Stefan Tscheppe

Retargeting Video Tutorials using Assembly Graph Constraints

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Denis Kalkofen
Institute for Computer Graphics and Vision

Dipl.-Ing. Peter Mohr-Ziak BSc
Institute for Computer Graphics and Vision

Graz, Austria, May 2019

Abstract

Tutorials pervade our daily lives. Be it from building a shelf at home or something more complex as repairing mechanical machinery. Current systems for the automatized creation of tutorials focus on one type of media. This thesis outlines a system for the automatized creation of XML based construction manuals, which can be used as a basis for tutorials of any type of media. The inputs to our proposed system are common videos depicting the assembly or disassembly of known objects. We use a CNN in combination with a 6 DoF pose estimation framework to extract information from such a video. By drawing conclusions from that information and by validating it against a disassembly graph of the known object, we build an XML based construction manual for the sequence shown in the video. We present two sample applications which create tutorials for different types of media based on one such an XML based construction manual.

Kurzfassung

Täglich kommen wir in Kontakt mit Gebrauchsanweisungen, beim Aufbau eines Regals zu Hause oder beim Reparieren von Komplexerem wie mechanische Maschinen. Derzeitige Systeme zur automatischen Erstellung von Gebrauchsanweisungen beschränken sich auf ein Medium. Diese Diplomarbeit beschreibt ein System zur automatischen Erstellung von XML basierten Anleitungen, die als Basis für Gebrauchsanweisungen beliebiger Medien verwendet werden können. Dieses System benötigt eine Videoaufnahme der Montage bzw. Demontage eines bekannten Objekts. Wir benutzen ein CNN und ein Framework zur Schätzung von Posen in 6 Freiheitsgraden, um Informationen aus der Videoaufnahme zu gewinnen. Mit den Informationen aus der Videoaufnahme und der anschließenden Validierung mit einem Graphen, der die Demontage des bekannten Objekts beschreibt, werden Rückschlüsse gezogen. Diese ermöglichen uns eine XML basierte Anleitung, die die Sequenz aus der Videoaufnahme widerspiegelt, zu erstellen. Wir präsentieren zwei Anwendungen, für unterschiedliche Medien, die Gebrauchsanweisungen auf XML basierenden Anleitungen erzeugen.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Date	Signature

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

People often come across tutorials, independent on their field of work. Tutorials often try to simplify a particular subject matter such that non-experts can complete an otherwise difficult task without the need to understand the underlying and often complex details fully. For example, that is the case for mechanical machinery, where recordings are used to demonstrate how mechanical machinery can be repaired in case of damage.

With display devices becoming smaller while also being able to support more computing-intensive applications, new types of tutorials are becoming more popular. Not only methodologies of tutorials have to be rethought, but also new concepts and new ways to interact with a tutorial arise as those display devices also support a wide range of different interactions with them. That often allows creating more concise and easier to understand tutorials.

There has been research on the automatized creation of tutorials based on given media like images or videos by Mohr et al. [22, 23] and Chi et al. [7] or recorded interactions by Wang et al. [45] for different types of tutorials. The focus of this thesis is the automatized creation of an XML based construction manual based on a given video tutorial for known 3D objects. Using one such XML based construction manual, it is then possible to target multiple different types of media like print or augmented reality to display guided instructions. Especially in the case of displaying guided instructions via augmented reality technologies, it is an improvement over plain video recordings, when, e.g., trying to repair mechanical machinery as virtual instructions can be directly overlayed over real objects.

One of the challenges in creating an XML based construction manual is the extraction of meaningful data from a given video. That is why we specifically focus on known 3D objects, where we then can try to extract information like visibility and occlusions and pose. By extracting information from the video and by drawing conclusions about the relations between the known 3D objects, we hope to create a meaningful XML based construction manual.

As mentioned above, tutorials try to hide complex details, and as such we propose a system that automatizes the creation of such a construction manual. That means no in-

trinsic knowledge about the automatization system itself is required and only the necessary input data has to be provided.

1.1 Conceptional Workflow

When we further lay out a plan based on the mentioned concepts, we arrive at the following involved parties *input provider*, *system* and *tutorial creator*. See Figure 1.1 for a conceptional overview of the workflow and the interfaces between these parties. Note that the result of this *system* is an XML based construction manual. Following we describe each party in more detail.

Input Provider. We mentioned that tutorials simplify a specific subject matter and as such often provide step by step instructions. From the *input provider* we require a video and a 3D object. The video should depict, e.g., some mechanical machinery being disassembled by an expert or a person with domain knowledge. The provided 3D object should be a virtual representation of this real-world mechanical machinery. Either that is the original Computer Aided Design (*CAD*) data or an adequate reconstruction. Both this video and the related 3D object are then given to the *system*.

System and XML based construction manuals. The primary part of this thesis is this *system* which creates XML based construction manuals. Given a video and a 3D object from any *input provider* as a basis, this *system* creates such an XML based construction manual.

Tutorial Creator. Finally, based on an XML based construction manual and its related 3D object, any type of tutorial can be created by any *tutorial creator*. The structure depicted by Figure 1.1 illustrates how *tutorial creators* only need to concern themselves with the actual presentational parts of a tutorial. Note that based on one XML based construction manual, any number of different tutorials can be built.

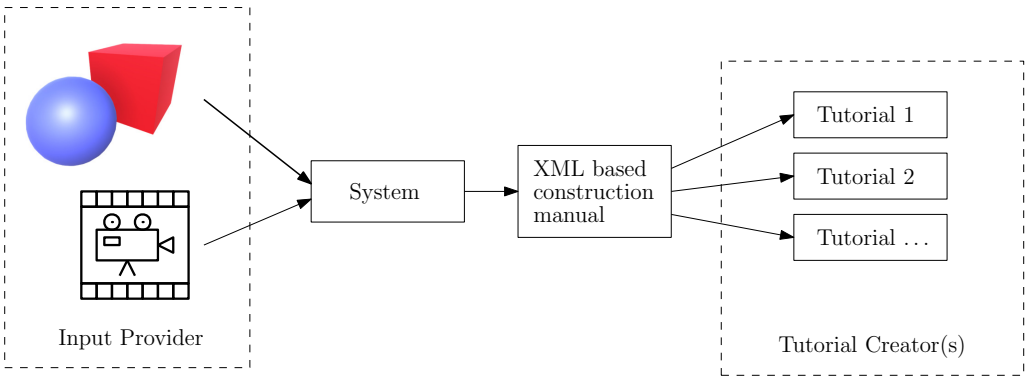


Figure 1.1: An overview of the conceptional workflow and the involved parties.

2 Related Work

Inferring from what was mentioned in the previous chapter, we need algorithms or frameworks which can detect predefined 3D objects in 2D space. Specifically, allowing us to determine if a 3D object is visible in a given image and if possible even get its pose.

Evaluating if an object is visible in a given image can be treated as an image classification problem, and we propose using a Convolutional Neural Network as they have long been established as powerful tools to solve this specific type of problem.

Estimating the pose of a 3D object in 6 Degrees of Freedom (DoF) is a well-defined research problem. While Convolutional Neural Networks are used to solve this type of problem as well, there exist solutions that use image segmentation on RGB images and calculate a 6 DoF pose by, e.g., detecting the 2D contours of known 3D objects.

2.1 Image Classification and Convolutional Neural Networks

We described earlier that Convolutional Neural Network (CNN) have long been established as powerful tools to solve image classification problems. The basic architecture of a Convolutional Neural Network usually consists of multiple layers and input passes through these layers in batches. In the training phase, a dataset containing images, which have annotations for the classes the CNN should recognize, is given to the CNN, and it trains on the dataset until a convergence criterion is met. After validating the training phase, a new image can be given to the CNN. Depending on the setup of the CNN the result will be a prediction if and where one of the trained classes is in this new image. When making predictions over consecutive images, e.g., when the input is a video, note that one can validate the prediction of a frame with its predecessor and successor.

2.1.1 Synthetic Datasets

The different classes Convolutional Neural Networks can detect are only limited by the datasets one can provide for training. While there are many annotated datasets available, often you need to create a dataset on your own. That is when you want to customize your *CNN* for your purposes and detect sometimes unusual classes in images.

On the one hand, you want a comprehensive dataset, where each class is covered from every possible angle and in different scenarios. On the other hand annotating is a very time-consuming task as it has to be done manually. For this reason, there exist tools to make this labeling process as frictionless as possible like, e.g., *LabelMe* [36]. A prominent example is the *Google-Image-Labeler* [11] which is a web-tool where you can label random images to help Google build its datasets. Building, specifically annotating, datasets can be seen as a problem of its own. Apart from specific tools, which try to aid the labeling process, it also is possible to build whole synthetic datasets.

The quality of 3D object renderings is gradually improving year by year. This can be observed when trying to tell apart two images, where one is a photograph and the other a rendering of the same object. Exploiting this fact, one approach to synthetic datasets is to create a virtual environment for each class and create renderings from different angles and in different scenarios. Note that in a virtual environment objects can be annotated algorithmically, which is an improvement over annotating images by hand. The drawback in this scenario is that you need to set up a realistic virtual environment and require 3D representations of the object classes you, later on, want the *CNN* to be able to detect. The more realistic these synthetic images are, the better the quality of the dataset. Keep in mind that in the end your *CNN* will classify real photographs.

