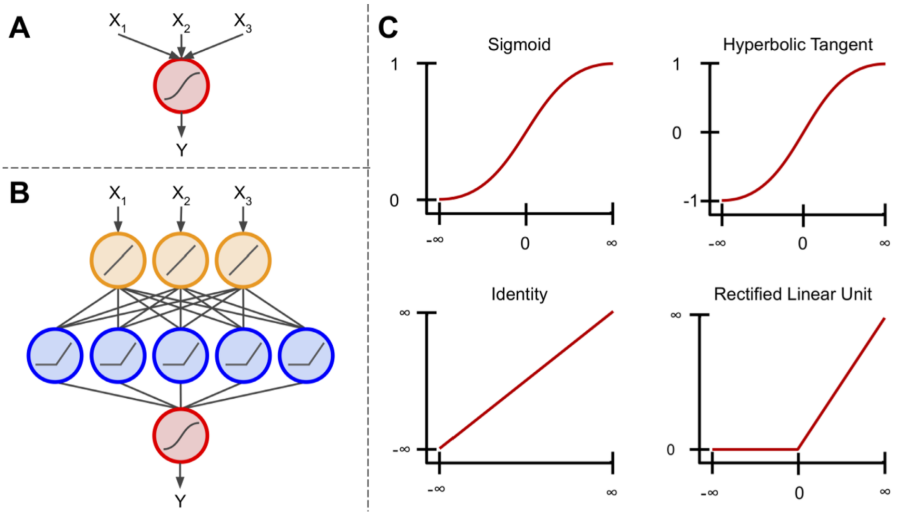


Figure 2.2: Overview of various activation functions (C), a single perceptron (A) and a simple feed-forward neural network (B) [6]

The aforementioned activation functions scale and map the inputs to outputs. In the case of the original perceptron, a step function was used, which transformed inputs to either exactly 1 or 0. Nowadays, popular functions are often non-linear and can produce non-binary outputs. Examples are depicted in figure 2.2. The rectified linear unit (ReLU), for example, can be used to remove negative inputs [6].

2.1.3 Feed-forward Neural Networks

A single perceptron alone is not a network, since an Artificial Neural Network (ANN) requires multiple layers of connected neurons. ANNs usually contain both an input and output layer as well as several *hidden layers* in between.

If there are no loops in the network, meaning all inputs are only passed forward towards the output nodes, the network is referred to as a feed-forward neural network. Each layer in the ANN modifies its inputs and eventually the output nodes provide, for example, the desired classification [6]. Figure 2.2 shows a simple feed-forward ANN with an input, output and hidden layer each.

To train a network, the input weights of the neurons are adjusted via *back-propagation*, although not all weights are necessarily adjusted each iteration. This process gets increasingly more difficult for deep networks with many layers, as the gradient descent can get stuck before reaching the first layers [4].

2.1.4 Convolutional Neural Networks

In image recognition tasks, a special case of ANNs called Convolutional Neural Networks (CNN) is often used, as traditional ANNs do not preserve spatial context. This network uses patches of images instead of single pixels, which prevents the loss of pixel relationships [6].

CNNs were inspired by Neocognitron, which was developed in 1980 by Kunihiko Fukushima. This network used two kinds of layers, one to extract features and one to organize the extracted features. This is very similar to human vision [29].

Modern CNNs also use two kinds of layers, namely, convolution and pooling layers. Convolutions are matrix operations that combine several pixels into one output. Examples for convolutions are edge detection and blur filters. Deeper layers, however, are usually more abstract. The other type are pooling layers, where features can be combined in a non-linear way [4]. Pooling reduces the amount of inputs by down-sampling the feature map. Examples are taking the average or maximum of a subset of pixels.

Finally, after many layers of convolutions and pooling, the compressed output is consumed by an ANN that is called Fully Connected layer (FC). The FC outputs the classification based on the extracted features. During training, both classification as well as feature extraction can be trained, since the matrix values of each convolution and the neuron weights of the FC can be adjusted during back-propagation [6].

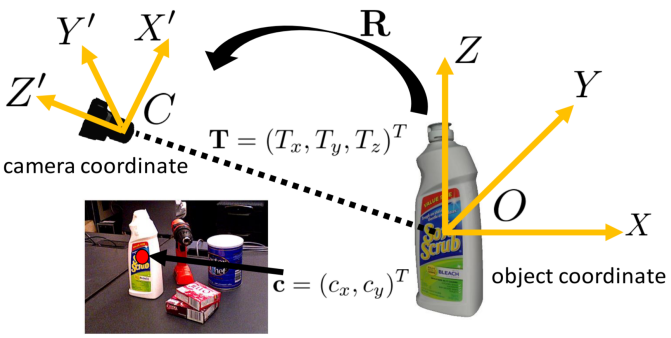


Figure 2.3: For a given image, rotation (R) and translation (T) of a visible 3D object (O) are calculated with a camera (C) as the origin. This is done by corresponding a a pixel (c) to a local point on the 3D object. [31]

2.2 Pose Estimation

In the fields of computer vision and computer graphics, a pose is usually understood as a combined position and rotation in space. This can be expressed as matrices in the form of 2.1, consisting of a 3x3 rotation matrix (R) and a translation vector (t), which denotes the relative distance to a given origin. In literature this is often referred to as an object having 6 degrees of freedom (DoF).

$$\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{2.1}$$

While a pose exists in three-dimensional (3D) space, usually only two-dimensional (2D) inputs in the form of images are available. In some cases, additional depth images are captured, which elevates the input data to 3D (from a fixed view point). Although there are several pose estimation problems, the one relevant to this thesis is the problem of associating 2D perspective-projected 3D data with another separate 3D data set. This is referred to as the *absolute orientation problem* [12].

The absolute orientation problem, which is also known as the *exterior orientation problem* in the field of photogrammetry, consists of estimating the unknown rotation and translation of a known 3D objects from 2D image points [12].

There are many possible approaches that could provide a solution to this problem, including PoseCNN [31] and BB8 [25]. Both PoseCNN and BB8 use CNNs to infer 6 DOF poses from 2D RGB images alone. Figure 2.3 illustrates the process of localizing pixels on an object and consequently its camera relative pose.

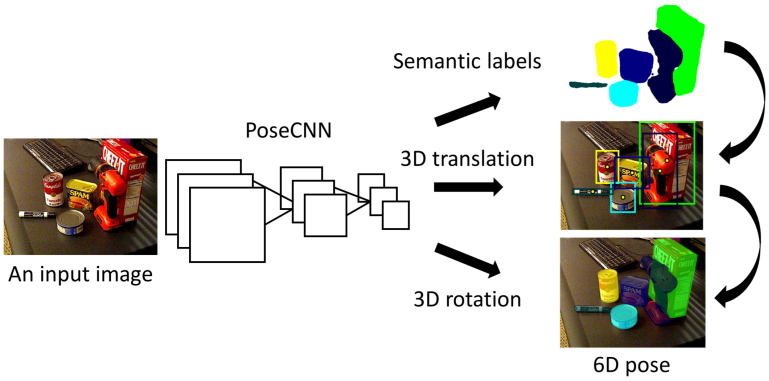


Figure 2.4: Overview of PoseCNN, its inputs and outputs [31]

To show how pose estimation may be implemented, PoseCNN is briefly summarized as an example. The network first classifies each pixel and estimates the 2D center and distance to each object. This information is then used in combination with the camera intrinsics to further estimate the 3D translation. In a final step, PoseCNN recovers the 3D rotation of an object using convolutions of pixels in the associated bounding box [31]. Figure 2.4 shows the various steps of the process and its intermediate outputs for one exemplary image.

2.3 Training Data

Every ANN needs to be trained in order to function, which in return requires annotated data. Quality, diversity and quantity all play a role in how well the network will eventually perform. In the case of pose estimation, the pose of every object in each frame of a data set must be annotated [19]. Broadly speaking, there are two ways to acquire training data: manually annotating real scenes or generating images synthetically.

2.3.1 Real Data Sets

Data sets consisting of purely real scenes and images are scarce. To show how real data sets can be generated and to discuss the challenges of creating them, HomebrewedDB [19] is used as an example. This data set consists of roughly 35k annotated images of varying degrees of complexity. It was created using both a structured light camera and a Microsoft Kinect 2, with pose annotations calculated from depth information.

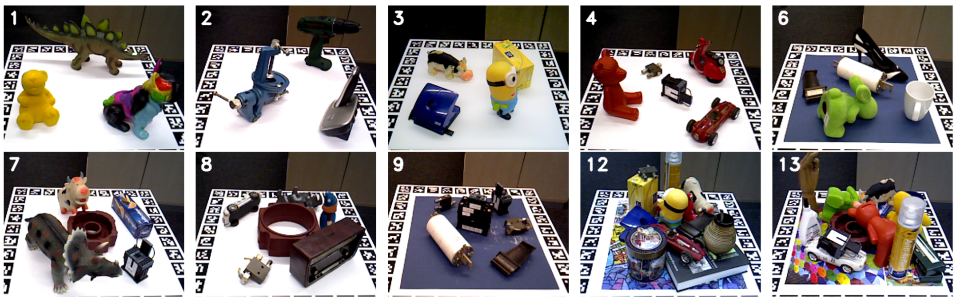


Figure 2.5: Images of varying complexity from HomebrewedDB [19]

Not only do Kaskman *et al.* [19] note that real data sets usually only contain a small number of objects and therefore do not scale well, but they also state that this kind of training data can often lead to overfitting. This stems from data sets being restricted to particular scenarios, such as household or industrial objects, as well as test and training data being to similar for a comprehensive assessment. It is also pointed out that real data sets do not address scene illumination, such as varying light color and intensity or even the introduction of additional lights.

Dataset	Method	Benchvise	Phone	Driller
LM	YOLO6D	81.80	47.74	63.51
	DPOD	95.34	74.24	97.72
HB	YOLO6D	15.30	6.50	0.10
	DPOD	57.24	33.09	62.82

Table 2.1: Detection accuracy of specific objects, when testing on sequences from the training data compared to HomebrewedDB [19]

Another drawback of using real training data is that networks are unable to generalize to new scenarios. This problem is illustrated in table 2.1, where two state of the art methods trained on the LineMOD data set were tested on a HomebrewedDB sequence. Even though the example is not complex, both networks experienced a significant drop in detection accuracy compared to the sequence they were trained on. This performance hit likely stems from overfitting to a particular data set [19].