Ways to create those renderings of 3D objects are, e.g., the engine Unity3D [42] or the 3D modeler Blender [4]. Examples, where *CNN*s are trained on synthetic datasets, are presented by Jensen and Selvik [15], Rajpura et al. [30] and Peng et al. [26]. Note that it is also possible to create mixed datasets where real images are mixed with synthetic images as shown by Tian et al. [40].

2.1.2 Choosing a Convolutional Neural Network

There exists a wide variety of *CNN* which try to solve the image classification problem. The Pascal Visual Object Classes (*VOC*) Challenge tries to evaluate those *CNN* by providing standardized annotated datasets and standardized evaluation procedures, in the end comparing how well each *CNN* solves the problem of image classification [10]. Similar to the Pascal *VOC* Challenge, there are also other competitions which share the same goal like, e.g., the *ImageNet Large Scale Visual Recognition Challenge* [35]. Aside from those competitions many datasets like, e.g., the *Caltech256* dataset [12] or the *Lotus Hill* dataset [46] are available to the community to provide comparable training data for their *CNN*s.

Real-Time Detectors	Train	mAP	FPS
100Hz DPM	2007	16.0	100
30Hz DPM	2007	26.1	30
Fast YOLO	2007+2012	52.7	155
YOLO	2007+2012	63.4	45
Less Than Real-Time			
Fastest DPM	2007	30.4	15
R-CNN Minus R	2007	53.5	6
Fast R-CNN	2007+2012	70.0	0.5
Faster R-CNN VGG-16	2007+2012	73.2	7
Faster R-CNN ZF	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Table 2.1: A comparison of real-time detectors on the Pascal *VOC* datasets stated by Redmon et al. [31, chapter *Experiments*, pp 6].

The Pascal *VOC* challenge itself was held annually starting from 2005 to 2012 [9]. For every challenge, a new dataset containing publicly available images were annotated and used as ground truth. Every competition was split into the two main categories of classification, where the presence or absence of an object is predicted, and detection, where bounding boxes around objects are predicted. The results of each competing *CNN* are then evaluated in the form of a report.

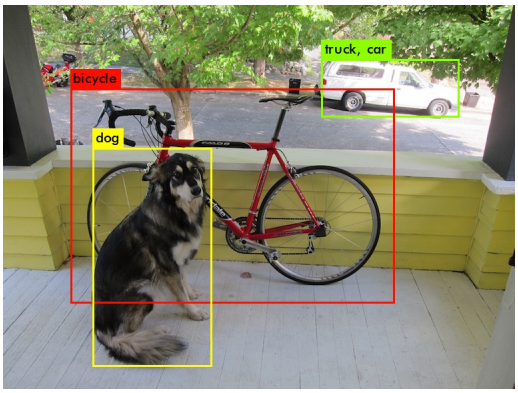
One criterion to keep in mind is the average time of detection on one image. For example for real-time applications that means you want to process images as fast as possible. Table 2.1 shows a comparison of real-time *CNN*s, where Pascal *VOC* datasets were used. Note that the mean Average Precision (*mAP*) can be in the range from 0 to 100 – the higher the *mAP*, the better the image classification capabilities of a *CNN*. Together with the frames per second (*FPS*), those two values are a good indicator of the real-time performance of a *CNN*. Note that state of the art *CNN*s are implemented to run on *GPU*s, specifically Nvidia® graphics cards.

2.1.3 You Only Look Once

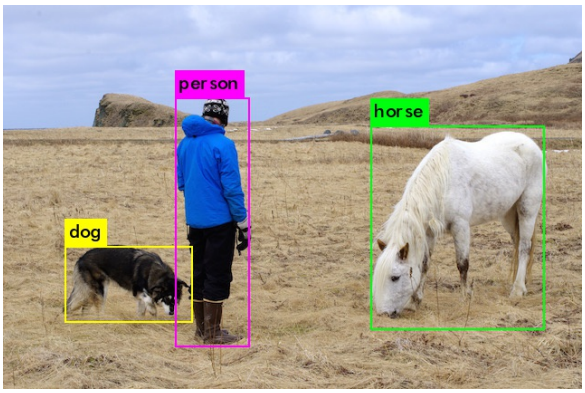
The Convolutional Neural Network You Only Look Once (*YOLO*) by Redmon et al. [31] is a state-of-the-art object detector, which stands out among other image classification *CNN*s due to its overall good prediction and fast detection rates – see Table 2.1 for a comparison.

YOLO takes a new approach to object detection and reasons globally about objects in a given image instead of, e.g., using the sliding window technique like other *CNN*s. One could say that *YOLO* predicts bounding boxes straight from image pixels – see Figure 2.1 for an example. For more examples, one can visit the website of *YOLO* https://pjreddie.com/darknet/yolo/.

Note that while *YOLO* makes fast predictions, it lags behind state of the art *CNN*s in terms of accuracy. It also struggles to localize objects in new or unusual configurations as stated by Redmon et al. [31, chapter *Limitations of YOLO*, pp 4]. Still, it outperforms other real-time object detection systems in terms of the ratio of accuracy and detection time.



(a) YOLO Prediction Example 1



(b) YOLO Prediction Example 2

Figure 2.1: Both pictures present bounding boxes where *YOLO* predicted an object class. For these predictions a pre-trained weight and example images were used. Note that in Figure 2.1a `dog` was predicted with 100%, `bicycle` with 99%, `car` with 25% and `truck` with 92%, while in Figure 2.1b `person` was predicted with 100%, `dog` with 99% and `horse` with 100%. It is noteworthy that `truck` and `car` feature the same predicted bounding box, but with vastly different prediction rates.

2.1.3.1 Different Versions

Since first publishing You Only Look Once (*YOLO*) in 2015 [31], the object detection system has been continuously researched on and currently its third version *YOLO*v3 [33] is available. *YOLO*v3 still has the same basic characteristics as its initial version, but its accuracy has been improved and its detection times got faster.

Based on *YOLO*, there has been research on its applicability in different real-time contexts. For example Shafiee et al. [38], show how *YOLO*v2 [32] can run on an embedded device where computation power is limited. Pedoeem and Huang [25] present how *YOLO* even runs in real-time on non-*GPU* devices. This is especially useful for, e.g., augmented reality applications, where devices are usually small and computation power is limited and strictly non-*GPU*.

2.2 Pose Estimation

While pose estimation is a core research problem, it has gained more attention in recent time due to its applicability in the context of augmented reality, where pose estimation is a basic problem [20]. Note that in this context additional means like sensors can be used to confirm or falsify predictions on the pose of 3D objects.

Looking at the types of input, 2D images or point clouds or a combination of both, and what is estimated, general poses like, e.g., human shapes or poses of specific 3D objects, there exist a variety of approaches to estimating 6 *DoF* for an object. For example, while Ye et al. [47] show how the pose of a human body is estimated, Aldoma et al. [1] estimate the 6 *DoF* pose for known *CAD* models. Note that both use point clouds, e.g., captured by a depth camera like the Microsoft Kinect, as input.

In this chapter, we want to further explore one specific approach which estimates a 6 *DoF* pose of a 3D object over a given video. By having continuous 2D images, one can reason about the pose of an object with prior or posterior knowledge. Following is an outline of the above-mentioned method called 2D-3D pose estimation [6, 8, 29, 37]. It requires that the 3D object is known and an initial pose for the first frame of the video, e.g., a recording of the 3D object, is set. Now for every frame, the 3D object is projected onto the 2D video frame. This projection is then used for image segmentation and the resulting contour for pose estimation – see Figure 2.2. The estimated pose is then used in the next frame and iteratively a 6 *DoF* pose is estimated for every frame of the video.

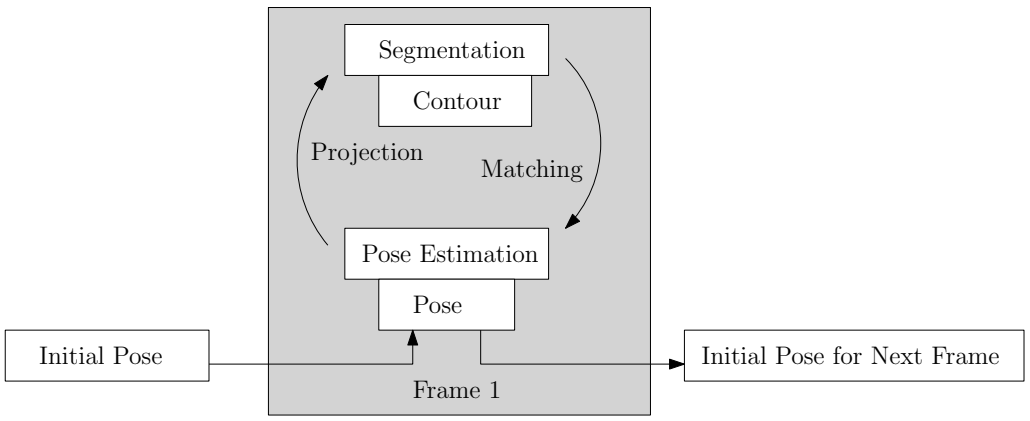


Figure 2.2: Pose estimation by image segmentation adapted from Brox et al. [6, chapter *Introduction*, pp 3]. The workflow of this 2D-3D pose estimation requires an initial pose for the first image in a continuous image stream like, e.g., a video. By projecting this initial pose onto the first frame, segmenting a contour and matching this to a pose of the known 3D object, an initial pose for the next frame is calculated.