2.3.2 Synthetic Data Sets

The many disadvantages of real training data make synthetic data sets more appealing, hence there is a need for data generators. Generally speaking, synthetic images can be created by altering existing real sequences or by rendering completely new ones using 3D models of objects and scenes. Fully synthetic approaches come with problems of their own, however, which will be laid out in more detail in chapter 3 and 5.

3 Related Work

Synthetic image generation is a hot topic issue and a big part of our approach. An overview of two different methods and corresponding implementations will be discussed in section 3.1.

When altering real images or rendering fully artificial ones, some level of realism will be lost. This loss of realism has led to a high level of research in the fields of inverse rendering, light estimation and image blending to help mitigate this problem.

In this thesis, part of the conducted project was dedicated towards improving the realism of the generated images, therefore some state of the art approaches in those fields are also highlighted in 3.2.

3.1 Generating Synthetic Image

When it comes to synthetically generating large amounts of diverse training sets, there are at least two distinct approaches: Modifying existing images and creating artificial ones. Both methods can potentially produce infinite amounts of images with a large contextual variety. However, they approach the task of minimizing the domain gap between synthetic and real images from different angles. There are also different types of generators; some rely on ANNs for augmentation while others render images fully artificially.

3.1.1 Network Driven Generation

Generators that modify data sets to produce new ones often rely on neural networks. Popular approaches include domain randomization [26] [34], where source data is randomly altered, and domain adaption [5] [3], which finds mappings or similarities between the source and target domains. One important difference is that domain randomization is independent of the target domain, while domain adaption is not. Where domain adaption strives to identify differences between source and target data, domain randomization attempts to make the network believe that the real world is just yet another variation.

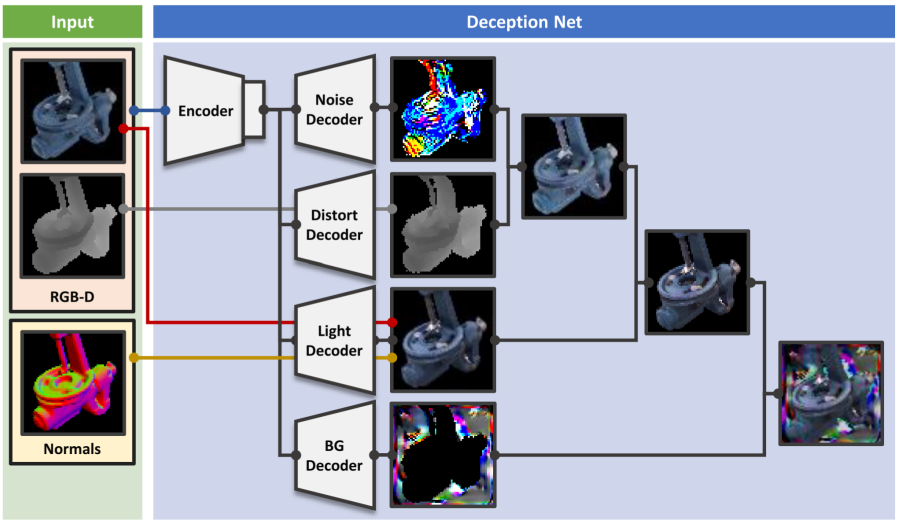


Figure 3.1: Overview over the deception network modules [34]

Domain Randomization

Zakharov *et al.* [34] use an encoder / decoder network, named *Deception Net*, to randomly perturb source images. Another network, called *Recognition Net*, is tasked with both determining the loss and recognizing the alterations. Training consists of two phases: one of the two ANNs is trained, while the other is frozen. The goal of Deception Net is to generate progressively more confusing images, as its loss function is inverse to the loss of the recognition network. Recognition Net, on the other hand, becomes increasingly more resistant to randomization. This is similar to how generative adversarial networks (GAN) are trained and function.

The deception network has several options (*modules*), depicted in figure 3.1, to augment its input. Source images have black backgrounds, hence why a background module fills the empty space through upsampling and convolutions with random, but complex content. There are also two modules dedicated towards deforming and distorting the source image as well as adding noise to it. Most notably, there is a light module that controls illumination based on the Phong model.

Tobin *et al.* [26] train an object detector purely on simulated images. The images are low-fidelity renders with randomized camera, illumination and object parameters. The detection works well enough that a grasping robot could successfully pick up targets in 38 out of 40 attempts, including cluttered setups. It is also noted, that pre-training only seems to be beneficial for performance if less training data is used. With large amounts of data, generalizing to real world scenarios appears to be possible regardless of whether the network was pre-trained or not.

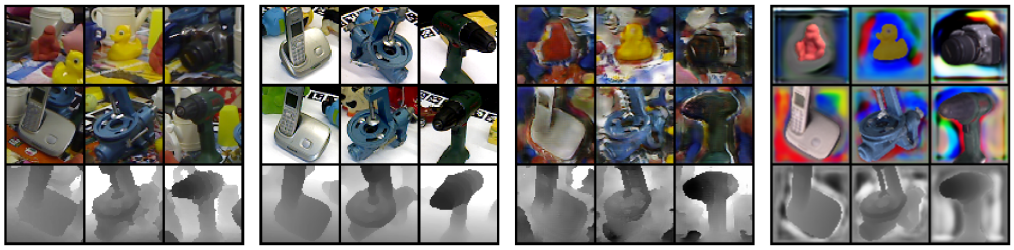


Figure 3.2: Exemplary images from the LineMOD sequence, an extended sequence and generated outputs from PixelDA & DeceptionNet (left to right) [34]

Domain Adaption

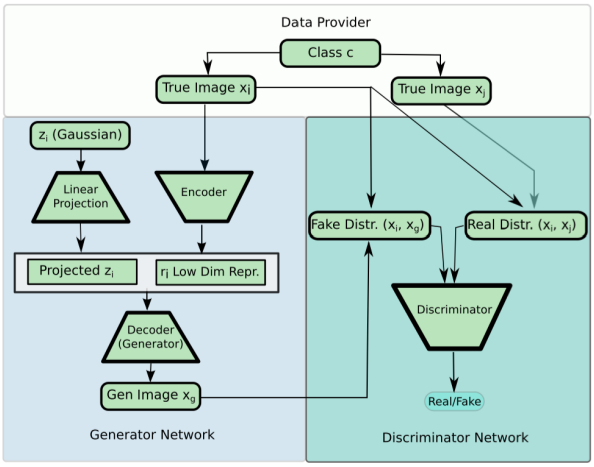
Many domain adaption techniques utilize GANs [5, 21, 32] to produce images that appear as if they were part of the target domain. In the case of PixelDA [5], Bousmalis *et al.* train a network to produce images similar to the ones in the target domain by adapting them from source images and noise vectors as the inputs. The generated images and samples from the target domain are then consumed by a discriminator, which determines whether the input is "real" or "fake". Additionally, a task-specific classifier is used for labeling. The architecture is illustrated in figure 3.3b.

$$\min_{\Theta_G,\Theta_T} \max_{\Theta_D} \alpha\mathcal{L}_d(D,G) + \beta\mathcal{L}_t(T,G) + \gamma\mathcal{L}_c(G) \tag{3.1}$$

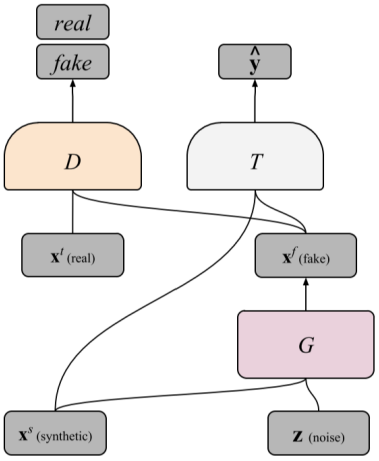
The minimax objective 3.1 of the GAN from [5] is split into three parts: The domain loss $\mathcal{L}_d$, a task-specific loss $\mathcal{L}_t$ and a content-similarity loss $\mathcal{L}_c$, which discourages generated images to be too different from the source data.

Antoniou *et al.* [3] propose a similar, GAN-based approach, which augments images by learning how to map out a data manifold. This is especially useful to achieve better performance in low-data settings. Figure 3.3a provides an overview over the network architecture. In the first step, an encoder converts the input to a lower dimensional representation. This, combined with a sample from a random normal distribution, is used by a decoder to generate the augmented images. Finally, a discriminator network is tasked with distinguishing between the "real" and "fake" distributions. This promotes the generation of images that appear just different enough from the source to be considered separate samples.

The discriminator uses either two samples from the source sequence, or one singular sample from the source sequence paired with the augmented output of the generator from that specific source sequence sample. Since no class information is provided, this ensures the GAN generates data that is closely related to the source, regardless of class.



(a) The DAGAN network, consisting of an encoder / decoder generator (left) and a discriminator (right) with access to the source domain [3]



(b) The PixelDA GAN consists of a generator (G), a discriminator (D) and a task-specific classifier (T) [5]

Figure 3.3: An overview of the architectures of PixelDA and DAGAN

3.1.2 Fully Artificial Rendering

Another way to generate synthetic data sets is to render them entirely artificially, without any source images. This approach usually consists of arranging textured 3D models into scenes, which are then captured by a virtual camera in process called *rendering*. The advantage lays in the fact that labeling / classification are seen as trivial tasks, since all information is inherently available. This process can then be easily automated.

Movshovitz-Attias *et al.* [22] discuss the usefulness of rendered training data in their paper, using the example of object viewpoint estimation. A rendered data set, based on 91 high-quality car models, is generated by adjusting various render parameters, including illumination and the cameras physical properties.

Viewpoint estimation accuracy is evaluated by comparing a model trained on the new training data to ones trained on two real data sets. The authors find that the model trained on synthetic data alone outperforms one of the models trained on real data sets, while only marginally falling short of the other one. It is noted that when combining real and synthetic data into one data set, a better performance is achieved than by combining the two real data sets.

Figure 3.4: An excerpt of the RenderCar sequence, at different render quality levels [22]

The importance and effect of render quality is evaluated as well. Figure 3.4 shows three levels of increasingly more sophisticated render results. It is shown that the error decreases and converges faster with improved render quality. Another finding is that increasing the amount of generated data has diminishing returns. It is suggested that using a larger variety of 3D models instead may be a more effective strategy.

The paper [15] by Hodan *et al.* is another example for fully rendered, synthetic training data. They use, among others, 15 3D models from the LineMOD [14] sequence in combination with six hand-crafted, furnished 3D scenes. The lights and spawn boxes for objects, called stages, were manually placed in each scene. Figure 3.5 shows thumbnails of those scenes.

The objects are arranged by instantiating them in air, slightly above the defined stages. After instantiation, a physics simulation is run, which enforces realistic, physically plausible poses. The camera poses are arranged so that at least one object is within the camera frustum and not majorly occluded. Rendering was done on a CPU cluster, featuring 400 16-core processors, using the Autodesk Arnold renderer. Arnold features physically based rendering (PBR), which is considered to be photo realistic.

The authors evaluate how the use of PBR images, differences between PBR quality levels and lacking scene context impact model performance. The evaluation is done using two exemplary object instance detection networks.

Figure 3.5: An overview over the six high-quality scene meshes [15]

Data/Obj. ID	1	5	6	8	9	10	11	12	mAP
Inception-ResNet-v2									
PBR-h	**60.3**	**44.5**	**56.7**	**53.4**	**81.8**	**48.6**	9.6	**92.3**	**55.9**
PBR-l	57.3	35.8	53.3	52.6	77.8	23.8	3.1	94.5	49.8
BL	30.7	45.4	42.5	32.4	77.1	33.4	**19.6**	76.7	44.7
ResNet-101									
PBR-h	**46.3**	**40.3**	**48.5**	**58.0**	**76.4**	**39.5**	4.7	**85.5**	**49.9**
PBR-l	44.1	26.6	41.6	53.7	73.7	24.5	1.1	91.6	44.6
BL	35.5	45.3	37.1	44.6	75.0	33.6	**12.7**	76.8	45.1

Table 3.1: Average, per-class detection precision on LineMOD [14] when trained using high (PBR-h), low (PBR-l) and non-PBR (BL) data [15]

Utilizing PBR images during training rather than ones from a non-PBR data set yields significant improvements. It is also noted, that especially more complex materials benefit from high quality PBR, which features more realistic reflections as well as other improvements. Finally, accurately modelling the scene context also considerably improves performance.

One big drawback of this method is the computational cost. Even on a powerful CPU cluster, rendering time for high-quality images averaged 720s. Considering the performance gains for more complex materials, as can be seen in table 3.1, rendering in low quality may seldom be an option.

While simpler materials only marginally benefit from higher quality rendering, some objects see substantially better detection accuracy. The difference between high and low quality settings is also clearly visible in figure 3.6.

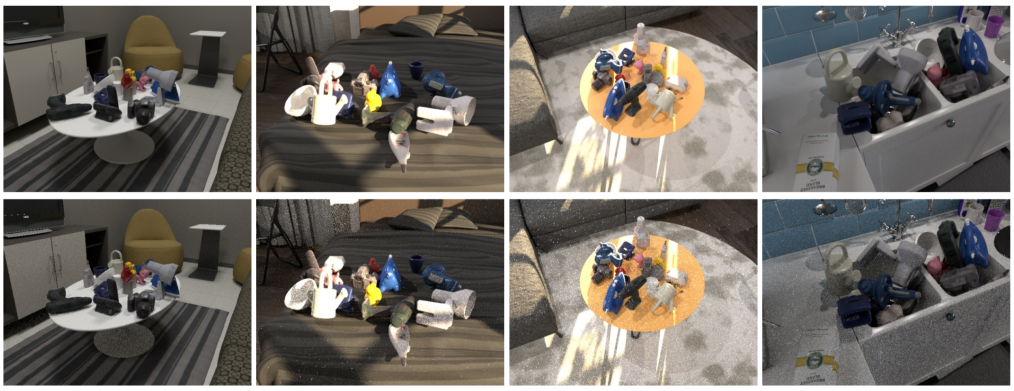


Figure 3.6: Example images rendered in high (top) and low (bottom) quality. Low quality images have more noise and fall behind, especially in dim areas [15]

3.2 Improving Rendering Realism

Models trained on synthetic data sets can be found to be lacking when trying to generalize to the real world. The domain gap between the real and virtual scenes becomes especially apparent if one compares outside factors, such as illumination. This is due to how under-constrained the problem of recovering those parameters from images is [11].

To be able to use real data for generating believable, new sequences, one must approximate, or estimate, said outside factors and incorporate them during or after generation.

3.2.1 Image Compositing

In photography editing, compositing images refers to adjusting a new foreground to a given background. Since the naive approach of pasting the new image onto the background does not take semantic nor contextual information into account, Tsai *et al.* [27] use a deep CNN for image compositing. This process is also referred to as harmonization.

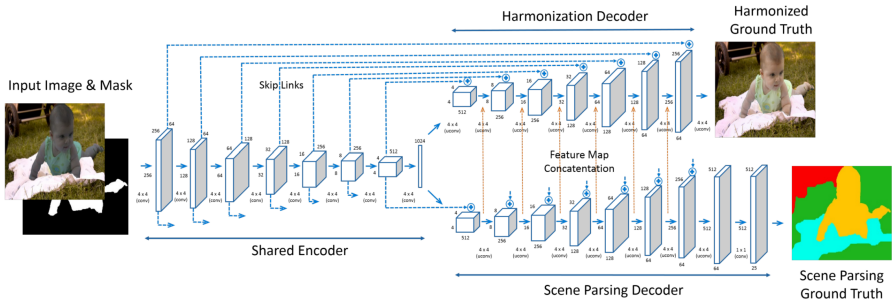


Figure 3.7: The deep image harmonization network architecture, featuring one shared encoder, a reconstruction and a contextual decoder [27]

The harmonization network features an encoder without pooling layers in order to preserve details. The fully-connected layer is then connected to two decoders. The task of the first decoder is to both harmonize and reconstruct the image from its lower-dimensional representation, while the second decoder predicts scene context and semantic information.

All convolutional layers are also connected with skip links, which enable the recovery of details lost due to compression. The contextual information provided by the scene parsing decoder is used by the reconstruction decoder during harmonization. These additional semantic labels promote context-aware compositing and improve the quality of the blended image further. The network architecture and the harmonization in- and outputs are illustrated in figure 3.7.

While image compositing by means of deep image harmonization does provide more visually sound results when compared to the naive approach (*"copy and paste"*), it is not necessarily physically accurate. The goal of compositing is not physical accuracy but rather making the result *look* believable. Considering the importance of both physical realism and contextual accuracy, as stated by Hodan *et al.* [15], models trained on harmonized composites may therefore overfit and not generalize well to real scenarios.

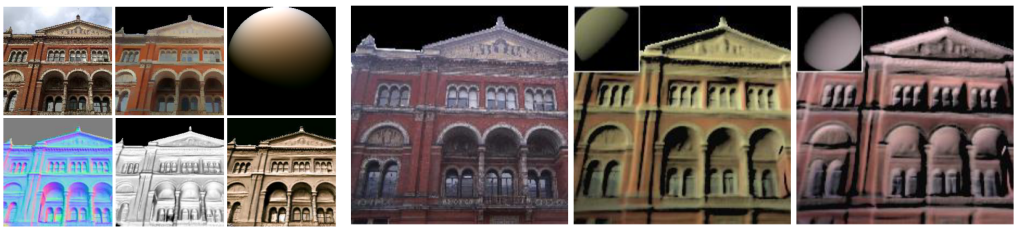
3.2.2 Illumination Estimation

Extracting or estimating light information from low dynamic range (LDR) images is no easy feat. This problem, which is also known as inverse rendering, is a field of active research and has gained a lot of interest due to the arrival of augmented reality (AR). Nowadays, most mobile platforms and smartphones support AR [1]. Considering the task and requirements, many approaches to solve light estimation and inverse rendering make use of CNNs.

InverseRenderNet [33] is a self-supervised CNN that regresses albedo color, surface normals and illumination estimates from an RGB image. During training, scenes reconstructed from image sequences, called multiview stereo (MVS), are used for supervision. MVS provides depth information, which enables the computation of surface normals for guidance.

Illumination can be inferred from the albedo color and normals and is constrained by a statistical model, which is based on real high dynamic range (HDR) environments. The lighting model uses spherical harmonics and supports diffuse light, while effects like reflections and shadows are not considered. An example for the inverse render outputs is depicted in figure 3.8a. Figure 3.8b shows, how those outputs can be used to change the illumination of, or relight, the image from example 3.8a.

[1] e.g. ARKit on iOS, ARCore on Android devices



(a) Input image, albedo color, illumination, normals, as well as frontal and estimated lighting [33]

(b) The original image and relighting using the inverse render outputs [33]

Figure 3.8: InverseRenderNet outputs (left) and the example image re-lit (right)

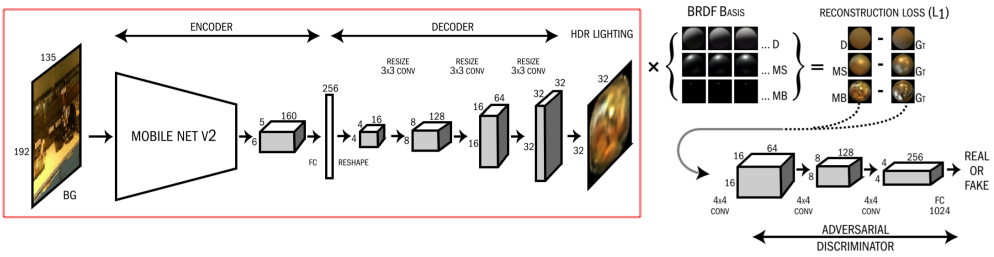


Figure 3.9: The DeepLight network architecture. Only the outlined part is necessary for light estimation [20]

DeepLight [20] is another example for network-driven illumination estimation. The network architecture, as seen in 3.9, is another encoder-decoder CNN. First, a vector representation is regressed from a downscaled LDR image through multiple convolutional layers and one fully-connected layer. The vector is then used to generate a HDR mirror ball mapping, where each pixel represents the illumination of a solid angle.