2.2.1 PWP3D

One such 2D-3D pose estimation system is PWP3D by Prisacariu and Reid [29][28]. While PWP3D tries to solve the problem of simultaneous image segmentation and pose estimation as other similar 2D-3D pose estimation systems, it achieves this in real-time. Note that for 2D-3D pose tracking over multiple frames, an accurate 3D model of the object to be tracked is required.

Figure 2.3 shows how PWP3D successfully recovers the pose of a 3D object – the initialized pose on the left, the final result on the right. Note that PWP3D can recover from positional misalignments in x, y and z of up to about 40% and rotational misalignments of about 50 degrees on the x- and y-axis and 70 degrees on the z-axis as stated by Prisacariu and Reid [29, chapter *Results*, pp 12].

Figure 2.3: Typical PWP3D run for one frame presented by Prisacariu and Reid [29, chapter *Results*, pp 12] with left – initialization, middle – intermediate iteration, right – final result.

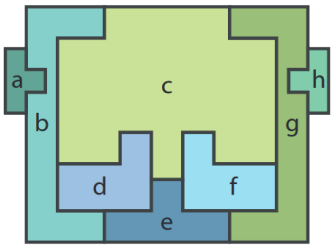
So far we have only discussed the case of monocular pose estimation. Note that PWP3D is capable of processing multiple views as stated by Prisacariu and Reid [29, chapter *Multiple Views*, pp 7]. Multiple views make it easier to discern ambiguous 2D projections as different poses of a 3D object can lead to the same projection. In general, multiple views are beneficial and contribute towards more accurate pose estimation.

2.3 Exploded View Diagrams

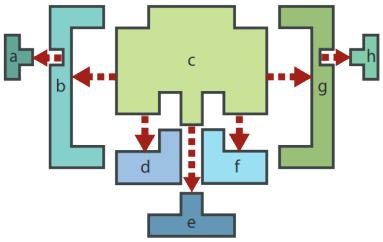
Conceptional drawings of objects or exploded view diagrams have cemented themselves as valuable visualization methods for presentational purposes. Initially exploded view diagrams were drawings on papers and made their transition to 3D with the coming of Computer Aided Design (*CAD*). In recent time exploded view diagrams transitioned even further, namely into the field of augmented reality as shown by Kalkofen et al. [17], which opened new ways of, e.g., interacting with them.

2.3.1 Explosion Graphs

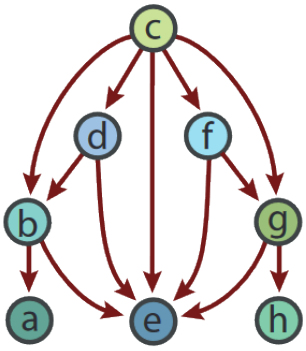
The main challenges when creating and designing virtual exploded view diagrams are the handling of occlusions, interaction with the diagram and the creation based on *CAD* or 3D object data itself. For this reason, the automatized creation of exploded view diagrams is topic of research and, e.g., Tatzgern et al. [39] describe a system which automatically creates an exploded view diagram based on given unmodified *CAD* data. While Tatzgern et al. [39] further define an algorithm to compute a disassembly sequence, Li et al. [18] create an explosion graph based on constraints derived from given 3D object data.



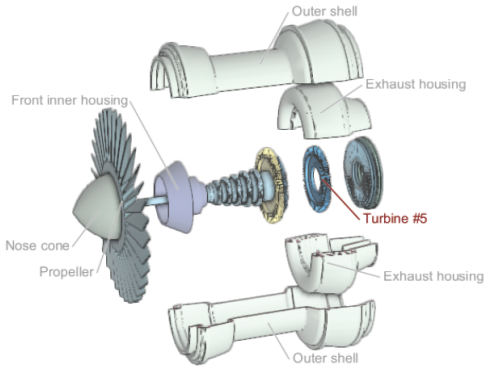
(a) Input Model



(b) Exploded Model



(c) Explosion Graph



(d) Exploded View Diagram of a Turbine [18]

Figure 2.4: Explosion graphs are based on a given 3D object which is used as input model. Figure 2.4a shows such an input model and Figure 2.4b its related exploded model. This exploded model can be more formerly encoded as an explosion graph as presented by Figure 2.4c. One application of explosion graphs is the automated generation of exploded view diagrams. One such exploded view diagram of a turbine presented by Li et al. [18] is shown in Figure 2.4d.

Specifically, Li et al. [18] derive constraints by verifying at each step of a disassembly what parts can be removed. This results in a directed explosion graph, where each node represents one part of a given 3D object. See Figure 2.4a, where the input model results in the explosion graph seen in Figure 2.4c. Note that Figure 2.4b illustrates every possibly disassembly of the input model and the directed edges of an explosion graph encode the dependencies between the different parts of a given 3D object.

As previously mentioned, one practical application of explosion graphs is the automated generation of interactive exploded view diagrams as shown in Figure 2.4d. Every node in the explosion graph also encodes the unconstrained direction of its related 3D object part. Note that all this additional information about the parts of a given 3D object can be used in visualizations and presentations of this object. Therefore, this generic exploded-view-diagram graph-description is well suited as a foundation for further research on applications of exploded view diagrams.

3 System Overview

In Chapter 1 we outlined the focus of this thesis and in Chapter 2 we discussed its core challenges. In the following chapter we combine the frameworks outlined in Section 2.1.3, Section 2.2.1 and Section 2.3.1 to build a system which is capable of the automatized creation of an XML based construction manual based on a given video tutorial for known 3D objects.

Figure 3.1 shows how the different parts of our proposed system build on each other. In a first step You Only Look Once (*YOLO*) trains on a synthetic dataset based on the known 3D objects. After training is finished, the video tutorial is fed to *YOLO* as input. The results of this detection are then used in the initialization step of PWP3D. Again, the video tutorial is fed as input, but this time to PWP3D, resulting in pose estimations for each frame of the video for the known 3D object. Together with the explosion graph of the 3D object, all previously created information is processed by a Unity3D [42] application resulting in an XML based construction manual. Note that the resulting construction manual can then be used as a basis to create different types of tutorials as the manual contains a (dis)assembly sequence of the known 3D object matching the recorded (dis)assembly sequence of this object.

3.1 Terminology

Following are explanations for terms, that will be used in later sections of this chapter.

Assembly. The goal of this XML based construction manual is to be able to build tutorials for a specific known 3D object. When we talk about an assembly, we will be referring to this known 3D object. As such sub-parts of this known 3D object, we refer to as sub-assembly.

Assembly Video. The corresponding video tutorial depicting a (dis)assembly of the assembly. Note that our use-case assumes that an expert is recorded (dis)assembling the object resulting in high-quality content as we already mentioned in Chapter 1.

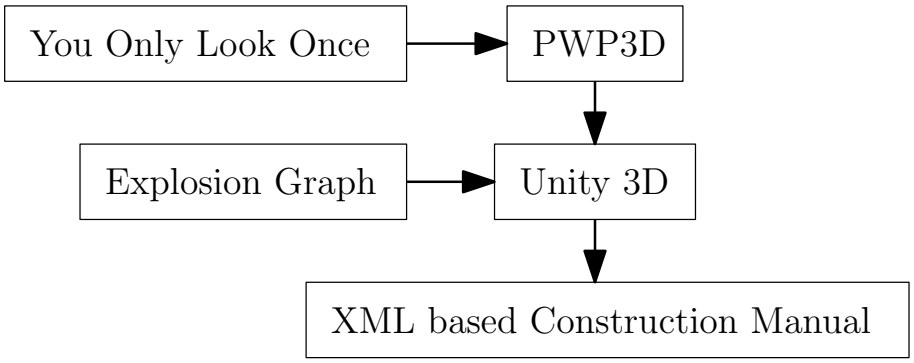
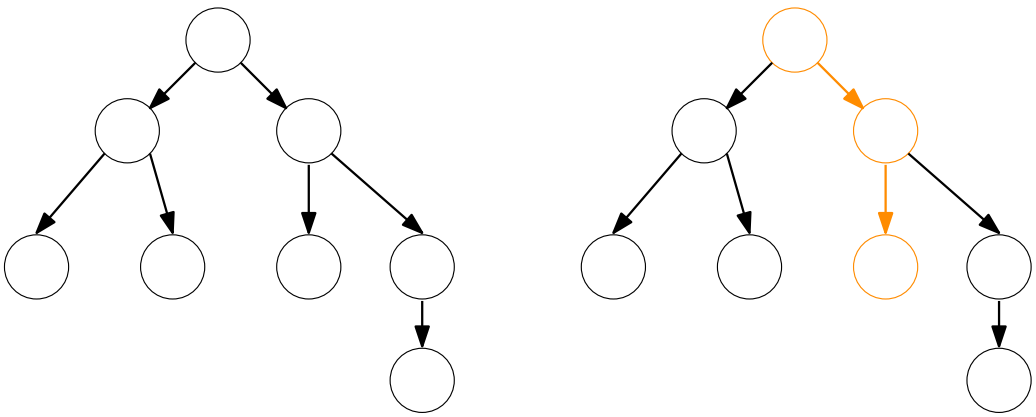


Figure 3.1: Following is an overview of the system, this Figure illustrating its workflow. The results of processing 2D information with You Only Look Once (*YOLO*), together with the 3D object, are used in PWP3D to calculate 6 *DoF* poses. An Unity3D uses the explosion graph corresponding to this 3D object and these 6 *DoF* poses as basis to build a XML based construction manual.

Constraints Assembly Graph. In Section 2.3.1 we have described explosion graphs. In the context of this thesis, it is more fitting to refer to this explosion graph as constraints assembly graph. Based on the constraint assembly graph one can build a graph which depicts all possible (dis)assembly sequences of an assembly.

Author Assembly Graph. The author assembly graph refers to a specific traversal through this graph which depicts all possible (dis)assembly sequences – see Figure 3.2. This traversal corresponds to the (dis)assembly sequence depicted in the assembly video.



(a) All Possible (Dis)assembly Sequences

(b) Author Assembly Graph

Figure 3.2: Based on a constraint assembly graph, one can build a graph which encodes all possible (dis)assembly sequences. The author assembly graph in 3.2b describes a specific traversal through this graph in 3.2a which encodes all possible (dis)assembly sequences.

3.2 Requirements

There are two requirements for the system we propose. We need a 3D representation of the object we want to create an XML construction manual for and we need a video tutorial. Following are some details.

Assembly. The 3D representation has to match the actual object as much as possible. That means the assembly needs a mesh that at least has the same proportions as the actual object. Textures are also required as parts of the system rely on color information.

Assembly Video. Note that *YOLO* and PWP3D operate on pixels and rely on color information. Therefore, the lighting in the assembly video should not be over- or underexposed. The camera has to be static. Hence it should be in a position where every sub-assembly is clearly visible, and the distance to the assembly is not too far off. The type of camera itself does not matter as much as long as it can record a high enough quality video, which most smartphones nowadays are capable of. Note that the intrinsic camera parameters have to be known or calibrated as shown by Heikkilä and Silvén [13] and Pollefeys et al. [27], because PWP3D requires them for tracking.

3.3 2D Tracking

The entry point of our proposed system is You Only Look Once (*YOLO*), a Windows version [3]. In this section, we outline the necessary steps for our system to be able to 2D track assemblies – specifically every sub-assembly of an assembly. Note that PWP3D needs an initial frame for 2D-3D pose estimation. After training *YOLO* and validating the resulting weights, the best weight is chosen. Using the best weight, we feed the assembly video to *YOLO* resulting in a `(class name, prediction rate, predicted bounding box)` triple for each detected object class for each frame of the assembly video. Note that multiple detections for the same object class are possible during the same frame. After filtering this information, we algorithmically pick one frame for each sub-assembly where the sub-assembly is not occluded. Hence we get the required initial frames for 2D-3D pose estimation with PWP3D.

3.3.1 Parameters and Configuration

We only made minor adjustments to the existing *YOLO* configuration file `yolo.cfg`. For one we raised `width` and `height` from 416 to 608. Note that by design `width` and `height` should be set to the same value, while the actual value must be a multiple of 32. The advantage of a higher input image resolution on the one hand is the improved quality of the individual class features *YOLO* trains. The drawback of a higher input image resolution on the other hand is the higher *GPU* memory usage. If you not yet use your *GPU* memory to the full or simply can replace your graphics card with one that has more built-in memory, there is no drawback at all. For every other case, the *GPU* memory that

YOLO requires during one training iteration can be adjusted by compensating higher input image resolution with higher values of `subdivision` in the *YOLO* configuration.

$$\text{sub-batches} = \frac{\text{batch}}{\text{subdivision}}$$

Together with `batch`, `subdivision` determines how many images are processed at a time during one training iteration of *YOLO*. Higher values of `subdivision` result in higher values of `sub-batches`, which reduces the strain on the *GPU* memory as less images are loaded and processed at a time. Note though that the more `sub-batches` there are, the longer one iteration takes to complete. In general, you want to configure *YOLO* s.t. you utilize your *GPU* memory to the full to improve the quality of the individual class features *YOLO* trains.

So far we have discussed entries of the configuration file that trade quality of trained features for *GPU* memory usage. Following we detail entries of the configuration file that are specific to the dataset that is used for training. The values for the entries `classes` and `filters` are calculated as follows [3]:

$$\begin{aligned} \text{classes} &= \{\#\text{ of classes}\} \\ \text{filters} &= (\text{classes} + 5) * 3 \end{aligned}$$

Note that both `classes` and `filters` depend on the number of object classes contained in the dataset.

Another entry that is specific to the dataset used for training is the `anchors` entry. The `anchors` represent initial sizes for the bounding boxes *YOLO* predicts. As stated by Redmon and Farhadi [33, chapter *Bounding Box Prediction*, pp 1] the most likely `width` and `height` ratios in the dataset are used as `anchors`. See Listing 3.1 for the command which invokes the built-in calculation of the `anchors` for *YOLO*.

For a more concrete overview of the original `yolo.cfg` see Listing B.1. Also note Listing B.2 for a more concrete overview of our adjusted configuration of *YOLO*.

```
darknet.exe detector calc_anchors {path}/{dataSetName}.data
-num_of_clusters 9 -width {width} -height {height}
```

Listing 3.1: YOLO Command To Calculate Anchors

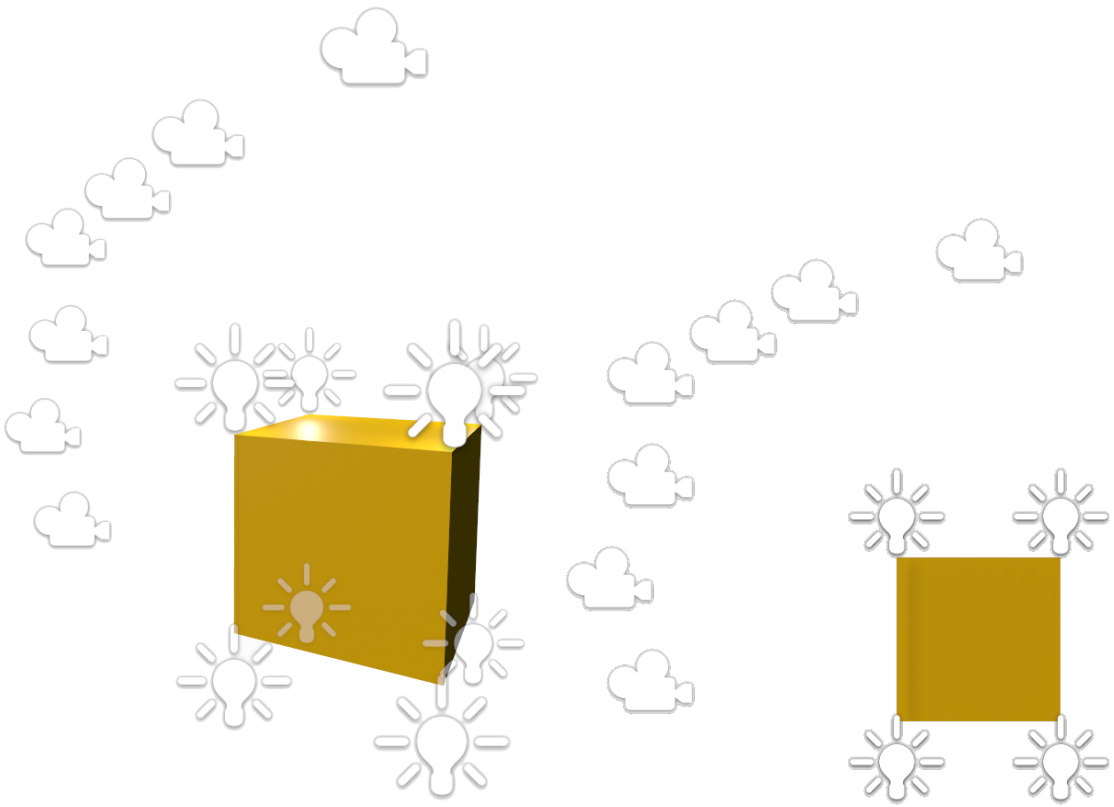
3.3.2 Creating a Synthetic Dataset

In Section 2.1.1 we outlined how synthetic datasets can be created by using renderings of 3D models. Keeping in mind our goal of creating a fully automatic system, we propose training *YOLO* on a synthetic dataset as prior research by Jensen and Selvik [15]. For this purpose, we created a Unity3D application which takes as input all sub-assemblies which we want *YOLO* to be able to detect.

3.3.2.1 Scene Setup

Figure 3.3 shows the general scene setup. There are seven cameras, starting at the top over to the side of the object, plus eight spotlights, placed at the corners of the sub-assembly. Note Figure 3.3b where for each camera also its field of view is indicated.

Using Unity3D to create renderings for our synthetic dataset is a trade-off of runtime and quality of renderings. Since Unity3D is a real-time engine, we can produce synthetic datasets very fast, but do not have the quality of renderings that would be possible with, e.g., Blender. We made this choice to have faster iteration cycles. That being said, the quality of the renderings of Unity3D is quite good as they can be improved by applying, e.g., post-processing effects like Antialiasing, Ambient Occlusion and others.