Figure 3.10: The generated (top) and ground truth (bottom) illumination for three different materials [20]

The generator output enables omnidirectional sampling of light color and intensity, which can be used for image based lighting (IBL). IBL is a global illumination technique, that projects the HDR mirror ball onto an environment, surrounding the objects to be rendered. Since all directions are covered and provide both light intensity and color, rendered objects feature both correct shading and accurate reflections [9]. Example 3.10 shows two IBL environments illuminating spheres with varying materials, comparing the network output to the captured ground truth.

Training of DeepLight uses both the difference between ground truth and relighting (λ_{rec}) and a discriminator network to improve specular reflections (λ_{adv}). The loss function 3.2 thus promotes mirror balls with believable reflections and accurate illumination. Figure 3.11 shows a comparison between real and synthetic objects, which are rendered using the estimated lighting.

$$G* = arg \min_{G} \max_{D} (1 - \lambda_{rec}) \mathcal{L}_{adv} + \lambda_{rec} \mathcal{L}_{rec} \tag{3.2}$$

While IBL is very useful for global illumination, which can be thought of as sophisticated ambient light, it does not model local light sources well, especially in indoor scenarios, where those are predominant.

Figure 3.11: The LDR input, its inferred lighting and a comparison of each a real and rendered object with matte and reflective materials [20]

In order to tackle this problem, Gardner *et al.* propose a method [11] which estimates light sources, rather than overall illumination. The network regresses both a fixed number of light sources and an ambient term from an LDR image. Each light source is defined by its direction, distance, color and size in steradiants.

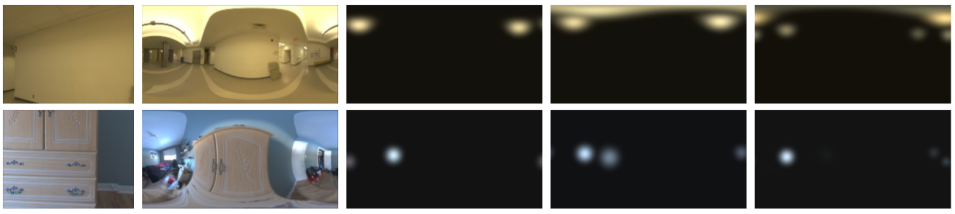


Figure 3.12: The input image, corresponding ground truth and light source estimations when constrained to 2,3 and 5 lights during training [11]

During training, estimated light sources are not directly compared to the ground truth at first, but rather projected onto an environment map. This ensures stability during the early stages, where matching those sets of light sources would normally be near impossible. Example outputs for varying light source counts can be seen in figure 3.12.

Figure 3.13: Light source estimation used for stock images relighting. Each image is shown before and after adding new, virtual objects [11]

After the network has converged sufficiently, it is refined by using the estimated parameters directly to calculate the loss. While the model does not support directional or focused light sources, but rather models all sources as area lights, it can generalize to most scenarios, even ones it was not trained for, such as outdoor scenes. It is also capable of generalizing to stock images, as is evident in figure 3.13. The authors do note, however, that accuracy could be improved further by taking scene appearance into account on top of illumination alone.

4 Physics Simulation

No rendered scene looks realistic if the objects poses are not plausible. Our method wants to mimic the real world as accurately as possible, which is why our application uses Nvidia PhysX [23], similar to Hodan *et al.* in their approach.

Nvidia PhysX is a physics engine which offers, among other things, fast and accurate rigid body interaction simulation. We use a range of features provided by the framework, described in sections 4.1 and 4.2, to speed up the collision computations as well as improve the plausibility and realism of the final poses.

4.1 Geometry & Rigid Bodies

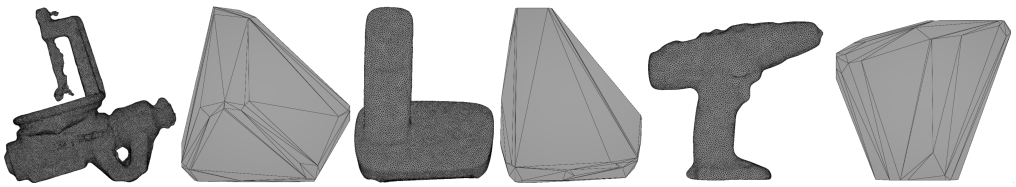


Figure 4.1: The meshes of three LineMOD [14] 3D models before and after cooking

The geometric representation of 3D objects is called mesh, which usually consist of many triangles, each built from three vertices. When simulating physics of objects with fixed geometry, such as the bench vise, phone and drill in figure 4.1, the instances are referred to as rigid bodies.

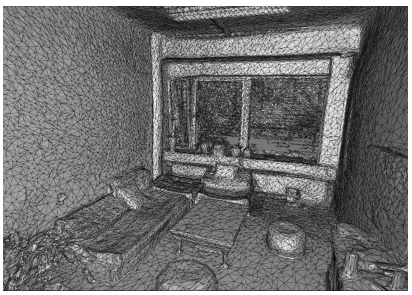


Figure 4.2: Triangle mesh of a room from 3RScan [18]

Simulating rigid body physics means calculating the acting forces for every object in every simulation step. This of course includes determining whether objects collide and correctly solving those collisions. This process, especially if triangle meshes of arbitrary size such as the one seen in 4.2 are used, is very computationally expensive. Many intersection tests are necessary to calculate not only if a collision happened, but also where objects are intersecting. Because of this computational cost, PhysX does not allow triangle meshes to be simulated as dynamic rigid bodies (see [23], rigid body collision). An alternative are convex meshes, which are polyhedrons where every line between two vertices is guaranteed to be within the shape (see [23], geometry).

Our application automatically converts objects used for simulation to convex meshes. This conversion is called *cooking* and provided by the PhysX framework. Cooking drastically reduces the amount of vertices and triangles, which allows usage of very efficient collision detection algorithms. Examples for cooked meshes and their original high-quality counterparts can be seen in figure 4.1. The scene meshes of the 3Rscan data set [18] are cooked as well. This ensures clean and optimized triangle geometry, which is used as a static collider during simulation.

4.2 Continuous & Hardware Accelerated Collision Detection

Physics simulations rely on calculating the passage of time in steps. This can lead to an effect called *tunneling*, where rigid bodies pass through other objects at high speeds (see [23], advanced collision detection). To avoid this, our application makes use of Continuous Collision Detection (CCD). CCD is a system that can drop some amount of time each simulation step and prevents tunneling. This can lead to slow-down effects in real time simulations but is no problem for us since we are only interested in the final poses. Considering that the scene meshes we use are paper-thin, which promotes tunneling, using CCD is the logical conclusion.

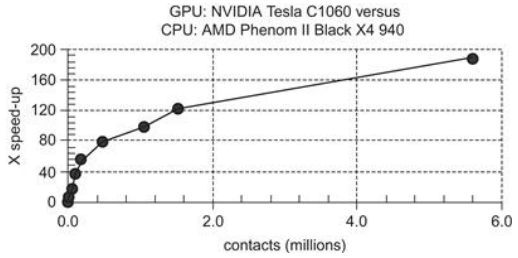


Figure 4.3: GPU rigid body simulation performance compared to CPU [30]

Another feature of Nvidia PhysX is graphics processor (GPU) accelerated simulation. This allows for the collision detection and dynamic rigid body computations to be outsourced to the GPU (see [23], GPU rigid bodies).

According to [30], most required computations can be easily done in parallel and are therefore ideal for the massively parallel architecture of GPUs. The evaluation of their GPU rigid body simulation shows performance gains of up 180 times compared to simulating on a CPU. Although speed-ups are not as drastic in our application, GPU acceleration is still used whenever possible. If no suitable GPU is available, CPU simulation is used as a backup instead.

5 Rendering

Synthetic training data will often rely on rendering at some point, be it to generate the images overall [15, 22] or to augment them [34]. Rendering in computer graphics is the process of generating 2D images primarily from virtual cameras, light sources and 3D objects [2].

To explain the reasoning for what problems our approach may solve, a closer look at the physics of light is required. After providing an overview over the many problems rendering needs to solve and how they are tackled in section 5.1, details about our choice of rendering framework are laid out in 5.2.

5.1 Physically Based Rendering

There are many algorithms designed to make rendering both accurate and fast. The method known as *rasterization* is very popular in real-time applications, as GPUs are used to accelerate the computations. Some of the most frequently involved calculations are even embedded as fixed functions in dedicated hardware. This approach can produce realistic images at interactive speeds but does not necessarily mirror the real world accurately.

Rasterization takes many shortcuts that are not noticeable to the human eye. This can, however, very well make the difference when it comes to computer vision, as evident in [15, 22]. To imitate the real world, it is necessary to accurately model the interactions of light and matter. This forms the basis for the *rendering equation* and what is known as *Physically Based Rendering* (PBR) [2].

5.1.1 Modeling Matter: BRDFs

$$\mathcal{L}_o(v) = \int_{l \in \Omega} f(l, v) \mathcal{L}_i(l)(n \cdot l) dl \tag{5.1}$$

Materials in the real world come in an almost infinite visual variety, even though they all simply modify incoming radiance, which is light of a certain intensity and color, in some way. Thus, a model that covers all possible appearances of matter must replicate how light is scattered mathematically. Reflection and refraction of light rays are examples for those light transfers.

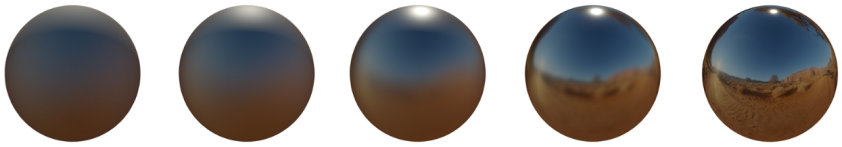


Figure 5.1: Materials with varying BRDFs, from fully diffuse to fully reflective [1]

Bidirectional Reflectance Distribution Functions, or BRDFs, were designed to solve this issue. Taking the reflectance equation 5.1 from [2] as an example, the arbitrary BRDF $f(l, v)$ calculates the outgoing radiance for a view direction v and light direction l. The reflectance equation 5.1 integrates those results for all possible light directions and returns the color and intensity of the reflected light [2].