(a) Scene Setup Perspective View **(b)** Scene Setup Orthogonal View

Figure 3.3: The scene setup we use for creating synthetic datasets showing a generic cube. 3.3a shows a perspective, while 3.3b an orthogonal view of the same scene. We use each of these cameras for sampling images when creating a synthetic dataset. Note that the camera array is rotated during this process s.t. the object is sampled from different perspectives.

3.3.2.2 Workflow

Initially, all sub-assemblies are imported into the application. Then for each sub-assembly, the camera array rotates around it and at each step every camera takes one screenshot. Additionally, the intensity of all spotlights, the camera positions along forward and the background is randomized – see Algorithm 1 for a more precise overview.

Algorithm 1: Creating a Synthetic Dataset for YOLO

```
Result: Synthetic Dataset
foreach SubAssembly do
    foreach Rotation of numberOfFullRotations do
        foreach Step of ⌊360°/stepSize⌋ do
            foreach Camera c do
                RandomizeCameraAlongForward();
                RandomizeSpotLightIntensities();
                PickRandomBackground();
                CalculateObjectBounds();
                TakeScreenshot();
                Rotate(c, stepSize);
SaveDataset();
```

Parameters mentioned in Algorithm 1 like `stepSize`, `numberOfFullRotations` and `BackgroundImages`, as well as other parameters like `width` and `height` can be adjusted to fit the current setup. Note how `width` and `height` of the images in the resulting synthetic dataset should be of the same ratio as the ratio of the assembly video. That is because each frame of the video is fed as input to *YOLO*, which downsamples the images during training and testing according to what is set in the `yolo.cfg`. In the end it is recommended that for each object class there are at least 2000 images in the dataset.

The basic structure of every *YOLO* dataset is the same. Following is a definition of this folder-based structure. Note that {`datasetName`} is replaced by the actual name of the dataset.

- `backup/`. *YOLO* is setup to save its weights every 100 iterations in this directory.
- `Images/`. This directory holds all images contained in the dataset. Note that all labels are saved near their related image.
- {`datasetName`}`.data`. A plaintext file describing the buildup of the dataset. It defines the number of classes and the paths to `backup`, `train.txt`, `valid.txt` and the `names` file.
- {`datasetName`}`.names`. One line in this file corresponds to one class' name in the dataset.

- `traint.txt`. Each line in this file contains the path to one image that will be used for training.
- `valid.txt`. Similar to `train.txt`, each line in `valid.txt` contains the path to one image that will be used for validation.
- `yolo.cfg`. The *YOLO* configuration file, which we detailed in Section 3.3.1.

The scene setup shown in Figure 3.3 and the randomizations presented in Algorithm 1 guarantee that each object is captured in different scales, different rotations under different lighting conditions and different backgrounds. This ensures robust object tracking by *YOLO*.

A more detailed explanation of the `CalculateObjectBounds` method in Algorithm 1 is shown in Algorithm 2. The vertices of the 3D object are first transformed to 2D camera screen space coordinates. Then, based on the minimum and maximum 2D coordinates, the camera specific 2D bounds of the 3D object are calculated. Listing 3.2 shows how *YOLO* labels its images and see Listing 3.3 for an example. Given the resulting 2D bounds of `CalculateObjectBounds`, we only have to calculate its center to be able to label an image such that *YOLO* understands it. There exists one *YOLO* labeling file per annotated image. Multiple annotations per image are represented by one labeling entry per object.

Algorithm 2: Calculating 2D Bounds

Input: A camera looking at a 3D object and the vertices of the same 3D object
Result: Camera specific 2D bounds of the given 3D object

```
vertices2D = VerticesTo2dSpace(camera, vertices3D);
min2D = Min(vertices2D);
max2D = Max(vertices2D);
size.x = max2D.x - min2D.y;
size.y = max2D.y - min2D.y;
bounds.upperLeft = min2D;
bounds.size = size;
```

```
{class.ID} {center.x} {center.y} {size.x} {size.y}
```

Listing 3.2: YOLO Labeling Format

```
0 0.507894 0.4906442 0.1222284 0.4406239
```

Listing 3.3: YOLO Labeling Format Example

After setting up the parameters and creating the dataset, *YOLO* can start training on the synthetic dataset and afterward validating the resulting weights. Note that the Unity3D application randomly picks 20% of all the images as the validation set.

3.3.3 Training and Validating

Assuming that a synthetic dataset was created based on the given assembly, the next step is the training with *YOLO*. As previously mentioned in Section 3.3.2, every 100 iterations *YOLO* creates one weight file as backup. Validation aims to pick the best weight of these weights for detection.

During the training of a *CNN*, at some point the question of when to stop training arises. For this purpose, one can either define a convergence criterion based on the average error of each iteration or define a maximum number of training iterations. For our purposes, we chose the latter and set

$$\text{maximum\_iterations} = \{\#\text{ of classes}\} * 2000$$

as indicated by AlexeyAB [3].

After training finishes, we validate every weight resulting in a (mAP, IoU) tuple for each weight. We opt to chose the best weight for detection based on a good (mAP, IoU) tuple. The mAP and IoU are defined based on the following definitions stated by Everingham et al. [10][3, 24].

- **True Positive (TP)**. A correct detection. Detection with $IoU \geq$ `threshold`.
- **False Positive (FP)**. A wrong detection. Detection with $IoU <$ `threshold`.
- **False Negative (FN)**. Ground truth was not detected.
- **True Negative (TN)**. This equals the case of all possible bounding boxes being correctly not detected. It represents a correct misdetection, which does not apply for the following metrics.

The `threshold` in *YOLO* is set to 0.24 per default.

Mean Average Precision (*mAP*). As stated by Everingham et al. [10, chapter *Evaluation of Results*, pp 11], the average precision is a measure for the area under the precision/recall curve and is defined as the mean precision of eleven equally spaced recall levels $[0, 0.1, \ldots, 1]$:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\text{all detections}}$$
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{all ground truths}}$$

$$\text{AP} = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 1\}} p_{\text{interp}}(r)$$

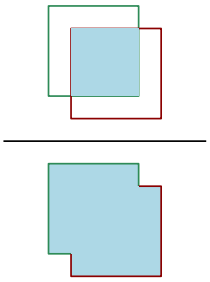
with

$$p_{\text{interp}}(r) = \max_{\widetilde{r} \geq r} p(\widetilde{r})$$

where $p(\widetilde{r})$ is the measured precision at recall $\widetilde{r}$. Following the mean average precision is the average of maximum precision at these 11 recall levels [14, 24].

Intersection-over-Union (*IoU*). The Intersection-over-Union measures how good a prediction actually is. Looking at the illustration below, which shows the relation of a predicted bounding box in red and a ground truth bounding box in green and their area of overlap [24]. The *IoU*, combined with a predefined `threshold`, determines if a detection is correct or wrong, see the definition of TP and FP above.

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}} =$$



We chose to prioritize *mAP* over *IoU*, meaning we pick the weight with the best overall *mAP*. In case there are two weights with the same *mAP* we pick the one with the higher *IoU*.

3.3.4 Interface to PWP3D

After having chosen the best weight for detection, *YOLO* is used over the assembly video resulting in a prediction for every frame of the video. Note that a prediction of one frame possibly consists of multiple detections of the same or different object classes. We made slight changes to the output format of *YOLO* such that it is easier to process its results. Processing all these predictions over the assembly video, our goal is to filter one good frame for each object class which we then can use to initialize PWP3D.

The basic idea of Algorithm 3 is to look for consecutive predictions over `framesRangeThreshold` frames. Note that `lookupRange` usually is set to a value way lower than `framesRangeThreshold`, in case the predictions of *YOLO* fall under the `predictionThreshold` for a few frames during a good detection streak. We aim to filter detection streaks for each object class and save the frame with the highest prediction value of such a streak. Finally, each class has a list with its highest prediction values, and we pick the frame of the entry with the highest prediction value in each list to be used as an initial frame in PWP3D.

Another way of looking at our filtering process is to reason in terms of occlusions why an object was detected over consecutive frames. An object is detected with low probability or not at all if it diverges too heavily from what *YOLO* was trained to detect based on

the given dataset. Since we did not train *YOLO* to detect partially occluded objects, we reason that our method of filtering detection streaks only filters such consecutive frames where the detected object is likely not occluded at all.

Having gone through the whole process of setting up *YOLO*, building a synthetic dataset, training this dataset, validating the results of the training, using *YOLO* to detect objects over the assembly video and finally filtering its results, we now obtain one specific frame of the assembly video for each sub-assembly we are interested in tracking its 6 *DoF* pose. The chosen frames are very likely to depict an unoccluded sub-assembly which is ideal for initializing PWP3D.

Algorithm 3: Filtering Good Frame for PWP3D Initialization

```
Input: Predictions for each frame of the assembly video; predictionThreshold;
       lookupRange; framesRangeThreshold
Result: One good frame for each object class
foreach class of ObjectClasses do
    predictions = allPredictionsOver(class, predictionThreshold);
    // picking the best prediction, if there are multiple detections
       of the same class in one frame
    uniquePredictions = uniquifyClasses(predictions);
    foreach uniquePrediction of uniquePredictions do
        frame1 = GetFrameAtOffsetStartingAt(uniquePrediction, -lookupRange);
        frame2 = GetFrameAtOffsetStartingAt(uniquePrediction, lookupRange);
        framesRange = abs(frame1 - frame2);
        if framesRange ≤ framesRangeThreshold then
            bestPredictions(class).Append(PickBestPredictionInRange());
GetBestPredictions(bestPredictions);
```

3.4 2D-3D Pose-Tracking over Assembly Video

In the previous Section 3.3 we described how we chose the initial frames for each sub-assembly required for 2D-3D pose tracking. The next step in the pipeline of our proposed system is the 2D-3D pose tracking over the assembly video with PWP3D. First we outline the steps necessary to initialize PWP3D, then we aim to estimate a pose for every frame of the assembly video for each sub-assembly one at a time.

3.4.1 Intrinsic Camera Parameters

Part of the requirements of PWP3D is that the intrinsic camera parameters of the device used to capture the video that is fed into PWP3D have to be known. One of the most common ways to calibrate a camera is to use OpenCV [5]. Since OpenCV is compiled for

multiple platforms, there exist camera calibration applications for every current platform, if you just look for them on, e.g., app stores or GitHub.

Usually some type of checkerboard is printed on paper, and, e.g., OpenCV extracts the intrinsic camera parameters of the device by correlating features of this checkerboard in snapshots made from different angles. The calibration extracts intrinsic camera parameters like focal length (f_x, f_y) and optical center (c_x, c_y). Those parameters can be summarized in the 3x3 matrix

$$\text{camera matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 0 \end{bmatrix}$$

which can be used to remove distortion from every image captured by the calibrated camera [27].

Additionally to the focal length and optical center the camera calibration files of PWP3D also contain the resolution that was used during calibration. See Listing 3.4 for such a PWP3D camera calibration and Listing 3.5 for an example with actual values. Note that once a camera is calibrated for some resolution, it can be transformed into a calibration of any other resolution.

```
Perseus_CalFile
{width} {height}
{f_x} {f_y}
{c_x} {c_y}
```

Listing 3.4: PWP3D Camera Calibration File

```
Perseus_CalFile
640 360
530.019   530.019
319.5     179.5
```

Listing 3.5: PWP3D Camera Calibration File Example

For example, we opted for an Android smartphone to record our sample assembly videos. Hence we used an Android application to calibrate the camera [34].

3.4.2 Initializing PWP3D

For 2D-3D pose estimation, an initial frame and an initial 6 *DoF* pose for the sub-assembly in this initial frame is required, as further described in Section 2.2. Additionally to this initial frame and initial 6 *DoF* pose, PWP3D also requires an initial mask. Following we describe how we manually set these 6 *DoF* poses and automatically create these initial masks.

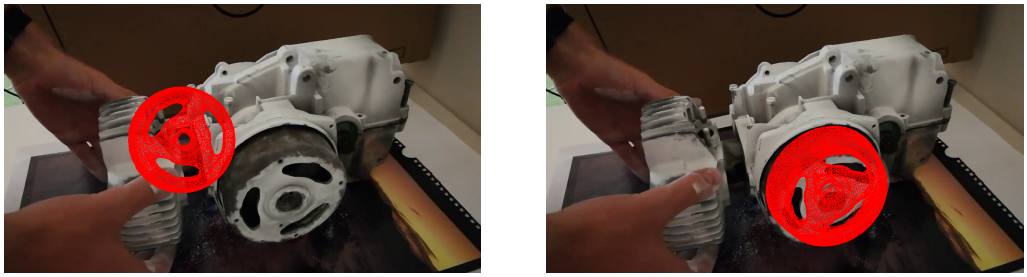
3.4.2.1 Setting an Initial Pose

At this point we are not able to automatize this process, so we manually set the initial 6 *DoF* pose required for 2D-3D pose tracking with PWP3D. Note that PWP3D has its own rendering implementation, at each iteration of 2D-3D pose tracking returning visualizations relevant to this process. Among others, this includes a visualization of the

current frame with the wireframe of the tracked sub-assembly in its current pose rendered on top. We use this specific visualization when manually setting an initial pose.

Before starting 2D-3D pose tracking, we run a pre-processing step in which no new frames are fed to PWP3D, and the optimization used for 2D-3D pose tracking is disabled. Basically, we run PWP3D for its visualizations only. Iteratively we look at the wireframe visualization of the current pose of the sub-assembly and adjust its position and rotation until it fits the 2D representation of the same object in the current frame.

See Figure 3.4 for this wireframe visualization of PWP3D showing a visualization when first running this pre-processing step on the initial frame and the manually set final-initial-pose of the sub-assembly in this initial frame.



(a) Step 1: Initial Pose **(b)** Step 2: Final Initial Pose

Figure 3.4: We run PWP3D in visualization mode only, the optimization related to 2D-3D pose tracking disabled, not feeding any new frames to the PWP3D. Iteratively we manually adjust the position and rotation of this 3D object whose wireframe is rendered on top of the initial frame. In the end this wireframe fits the 2D representation of the same object in the initial frame as shown in Figure 3.4b, giving us the initial 6 *DoF* pose PWP3D requires for 2D-3D pose tracking.

3.4.2.2 Creating an Initial Mask

The second part of the initialization of PWP3D is the creation of an initial mask. We can create an initial mask automatically given the initial frame and initial 6 *DoF* pose of the 3D object we want to 2D-3D pose track.

Similarly to the manual setting of the initial pose, which we described in Section 3.4.2.1, we use parts of the rendering implementation of PWP3D that returns visualizations during each iteration of the 2D-3D pose tracking. The specific visualization we use for creating an initial mask is a depth-image encoding depth-information about the 3D object in its initial pose.

We create a new initial mask every time before we start the actual 2D-3D pose tracking in a pre-processing step. Using this depth image and the following functions provided by OpenCV, we can create an initial mask:

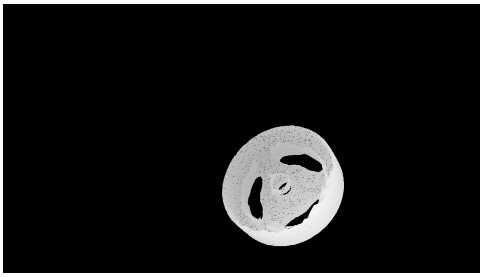
1. `OpenCV.normalize(..., NORM_MINMAX)`

2. `OpenCV.bitwise_not(...)`

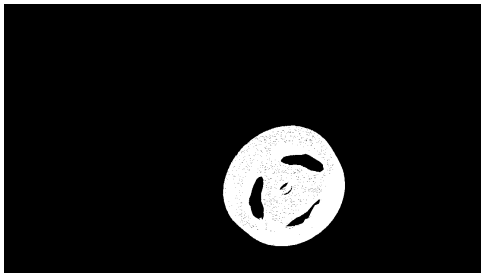
3. `OpenCV.threshold(..., THRESH_BINARY)`

First we `normalize` the values we get from PWP3D resulting in a depth image as shown in Figure 3.5a. Applying `bitwise_not` simply inverts the depth image as depicted by Figure 3.5b. Finally, we use binary thresholding to get a binary image from our grayscale image. The final result of these operations in the proposed order results in an initial mask as illustrated in Figure 3.5c. Note that the resulting initial mask corresponds to the initial 6 *DoF* pose of the 3D object in the initial frame shown in Figure 3.4b.

(a) Step 1: Depth Image



(b) Step 2: Inverting Depth Image



(c) Step 3: Thresholding Depth Image

Figure 3.5: By using OpenCV and the initial 6 *DoF* pose of a 3D object in its initial frame, we are able to automatize the creation of an initial mask which is required by PWP3D for its 2D-3D pose tracking of this 3D object. First we normalize the results of the PWP3D depth calculation as shown in Figure 3.5a. After inverting this image in a second step and applying a binary threshold afterwards, we get the final initial mask depicted by Figure 3.5c.

3.4.3 Pose Tracking

After manually setting the initial 6 *DoF* pose and the automatized creation of a related initial mask, every requirement for 2D-3D pose tracking with PWP3D is met. Following is a complete overview of those requirements:

- sub-assembly (3D object)
- initial frame
- initial mask
- initial 6 *DoF* pose of the sub-assembly in the initial frame
- assembly video
- intrinsic camera parameters of the device the assembly video was captured with

Finally feeding all this information into PWP3D, we then get a 6 *DoF* pose for each frame of the assembly video. Note that we 2D-3D track one sub-assembly at a time. Using the initial frame as a starting point, we move forward and backward through the assembly video. By further processing of the gathered information, we want to create the author assembly graph in the next step.

3.5 Building the Author Assembly Graph

In previous sections, we explained the flow of information through different software frameworks and the resulting outcomes. This following section will go into more detail on how the results of the 2D-3D pose estimation with PWP3D and a given constraint assembly graph are used to build an author assembly graph. Note our definition of a constraint assembly graph in Section 3.1 and its relation to author assembly graphs depicted in Figure 3.2. We further pre-define basic patterns which our system uses to detect constraining components. That enables the author assembly graph also to feature what we call tool constraints. Author assembly graph based tutorials are then not only able to show a (dis)assembly sequence but also correctly indicate where tools have to be applied at each step.

In the following section, we will only refer to disassemblies and disassembly sequences for the sake of readability. Note that the algorithms and methods described can also be applied on assembly sequences; it depends on your point of reference.