The simplest BRDF is Lambertian reflectance $f(l, v) = \frac{c_{diff}}{\pi}$, which models a surface with a constant value c_{diff}, called *diffuse color* or *albedo* [2]. Other BRDFs exist to imitate Fresnel reflectance, micro geometry and other physical material properties. Figure 5.1 shows five different BRDFs ranging from fully diffuse to fully reflective.

5.1.2 Modeling Light: Illumination

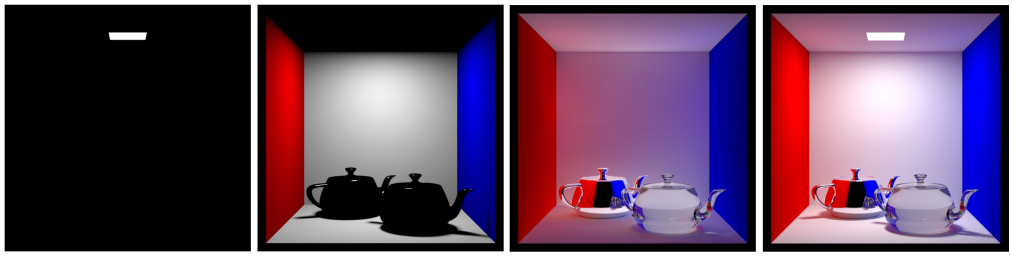


Figure 5.2: Light from the light source, direct, indirect and combined illumination [7]

Another major component in physically based rendering is light transportation, or illumination. According to [7], light is typically divided into direct and indirect illumination. Direct, or local illumination refers to light coming directly from a light source that was reflected at most one time before hitting the camera.

Indirect, or global illumination is more nuanced, as it can be divided further. Incoming light from a rough or matte surface is called indirect diffuse illumination. Indirect diffuse light can also be specularly reflected or refracted further. Incoming radiance from a diffuse surface after a reflection or refraction is called *caustics* [7]. Figure 5.2 shows a scene when only considering local, global and combined illumination or light from the light source directly.

[1] Source: https://google.github.io/filament/Filament.md.html

5.1.3 The Rendering Equation

In order for a renderer to be able to compute radiance arriving at the camera, a function called the *rendering equation* is used. It combines both the physically based material and illumination models by integrating all paths incoming light can take. The reflectance equation 5.1 is a simplified version of the full equation 5.2 [2].

$$\mathcal{L}_o(p,v) = \mathcal{L}_e(p,v) + \int_{l\in\Omega} f(l,v)\mathcal{L}_o(r(p,l),-l)(n\cdot l)^+ dl \tag{5.2}$$

Simply put, 5.2 is the sum of emitted light $\mathcal{L}_e$ and the integrated incoming light, modified by a BRDF, of all possible directions in a hemisphere above a shading point p. This means, that to correctly shade just one point, an infinite number of directions need to be considered. To make matters worse, the rendering equation is a recursive function. Each incoming light direction is required to be radiance as calculated by the rendering equation, unless it is coming from a light source [2].

Considering this, any rendering software can only ever estimate, but never compute, the result of 5.2. Photo realistic rendering means minimizing the inevitable integration errors as best as possible, while using materials and illumination based on physical models and properties.

5.1.4 Ray Tracing

Rasterization, while modeling local illumination very well, struggles when it comes to global illumination, which is why most rendering software aiming to be photo-realistic uses ray tracing instead. This rendering technique simulates light by tracing quantities of light through a scene. At least one light ray is cast through each pixel on the image plane of a camera. Whenever a surface is hit, a ray between the hit point and each light source determines whether the point is in shadow or not. Depending on the type of ray tracing, this first hit point may be the final color [17].

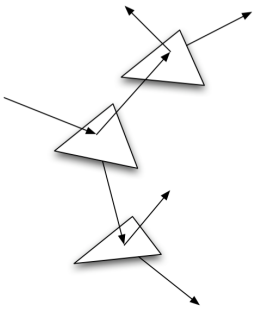


Figure 5.3: Recursive ray tracing [17]

Most ray tracers rely on recursion to achieve a good approximation of the rendering equation. Each hit generates additional rays, depending on the properties of the surface that was hit. This recursive method, also known as *classical ray tracing* or *Whitted-style ray tracing*, is rarely used in modern ray tracing solutions, where *path tracing* is used instead [17]. Figure 5.3 shows an illustration of how recursive ray tracing works.

5.1.5 Path Tracing

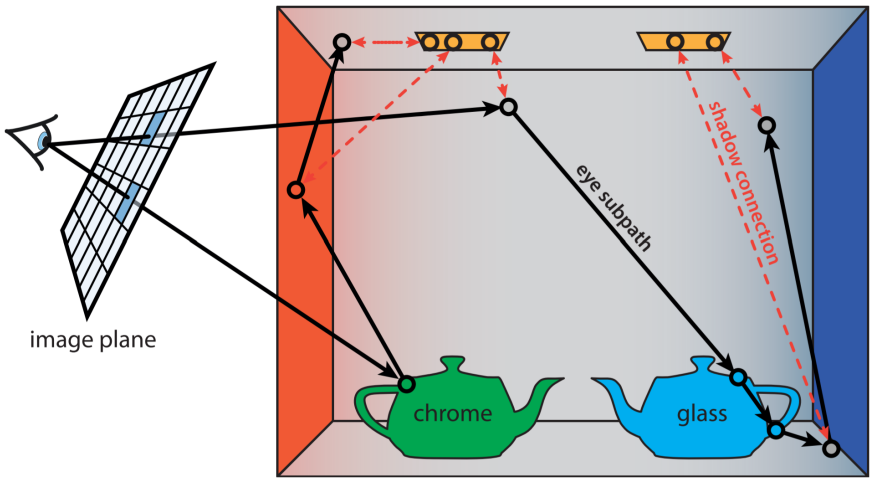


Figure 5.4: Two paths that a ray may take from the eye to the light source [7]

Path tracing is a good trade-off between performance and accuracy and used by most modern rendering softwares such as Autodesks Arnold, Pixars RenderMan and Blenders Cycles. Fundamentally, path tracing is also a recursive ray tracing method, the difference, however, lies in the amount of spawned rays.

The first intersection calculates direct illumination, then a new ray is spawned according to the material properties. The ray type, which can be both a diffuse or specular reflection and refraction, is stochastically determined and traced further. This continues until a light is reached or the path is terminated manually [7]. An example for two paths being traced through a scene is shown in figure 5.4.

The name *path tracing* stems from tracing light paths instead of single rays or branching trees. Each path corresponds to one sample of the rendering equation integral. Given enough samples per pixel, path tracing can thus approximate the integral arbitrarily precise [2].

For rendering photo realistic synthetic image, path tracing is the best available solution and thus what we chose for our approach. One problem that such a generator faces is the lack of scene context. This is apparent when looking at the rendering equation. Indirect illumination relies on the scene context being available, as it needs information about objects other than the one being rendered. Our proposed solution to this problem is going to be detailed in chapter 6.

5.2 Rendering Engines

Figure 5.5: An example scene, rendered using Appleseed [2]

Like many synthetic image generators, our approach uses a render engine for image generation. Rendering engines are frameworks that are capable of rendering an image for a provided scene description, which includes cameras, lights, objects and other parameters. There is a wide range of options available. For our approach we chose Blender in combination with the Appleseed renderer.

5.2.1 Blender

Blender is an open source 3D toolkit, which is, among other things, capable of modeling, animating and most importantly, rendering. It is mainly a dedicated application but can also be used headless in a console or imported into Python [10].

For our application, we import Blender in Python and use a custom Application Programming Interface (API) to provide scene descriptions and control rendering from the core program, which is written in C++. The API reads a scene description file, called Renderfile, parses it into a blender scene and then processes it, outputting the finished renders to disk. This can done in multiple processes at once, thus using all available computing resources and maximizing rendering speed.

[2]Source: https://appleseedhq.net/img/renders/jc-interior.png

5.2.2 Appleseed

While we use Blender as a rendering framework, the actual render engine is Appleseed [24]. This open source renderer features support for path tracing, many light and camera models, a selection of PBR BRDFs and supports the Open Shading Language (OSL). It also has several built-in shading overrides, for example, to render ambient occlusion or object positions. Appleseed also implements progressive rendering and denoising, which can drastically reduce noise in the final image as well as render duration. As an example for a scene rendered with Appleseed, see figure 5.5.

The choice of Appleseed over Cycles, which is a built-in path tracer that comes with Blender, is due to its performance. Appleseed, according to the developers, produces high quality images with fewer samples, while also featuring adaptive sampling to reduce render times further[3].

5.2.3 Open Shading Language

OSL is a programming language to describe materials, lights and more for rendering applications. OSL shaders are programs close to the C language that are invoked during rendering each time a ray hits a surface [16]. It is used in our application to describe custom non-photo realistic (NPR) BRDFs and needed for certain rendering steps of the generation pipeline.

[3]Source: https://forum.appleseedhq.net/t/appleseed-vs-cycles/809

6 Training Data Generation

With the required background knowledge in mind, an explanation of our approach to generating training data now follows. Before a detailed explanation of the generator pipeline in 6.2, the scene scans we used for this thesis are described in section 6.1.

6.1 Scene Data Set

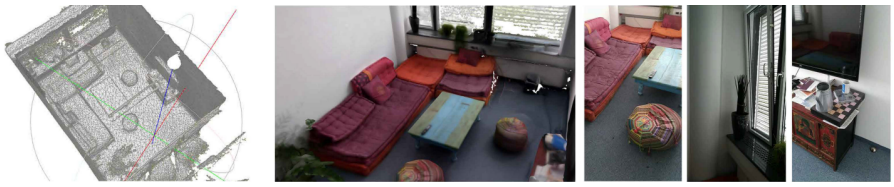


Figure 6.1: The wireframe & textured 3D mesh and some of the RGB images used for reconstruction [18]

For our generator to work, real environments reconstructed from RGB-D images are required. Each scene needs to contain at least the reconstructed, textured 3D mesh, information about the camera and the data set used for reconstruction. The data set has to consist of the RGB images and corresponding camera poses.