Graph structure. For the constraint assembly graph we use a graph structure based on the Directed Graph Markup Language (*DGML*) by Microsoft [21] as it is a very flexible standard. Most notably is that this graph structure only saves `Nodes` and `Links`. The advantage of only saving lists of the elements `Node` and `Link` is the flat and more readable hierarchy in the resulting serialized file. Although a *DGML* graph is saved in a flat

hierarchy, through `Links` the directed structure of the graph itself is encoded. Note that one such `Link` contains the two `Id`s of the two `Node`s it connects. The original *DGML* standard also features other elements such as `Category` or `Property` and additional attributes like `Label`, `Background`, `Stroke`, etc. which are not needed for our purposes. For an example of such a *DGML* graph see Listing B.3.

The application. For the actual application, we use Unity3D to build the author assembly graph. Using Unity3D allows us to more easily illustrate applications of the author assembly graph and still have everything essential gathered in one application.

3.5.1 Finding Movement Candidates

The first step in building the author assembly graph is the filtering of the results of the 2D-3D pose estimation with PWP3D described in Section 3.4. We are interested in knowing when a sub-assembly starts moving because a moving sub-assembly indicates that this object is being removed from the assembly right now. We calculate multiple so-called start-of-movement-candidates for each sub-assembly.

Later we validate the start-of-movement-candidates against the constraint assembly graph to get the author assembly graph. In the end, we want to build an author assembly graph which describes the same disassembly sequence as depicted in the assembly video.

Algorithm 4 shows how the start-of-movement-candidates for one sub-assembly are calculated. By determining when a sub-assembly starts moving, we later get the actual order of removal by validating against the constraint assembly graph. Note that in Algorithm 4 we first calculate the distances between the estimated pose positions of every frame. One way of looking at this list of distances is as a time series and the problem of determining when a sub-assembly starts moving turns into a problem of finding peaks in this time series. For this reason, we apply the Z-Score algorithm by van Brakel [43] on this time series of distances. The results of this Z-Score algorithm are then filtered to get the start-of-movement-candidates. Note that the Z-Score algorithm returns either $\{0, 1, -1\}$ for each entry in the time series. We filter the positive peaks, which indicate a moving sub-assembly to get the final list of start-of-movement-candidates. We use Algorithm 4 to calculate the start-of-movement-candidates for each sub-assembly.

Algorithm 4: Filtering Sub-Assembly Start of Movement Candidates

Input: Pose for each frame of the assembly video for a sub-assembly; Z-Score parameters: lag, influence, threshold
Result: Start of movement candidates for this sub-assembly

```
// VideoPose is a tuple (frame, position, rotation)
foreach videoPose of VideoPoses do
    distances.Append(
        (Previous(videoPose).position - videoPose.position).magnitude);
peaks = ZScore(distances, lag, threshold, influence);
// either 0, 1, -1 is signaled by ZScore; we only care about the
   positive signals
candidates = GetPositivePeaks(peaks);
```

Z-Score algorithm by van Brakel [43]. The algorithm constructs a separate moving mean and standard derivation and signals a peak if a new datapoint is x number of standard derivations off the current mean. Following we detail the input parameters of the algorithm:

- `lag`: Defines the number of prior datapoints considered in the calculation of the moving mean and standard derivation. Essentially, low values of `lag` allow the algorithm to more quickly adapt to new trends.

- `influence`: Determines the influence of peaks on the algorithm's detection threshold. Setting `influence` to, e.g., 0 assumes stationarity and the calculation of the moving mean and standard derivation stays uninfluenced.

- `threshold`: New datapoints are detected as peaks if they are a `threshold` number of standard derivations off the current mean. Generally lower `threshold` values will result in more detected peaks, while higher `threshold` values in less detected peaks.

3.5.2 Validating against the Constraint Assembly Graph

We mentioned that by knowing when a sub-assembly starts moving, we can establish an order of removal by validating against the constraint assembly graph. Note that the directed graph structure of the constrained assembly graph is based on the Directed Graph Markup Language (*DGML*) as previously described. We further enhance a `Node` of this graph by adding the element `UnconstrainedDirection`. This `UnconstrainedDirection` is related to the sub-assembly the `Node` represents. It is a direction vector indicating in which direction the sub-assembly can be removed without being blocked or constrained by any other parts of the assembly. This graph definition is shown in Listing 3.6.

```
<ConstraintAssemblyGraph>
    <Nodes>
        <SubAssemblyNode id="...">
            <UnconstrainedDirection>
                <x>...</x>
                <y>...</y>
                <z>...</z>
            </UnconstrainedDirection>
        </SubAssemblyNode>
        ...
    </Nodes>
    <Links>
        <Link sourceNodeId="..." targetNodeId="..." />
        ...
    </Links>
</ConstraintAssemblyGraph>
```

Listing 3.6: Constraint Assembly Graph Definition

Note that in Algorithm 5 we are referring to a candidate as an object with the tuple `(nodeId, start-of-movement-candidates)`. Algorithm 5 describes the actual validation against the constraint assembly graph. First, we order all candidates by the frame of their first start-of-movement-candidate to establish a timeframe. After that, we check if the first candidate `CanBeRemoved`. For a candidate/sub-assembly to be removable, it either has to be a leaf in the constraint assembly graph, or its constraining sub-assemblies have all been removed previously. We assert this for each node reachable from the current candidate/sub-assembly. If the current candidate can be removed, we do so by adding this candidate to our chronologically ordered list of instructions and removing all its remaining start-of-movement-candidates. In case a candidate cannot be removed, we remove the corresponding start-of-movement-candidate and order the candidates with `SortByFirstFrame` again to establish a new timeframe. That results in a new candidate at the top of the list which we then again can try to remove. Algorithm 5 terminates when we have a start-of-movement-candidate for each sub-assembly in our list of chronologically ordered instructions, or we run out of start-of-movement-candidates. Previously we mentioned that one can build a graph which encodes all possible disassembly sequences based on the constraint assembly graph. Note that this list of instructions describes a specific traversal through this graph that encodes all disassembly sequences.

Algorithm 5: Validating against the Constraint Assembly Graph

Input: The start-of-movement-candidates of each sub-assembly and the constraint assembly graph.
Result: The author assembly graph.

```
// a candidate contains a tuple of
   (nodeId, start-of-movement-candidates)
// SortByFirstFrame orders the candidates by the first frame of their
   containing start-of-movement-candidates
candidates = SortByFirstFrame(candidates);
for i = 0; i < candidates.Length; do
    candidate = candidates[i];
    if CanBeRemoved(candidate, instructions) then
        instructions.Append(candidate);
    else
        RemoveFirstStartOfMovementCandidate(candidate);
        candidates = SortByFirstFrame(candidates);
        continue;
    i++;
```

3.5.3 Detecting Constraining Components

By applying Algorithm 5 we calculate the order of removal of the sub-assemblies matching the sequence depicted in the assembly video as described in the previous section. In this section, we further enhance the resulting author assembly graph by detecting constraining components.

Constraining component. We define a constraining component as a specific part of a sub-assembly. Such a constraining component can only be removed by using a specific tool.

Tool database. Our system features a fixed set of pre-defined tools. Through pattern matching, we can identify constraining components and derive the required tool for this specific component.

Tool constraints definition. Additionally to the tool database we also pre-define the patterns used for matching. We define one or more patterns for each tool in the tool database. When detecting such a pre-defined pattern in a sub-assembly, we can infer the tool of the tool database required for the disassembly. See Table 3.1 for an example of such a tool constraints definition. The serialized version of the same tool constraints example is shown in Listing B.4.

In Algorithm 6 we use the patterns of a tool constraints definition to identify if a sub-assembly contains constraining components. The algorithm searches through the hierarchical structure of a given sub-assembly to assess if any part of the sub-assembly matches any patterns defined in the given tool constraints definition. At this point, we

Tool of Tool Database	Patterns
`Wrench`	`nut`
	...
`Screwdriver`	`screw`
	...
...	

Table 3.1: An example of a tool constraints definition. One tool is mapped to one or more patterns. We look for constraining components based on the patterns defined in a tool constraints definition. This allows us to infer a related tool, if a pattern is matched in a sub-assembly. We can further enhance the author assembly graph with this additional information.

check if any of the series of literals defined as patterns are part of any names of the parts of a sub-assembly. For example when applying Algorithm 6 with the tool constraints definition in Table 3.1, we detect a constraining component if any part of a sub-assembly matches the pattern `nut` or `screw`, inferring the tools `Wrench` or `Screwdriver` respectively.

Algorithm 6: Matching Tool Constraints

Input: The sub-assembly and a tool constraints definition.
Result: An author assembly graph enhanced by tool constraints.

```
foreach part of sub-assembly do
    foreach constraintDefinition of constraintDefinitions do
        foreach pattern of constraintDefinition.Patterns do
            if Match(part, pattern) then
                instruction.ToolConstraint.Append(constraintDefinition);
```

Building this relation of constraining components in sub-assemblies and our tool database via a tool constraints definition we can further enhance the author assembly graph. Note that by detecting these constraining components we also extract their position in relation to their sub-assembly. Therefore the author assembly graph also contains the positions of these constraining components and tutorials based on such an author assembly graph can correctly indicate which and where a tool is needed for each instruction.