The camera intrinsics, which are the principle point and focal length [1], and the image resolution are used to recreate the camera virtually. To match the viewpoint of each captured RGB image, the provided 6 DoF poses, which are relative to the scene origin, are adopted.

The 3RScan data set by Johanna Wald *et al.* [18], which itself is based on the ScanNet data set [8], fulfills those requirements. It consists of 478 unique indoor scenes over a total of 1482 scans, captured in 13+ countries, thus covering a large variety of scenarios. These scans also feature annotations such as semantic segmentation. The application is optimized for this data set, though other correctly formatted scans work as well. Figure 6.1 shows one scene from the 3RScan data set.

[1] Source: https://developer.apple.com/documentation/arkit/arcamera/2875730-intrinsics

6.2 Generator Pipeline

Our training data generator pipeline is split into several steps, the first of which is selecting a scene consisting of RGB images, camera poses and a textured 3D mesh. Next, images that are likely to not be blurred are chosen from the RGB set according to the filter algorithm defined in 6.2.1. Then, synthetic objects are instantiated in the scene and physically simulated for a period of time as described in 6.2.2.

In the first rendering step, we generate depth maps for each selected camera pose. This step, as explained in 6.2.3, is done separately for the scene mesh and virtual objects. After another filtering step, annotations for images with enough visible objects are generated as detailed in 6.2.4.

Finally, ambient occlusion and PBR images are rendered for the objects only. The final image is then created by combining ambient occlusion and PBR renders with the corresponding real image from the scene data set. Section 6.2.5 outlines those final pipeline steps.

6.2.1 Image Selection

The image selection algorithm is split into two steps and is run on every image of a set. First, each image is converted to grayscale and Canny edge detection [2] is performed.

For the lower and upper thresholds t_l and t_u, we use $t_l = 150$ and $t_u = 250$ respectively. The mean edge value μ_e is then calculated and stored. If $\mu_e \geq 0.5$ holds true, the image is marked as a candidate for the next step.

This step removes most heavily blurred images, although some exceptions, like excessive distortions in one direction, can not be caught.

After this preselection process is completed, the remaining images are filtered further. The Laplacian[3] $L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$ is calculated for each pixel $I(x,y)$ and the standard deviation σ_L of the Laplacian is calculated and stored.

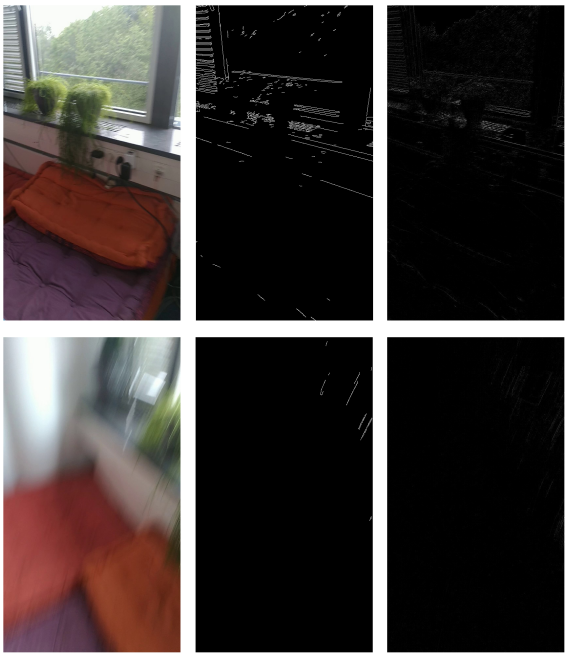


Figure 6.2: A clear (top) and blurred (bottom) image, each with detected edges and Laplacian

[2] See https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html
[3] Source: https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm

Finally, equation 6.1 is used in the second step to discriminate between blurry and clear candidates from the list of preselected images. For a candidate c, μ_c and σ_c are the mean edge value and standard deviation of the Laplacian. The image set size is n and the preselection list contains m candidates.

$$\mu_c \geq \sum_{x=1}^{n} \mu_e(x) \cap \sigma_c \geq \sum_{x=1}^{m} \sigma_L(x) \tag{6.1}$$

This algorithm is based on the assumptions that blurry images do not contain as many edges overall compared to clear ones and that overall the amount of edges varies from scene to scene.

An example of a blurry and clear image as well as their corresponding detected edges and Laplacians can be seen in figure 6.2.

6.2.2 Simulation

If any clear images were detected, the next step is to physically simulate the objects in the scene. The scene mesh is instantiated as a static, triangle-based collider in the physics engine. Then, the dimensions of the bounding box containing it are stored.

The objects are randomly selected from a predefined list; we used the bench vise, drill and phone from the LineMOD [14] 3D scans. The selected objects are instantiated inside a specified spawn box relative to the scene bounds. We opted for an area spanning most of the bounds horizontally but slightly above center vertically. This increases the likelihood that most object spawn in the air and within the scene mesh.

Each object also has a small chance to have a random velocity and torque applied before the simulation starts. Without those random forces, objects tend to land in the same spots, thus limiting the variety of the final poses.

The simulation runs in 0.02s steps for a simulated time span of 40s, ensuring objects have stopped moving and are in a resting position. Those final poses are then used for rendering.

6.2.3 Depth Maps & Occlusion Masks

Figure 6.3: A blended depth map, an object depth map and the resulting occlusion mask

The first rendering step consists of creating the scene and object depth maps, which contain the metric distance to the respective pixel. The depth maps are blended according to 6.2 and a binary occlusion mask is created by evaluating 6.3. Figure 6.3 shows an example for both an object and blended depth map as well as the resulting mask.

$$Depth = \min_{distance}(D_{objects}, D_{scene}) \tag{6.2}$$

$$Mask = D_{objects} > D_{scene} \tag{6.3}$$

The mean value of the occlusion mask is also calculated and used to determine whether an image should be processed further. More than $\approx 1\%$ of the mask has to contain visible objects, otherwise the image is discarded. This threshold ensures no resources are wasted on empty images, while also being conservative enough to not abort generation too often.

While the blended depth map is no longer useful and thus stored on disk, the occlusion mask is vital for the remaining steps of the generation process. It is later used to accurately combine the real RGB images and PBR renders of the synthetic objects.

It should also be noted, that the scene depth maps are only rendered once for each image and, if available, read from disk instead. This optimization saves some performance, as the scene depth maps never change.

6.2.4 Annotations

After computing the occlusion mask and determining whether to proceed with the iteration, a labeled image is created. This step renders the objects with their respective instance IDs as the corresponding shade of gray. Because the white background takes up one of the 256 possible shades an 8 bit channel allows, there can be at most 255 object instances total.

The labeled image is then used to generate the annotation file, which is available for each data point separately. For each visible object instance in an image, it contains the corresponding 2D bounding box, a classifier and name, the pose including scale and the intrinsics of the camera used for rendering. An object is considered visible, if less than 70% of its original area is lost due to occlusion.

An example for a labeled image used to generate annotations can be seen in figure 6.4.

Figure 6.4: Example for a labeled image

6.2.5 Synthetic Object Rendering

The last steps in the pipeline consist of simulating ambient occlusion (AO) and rendering the objects physically based, including the scene indirectly. All renders and the real image are then combined into a final, augmented RGB image.

Ambient Occlusion Pass

Figure 6.5: Ambient occlusion of objects and scene, objects rendered without scene influence, objects rendered with the scene included for indirect illumination (left to right)

According to [17], AO can be thought of as the illumination of an overcast day, with light coming from a large, distant light source, such as the sun.

The scene mesh is included in the AO pass, thus adding soft, global illumination shadows to the synthetic objects. This effectively narrows the gap between shading in the virtual and real scene. Corners and creases, which receive little to no ambient light due to occlusion, appear darker in the final image. Although AO is not entirely physically accurate, it does estimate indirect illumination and shading convincingly[4].

The leftmost image in figure 6.5 shows an example of AO for a scene including various objects. Darker colors indicate, that less ambient illumination is received and thus higher occlusion.

[4]Source: https://docs.blender.org/manual/en/dev/render/cycles/world_settings.html

Physically Based Rendering Pass

In the last pass, the objects, including the scene, are rendered with textured PBR materials. The scene mesh, however, is not rendered directly, as it is only used to calculate indirect diffuse and specular light during path tracing. A path is only fully traced if an object is hit in the beginning. All paths that hit the scene mesh before any object are immediately discarded.

The reason for including the scene mesh indirectly is to approximate both illumination and shading in the real scene. This enhances rendering realism by providing plausible global illumination and even convincing specular reflections. Thus, synthetic objects seem soundly integrated into the scene, as their appearance is influenced by it.

The clear difference between rendering with and without including the scene indirectly can be seen in figure 6.5. All colors and shading on the objects stems from global illumination provided by the scene alone.

For both the object and scene materials, we use Appleseeds *SbsPbrMaterial*[5] shader. This shader is based on the PBR workflow that Allegorithmics Substance Painter[6] offers. The material can easily be customized and fine-tuned to appear more realistic, as it provides sockets for parameters like roughness & metallic texture maps and refraction for transparent objects. For this thesis, however, we only set the diffuse albedo texture and constant metallicness for each unique material.

[5]See https://appleseed.readthedocs.io/projects/appleseed-maya/en/master/shaders/material/as_sbs_pbrmaterial.html

[6]See https://www.substance3d.com/products/substance-painter

Final Image Composition

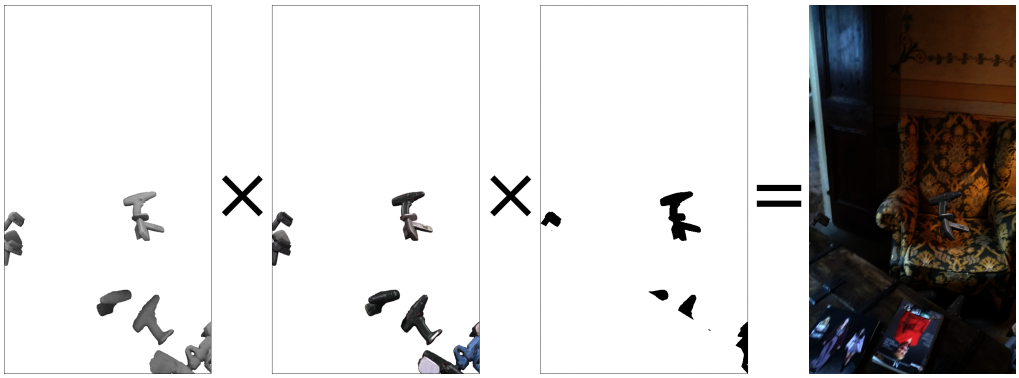


Figure 6.6: The AO render, PBR image, occlusion mask and combined & blended with the corresponding real image (left to right)

After gathering all required renders, the final image can be composed. The blending algorithm 6.4 is used to determine the final color for every pixel $P(x, y)$. An example blending process is illustrated in figure 6.6.

$$\forall P : c_{blend}(x, y) = \begin{cases} c_{pbr}(x, y) \cdot c_{ao}(x, y), & \text{if } Mask(x, y) > 0 \\ c_{real}(x, y), & \text{otherwise} \end{cases} \tag{6.4}$$

Some additional results with varying scenarios can be seen in figure 6.7. It should be noted that the objects do not cast any shadows on the scene. While this is supported and can be enabled, it does not look realistic, as there is no light source estimation integrated. Instead, the lights are always arranged in a rectangle pattern above the scene.

Figure 6.7: Several generated images, based on various scenes

7 Evaluation

With the functionality of the proposed generation pipeline explained, the resulting images will now be evaluated. Both a qualitative and quantitative evaluation is performed for a data set we generated, consisting of 10k images.

First, the results are directly compared to results from related, state of the art synthetic generators. Finally, we train a Mask R-CNN [13] object detection and segmentation network on the data set generated with our proposed approach. The network is evaluated on the LineMOD [14] test sequence and compared to a baseline evaluation from [28].

7.1 Qualitative Evaluation

For the qualitative evaluation, outputs from our approach are compared to both a network driven and a pure rendering based generator. For the network driven method, our images are compared to outputs at several stages of training of *DeceptionNet* [34]. *Photorealistic Image Synthesis for Object Instance Detection* [15], being closely related to our proposed generator, is used for the comparison to pure rendering.

When comparing our results to DeceptionNet [34] as seen in figure 7.1, the main difference to our approach is the random nature of domain randomization. While the goal of those techniques is to provide enough variations to a network to make it resistant to changes, we achieve similar variation through the amount of available scenes. The advantage of our method, however, lies in the availability of scene context, which may provide additional information and can promote generalization as well according to [22].

While our generator currently has no shadow rendering, we still achieve realistic results due to path tracing. The objects in the scene are positioned correctly because of the rigid-body physics simulation. Even though the results from Hodan *et al.* [15] still look a bit more realistic due to, among other things, shadows and better light setup, our results are still of comparable quality. The advantage of our generator is that not only are render times significantly better, but our generator has a larger visual variety overall. A selection of outputs of both generators can be seen in figure 7.2.

7.2 Quantitative Evaluation

For the quantitative evaluation, we train a Mask R-CNN [13] object detection and segmentation network on the 10k synthetic data set we generated. The network is trained in 60 epochs of 100 steps each, starting with the pre-trained Coco weights from [1]. The F1 and IoU scores of resulting network are then calculated for the LineMOD [14] sequence for each object class.

The F1 score combines precision, which describes how many positive predictions were correct, with recall, which is a metric for how many positive predictions were missed. The F1 metric, therefore, takes false positives and negatives into account[1]. Thus, the F1 score determines the accuracy of object class predictions.

Intersection over union (IoU), on the other hand, is used to calculate how accurate the predicted segmentation masks are. Put simply, the score $IoU = \frac{A_O}{A_U}$ is the result of the area of overlap A_O in relation to the bounding area that contains both ground truth and prediction A_U[2]. It therefore serves as a measure for segmentation prediction accuracy.

Object	F1 TO	IoU TO	F1 NE	IoU NE
Benchvise	0.29	0.41	0.50	0.60
Phone	0.15	0.22	0.56	0.64
Drill	0.17	0.21	0.63	0.55
Mean	0.20	0.28	0.56	0.60

Object	F1 TO	IoU TO
Benchvise	0.92	0.92
Phone	0.85	0.85
Drill	0.92	0.92
Mean	0.90	0.90

Table 7.1: F1 and IoU scores for our data set (left) compared to results from [28] (right)

The resulting scores are split into total scores (TO) and scores that only count results where an object was detected in the first place (NE). While the scores of the network trained on our data set are not as good, they are considerably better when there are actually objects detected. A possible explanation for these drastic differences may be, that more training steps would have been necessary. Our network was only trained for 6k steps total, less than the size of the data set. A longer and more thorough training could therefore lead to comparable or better results than the NE scores. Our results are on the left of table 7.1, the results from [28] are on the right.

To evaluate the results qualitatively, figure 7.3 shows six images with predicted labels and corresponding masks from the quantitative evaluation. The results are, as expected, not always accurate, as there are often too many objects being detected. While the network sometimes fails to detect an object in the first place, it is fairly accurate when the correct object is detected. This supports the assumption, that more training may have yielded better results.

[1] Source: https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/

[2] Source: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

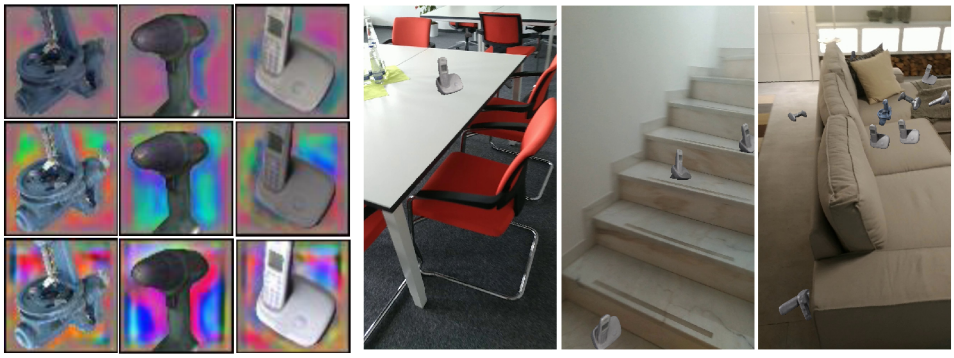


Figure 7.1: Outputs from DeceptionNet [34] (left) compared to three outputs from our generator (right)

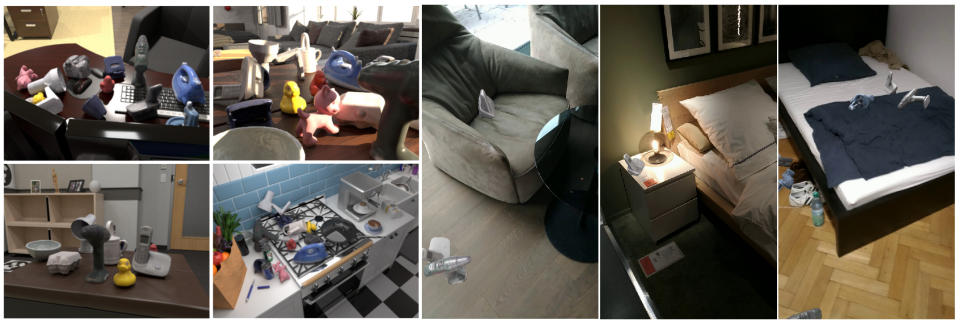


Figure 7.2: Ouputs from Hodan *et al.* [15] (left) compared to three outputs from our generator (right)

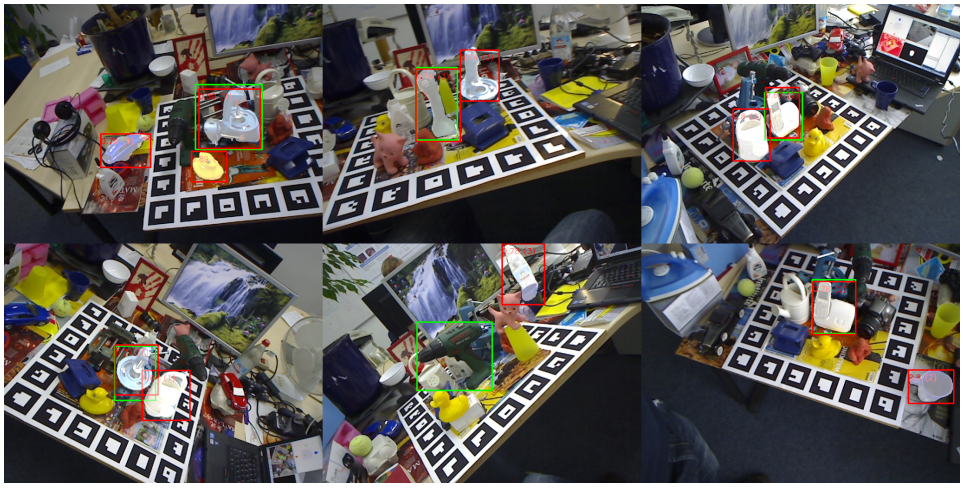


Figure 7.3: The trained Mask R-CNN [13] network evaluated on LineMOD [14], including predicted segmentation masks and labels

8 Discussion

Generating synthetic training data is a challenging problem and a field of active research. There are many different approaches that have been tried in an attempt to close the domain gap between real and synthetic data, yet there is still no gold standard.

Network driven generators relying on domain randomization [34, 26] and domain adaption [5, 21, 32] can create large amounts of data from existing data sets but the outputs are not photo-realistic images. While networks trained using those images seem to perform well overall, generalization to real world scenarios it is not guaranteed.

Synthetic data sets have also been generated purely through rendering [15, 22] and while the results are photo-realistic and comparable to real data in most aspects, rendering is computationally very expensive. Thus, generating enough data is constrained heavily by both the availability of time and resources.

We proposed a new approach in this thesis, which combines low computational costs of network driven generators with the photo-realism of pure rendering. Through the means of physically based rendering combined with physics simulation, we achieve high quality renders of objects in plausible poses.

8.1 Advantages

Our proposed method shows a lot of potential in generation speed and realism.

Having access to varied scenes from 3RScan [18] allows for sufficiently varied training data, therefore increasing the chances of a network generalizing well to unseen data and the real world. It has been shown multiple times before [26, 34], that more varied training data encourages networks to be more resistant to performance degradation in unexpected scenarios.

Since we use reconstructed scene meshes, we can accurately simulate the rigid-body physics of the virtual objects. This provides us with convincing poses, which are employed during rendering. Using the scene meshes during path tracing circumvents wrong or missing global illumination, reflections and overall shading. These features at least partly solve the lack of scene context in synthetic images, as described in 3.1.2.

Because we split the generator into several steps, we can abort generation early if an image is not promising, which in return improves generation times. Rendering only the objects rather than the entire scene improves render times drastically as well, as the i7-6700k CPU we used for rendering was able to generate roughly 5k images per day. A pure rendering approach like the one used by Hodan *et al.* [15] does not only take longer on average, but also requires CPU clusters to generate enough data.

8.2 Challenges

Figure 8.1: Lighting not matching scene illumination, "clipping" due to an inaccurate scene mesh, low quality result due to undetected blur (left to right)

Even though our approach yields promising results, there are still some hurdles to overcome before it can be a replacement for real training data. With photo-realism being the goal, there is still a lot of work to be done in the scene lighting setup. We arranged four lights in the upper corners of the scene boundaries, which does provide sufficient light, but is not accurate in most cases. Because our lighting setup does not match the original scene, rendered shadows would be incorrect as well. Without shadows, however, there is a loss of both realism and scene context.

Another problem is the quality of the scene meshes. While most meshes are not necessarily extremely high quality 3D scans, they do suffice for our generator. One very common problem, however, is virtual objects *"clipping"* into chair legs, lamp stands and other thin, small furniture. Those objects can not be reconstructed well enough and are thus either wrong or not present in the 3D scan at all.

The last area that needs improvement is the blur detection algorithm. Although it is fast and filters out most blurred images, it does also filter out images that are clear. This leads to some scenes only having single digit accepted images and therefore limiting the variety in the data set. Another issue is that even though most selected images are clear, there are also some forms of blur that can not be filtered out, meaning not all final images are guaranteed to be of the same quality or usable at all.

An example for each of the described problems can be seen in figure 8.1. It should be noted, however, that these are outlier cases, most problematic images have these issues to a smaller degree. The most pronounced problem by far is object *"clipping"* and only a real issue in a subset of the available scenes.

9 Future Work

The proposed method has a lot of potential and already yields mostly promising results, yet there is still a lot of room for extensions and improvements.

The first and most obvious improvement is implementing some form of light estimation, as described in 3.2. The most beneficial solution would be light source estimation rather than global illumination approximations, as this is already partly implemented. While adding light estimation was not possible in the scope of this thesis, it would allow the generator to output not only shadows but also improve overall photo-realism further.

Pre-processing of the scene meshes by removing unnecessary geometry and fixing inaccuracies such as holes is another potential area of improvement. Since the original depth maps are available, it may be possible to reduce *"clipping"* by incorporating them into the masking process. Those real depth maps could be combined with the rendered scene depth to improve the overall results.

Finally, using a different render engine than Blender would allow for even faster render times and less performance overhead through the nature of Python. Switching to a different rendering system would also possibly support GPU rendering, which can significantly boost rendering performance as well. Although there are not many open source platforms to choose from, LuxCoreRender[1] might be a valid alternative, provided OSL is no longer a requirement and can be removed.

[1] See https://luxcorerender.org/

List of Figures

List of Tables

Bibliography

[1] W. Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. (accessed 25.09.2020). 2017. URL: https://github.com/matterport/Mask_RCNN.

[2] T. Akenine-Mller, E. Haines, and N. Hoffman. *Real-Time Rendering, Fourth Edition*. 4th. USA: A. K. Peters, Ltd., 2018. ISBN: 0134997832.

[3] A. Antoniou, A. Storkey, and H. Edwards. "Data augmentation generative adversarial networks." In: *arXiv preprint arXiv:1711.04340* (2017).

[4] L. Arnold, S. Rebecchi, S. Chevallier, and H. Paugam-Moisy. "An Introduction to Deep Learning." In: vol. 1. Jan. 2011, pp. 477–488.

[5] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan. "Unsupervised pixel-level domain adaptation with generative adversarial networks." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 3722–3731.

[6] R. Y. Choi, A. S. Coyner, J. Kalpathy-Cramer, M. F. Chiang, and J. P. Campbell. "Introduction to Machine Learning, Neural Networks, and Deep Learning." In: *Translational Vision Science & Technology* 9.2 (Feb. 2020), pp. 14–14. ISSN: 2164-2591. DOI: 10.1167/tvst.9.2.14.

[7] P. H. Christensen and W. Jarosz. "The path to path-traced movies." In: *Foundations and Trends® in Computer Graphics and Vision* 10.2 (2016), pp. 103–175.

[8] A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, and M. Nießner. "Scannet: Richly-annotated 3d reconstructions of indoor scenes." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5828–5839.

[9] P. Debevec. "Image-based lighting." In: *ACM SIGGRAPH 2006 Courses*. 2006, 4–es.

[10] B. Foundation. *Blender Website*. (accessed 21.09.2020). 2020. URL: https://www.blender.org/.

[11] M.-A. Gardner, Y. Hold-Geoffroy, K. Sunkavalli, C. Gagné, and J.-F. Lalonde. "Deep parametric indoor lighting estimation." In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 7175–7183.

[12] R. M. Haralick, H. Joo, C.-N. Lee, X. Zhuang, V. G. Vaidya, and M. B. Kim. "Pose estimation from corresponding point data." In: *IEEE Transactions on Systems, Man, and Cybernetics* 19.6 (1989), pp. 1426–1446.

[13] K. He, G. Gkioxari, P. Dollár, and R. Girshick. "Mask R-CNN." In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[14] S. Hinterstoisser, V. Lepetit, S. Ilic, S. Holzer, G. Bradski, K. Konolige, and N. Navab. "Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes." In: *Asian conference on computer vision*. Springer. 2012, pp. 548–562.

[15] T. Hodan, V. Vineet, R. Gal, E. Shalev, J. Hanzelka, T. Connell, P. Urbina, S. Sinha, and B. Guenter. "Photorealistic Image Synthesis for Object Instance Detection." In: Sept. 2019, pp. 66–70. DOI: 10.1109/ICIP.2019.8803821.

[16] S. P. Imageworks. *Open Shading Language Website*. (accessed 21.09.2020). 2020. URL: https://github.com/imageworks/OpenShadingLanguage#readme.

[17] H. W. Jensen and P. Christensen. "High quality rendering using ray tracing and photon mapping." In: *ACM SIGGRAPH 2007 courses*. 2007, 1–es.

[18] A. A. Johanna Wald, F. T. Nassir Navab, and M. Niessner. "RIO: 3D Object Instance Re-Localization in Changing Indoor Environments." In: *Proceedings IEEE International Conference on Computer Vision (ICCV)*. 2019.

[19] R. Kaskman, S. Zakharov, I. Shugurov, and S. Ilic. "Homebreweddb: Rgb-d dataset for 6d pose estimation of 3d objects." In: *Proceedings of the IEEE International Conference on Computer Vision Workshops*. 2019.

[20] C. LeGendre, W.-C. Ma, G. Fyffe, J. Flynn, L. Charbonnel, J. Busch, and P. Debevec. "Deeplight: Learning illumination for unconstrained mobile mixed reality." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 5918–5928.

[21] M.-Y. Liu and O. Tuzel. "Coupled generative adversarial networks." In: *Advances in neural information processing systems*. 2016, pp. 469–477.

[22] Y. Movshovitz-Attias, T. Kanade, and Y. Sheikh. "How useful is photo-realistic rendering for visual learning?" In: *European Conference on Computer Vision*. Springer. 2016, pp. 202–217.

[23] Nvidia. *Nvidia PhysX Documentation*. (accessed 18.09.2020). 2018. URL: https://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXGuide/Manual/Index.html.

[24] appleseedhq Organization. *Appleseed Website*. (accessed 21.09.2020). 2020. URL: https://appleseedhq.net/features.html.

[25] M. Rad and V. Lepetit. "BB8: A scalable, accurate, robust to partial occlusion method for predicting the 3D poses of challenging objects without using depth." In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 3828–3836.

[26] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. "Domain randomization for transferring deep neural networks from simulation to the real world." In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 23–30.

[27] Y.-H. Tsai, X. Shen, Z. Lin, K. Sunkavalli, X. Lu, and M.-H. Yang. "Deep image harmonization." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 3789–3797.

[28] G. Wang, F. Manhardt, J. Shao, X. Ji, N. Navab, and F. Tombari. "Self6D: Self-Supervised Monocular 6D Object Pose Estimation." In: *arXiv preprint arXiv:2004.06468* (2020).

[29] H. Wang and B. Raj. "On the Origin of Deep Learning." In: *arXiv preprint arXiv:1702.07800* (2017).

[30] W. H. Wen-mei. *GPU Computing Gems Jade Edition*. Elsevier, 2011. Chap. 20.

[31] Y. Xiang, T. Schmidt, V. Narayanan, and D. Fox. "PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes." In: June 2018. DOI: 10.15607/RSS.2018.XIV.019.

[32] D. Yoo, N. Kim, S. Park, A. S. Paek, and I. S. Kweon. "Pixel-level domain transfer." In: *European Conference on Computer Vision*. Springer. 2016, pp. 517–532.

[33] Y. Yu and W. A. Smith. "InverseRenderNet: Learning single image inverse rendering." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 3155–3164.

[34] S. Zakharov, W. Kehl, and S. Ilic. "Deceptionnet: Network-driven domain randomization." In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 532–541.