3.5.4 Final Definition of an Author Assembly Graph

The basic structure of an author assembly graph is defined by the ordered list of instructions we generate by validating the start-of-movement-candidates against a constraint assembly graph. Each instruction is describing how its related sub-assembly can be removed from the overall assembly, the order of this list indicating the order of removal of

these sub-assemblies. We include the unconstrained direction of the related sub-assembly, obtained from the constraint assembly graph, in an instruction of an author assembly graph. Note that the unconstrained direction is a direction vector indicating in which direction a sub-assembly can be removed without being blocked or constrained by any other parts of the assembly.

Using a tool constraints definition we can relate the tools of our tool database to constraining components. By detecting constraining components, we infer not only the tool necessary for further disassembly, but also the positions of these constraining components. We further enhance each instruction of an author assembly graph by adding information about constraining components, if any are detected. That allows indicating where tools have to be applied at each step of a disassembly.

In the end each `Instruction` encodes an `UnconstrainedDirection` and a list of `ToolConstraints`. In Listing 3.7 we show a generic serialization of the final definition of an author assembly graph containing `Instructions` ordered by their order of removal.

```
<ArrayOfInstruction>
    <Instruction id="...">
        <UnconstrainedDirection>
            <x/><y/><z/>
        </UnconstrainedDirection>
        <ToolConstraints>
            <ToolConstraint>
                <Tool>...</Tool>
                <ConstraintPosition>
                    <x/><y/><z/>
                </ConstraintPosition>
            </ToolConstraint>
            ...
        </ToolConstraints>
    </Instruction>
    ...
</ArrayOfInstruction>
```

Listing 3.7: The final definition of an author assembly graph. While the unconstrained direction of each instruction is derived from the constraint assembly graph, the tool constraints are inferred by detecting constraining components. The list of instructions is ordered s.t. the first instruction in this list describes the first step of the disassembly sequence, the last instruction the final step of the disassembly sequence.

4 Results

So far we described the motivation of our proposed system and its inputs and workflow. For an overview of all its frameworks and the interfaces between them see Chapter 3. In this chapter, we present the results of using the involved frameworks leading up to the creation of an author assembly graph. We also outline how an author assembly graph can be used to automatically build tutorials for different types of media, as initially proposed in Chapter 1. Specifically we describe the creation of a 2D Print Tutorial and an Augmented Reality Tutorial based on an author assembly graph. While the result of the 2D Print Tutorial is a PDF, which can be printed, the result of the Augmented Reality Tutorial is an Android smartphone application.

4.1 Basic Tutorial Elements

We not only wanted to show that tutorials of two different media types can be built by basing them on the same author assembly graph but also wanted to show that this is possible while also sharing the same basic tutorial elements. In its essence, both the 2D Print Tutorial and the Augmented Reality Tutorial use the same basic set of tutorial elements although one is a static print tutorial and the other a real-time smartphone application. Following we describe these basic tutorial elements in more detail.

User interface. We designed a generic user interface featuring a textbox and a panel used to display additional tool related information. See Figure 4.1 for an illustration of this generic 2D canvas. At each step of an instruction, the textbox is filled with appropriate text to help accomplish the instruction. Optionally we also display a panel illustrating the required tool with its accompanying additional information as text to be able to carry out the instruction.

Tool indicators. In close connection to the optional panel for tool related instructions, we indicate where tools have to be applied. Specifically, we display a ring as billboard rendered on top of every other object. In Figure 4.1 an example of such a billboard is shown. Note that a billboard will always face the main camera, which effectively ensures

that it stays a circle and is not warped into an ellipse through distortion caused by a different perspective.

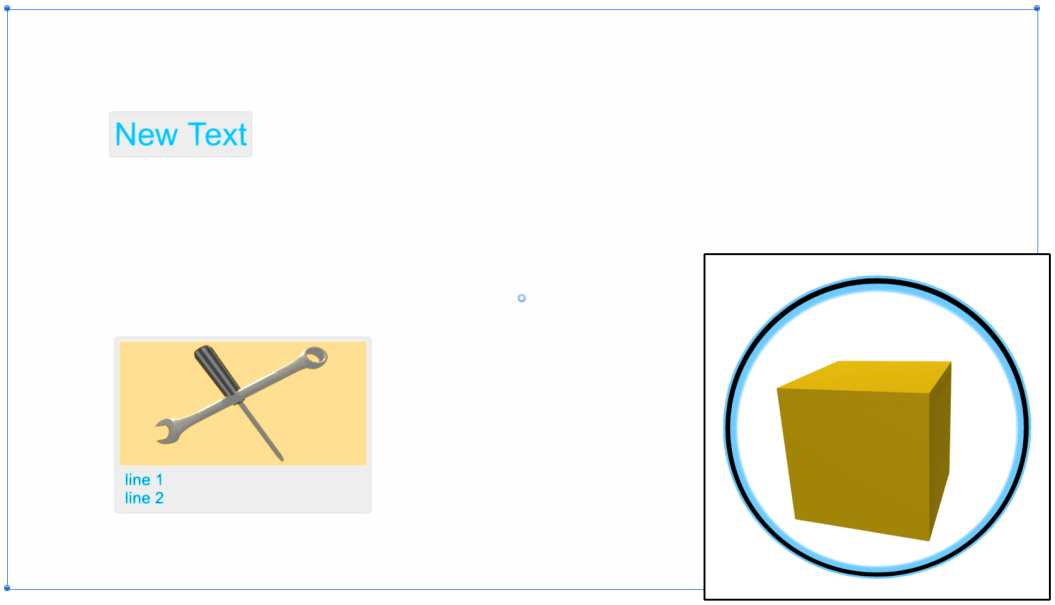
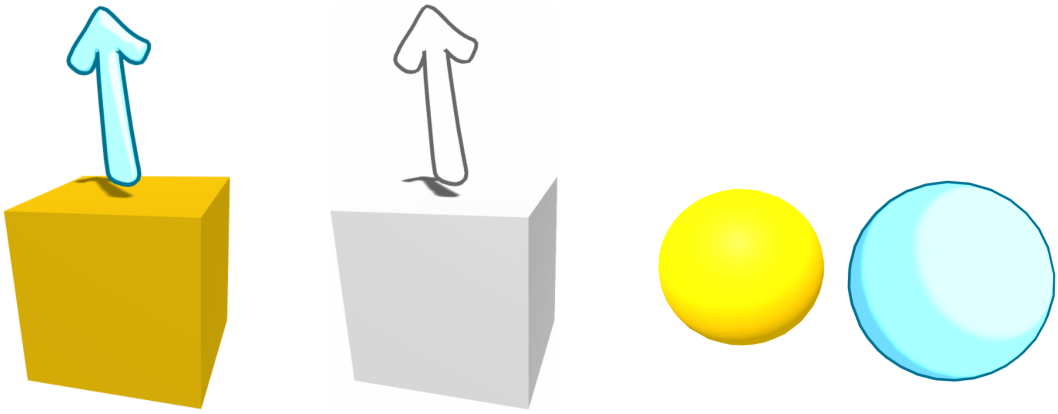


Figure 4.1: The generic 2D canvas is featuring a textbox and an optional panel which displays tool related information if necessary. During each instruction the textbox and panel are set up to display information, offering guidance to the user. The bottom right corner features a cube highlighted by a 2D ring as billboard rendered on top of every other object. We use these billboard rings to indicate constraining components s.t. the user knows where to apply tools following the displayed instruction.

Glyphs. We use 3D glyphs to indicate directions of removal. The specific direction of removal is derived from the unconstrained direction of the current instruction. In Figure 4.2a we illustrate an arrow-style 3D glyph.

Color grading. Regular print tutorials often only use black colors on white paper for their illustrations. We chose to apply color grading to achieve a similar look and feel as is the case for regular construction manuals. For an example of color grading see Figure 4.2b.

Toon shading and outline. It is common in tutorials to highlight the relevant parts of the current instruction. We follow the same principles and highlight the sub-assembly of the current instruction by applying a toon shader. In Figure 4.2c we show a toon shaded sphere, comparing it to a regular sphere with diffuse shading. Note the outline at the edges of the toon-shaded sphere making it more distinct.



(a) 3D Glyph with a Cube

(b) Color Grading

(c) Toon Shading with Outline

Figure 4.2: While Figure 4.2a shows how we use 3D glyphs to indicate directions of removal, the same setup is used in Figure 4.2b to illustrate the effects of black and white color grading. In Figure 4.2c we depict a regular sphere with diffuse shading on the left, and a toon shaded sphere on the right. Note that we use toon-shading to highlight the sub-assembly of the current instruction.

4.2 Engine

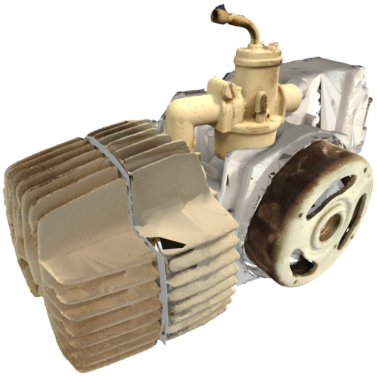
In this section, we detail how we build an author assembly graph for an engine based on an assembly video depicting the disassembly of this engine in the real world. We also illustrate how to create both a 2D Print Tutorial and an Augmented Reality Tutorial based on the same author assembly graph of this engine and the basic elements described in the previous section. Note that the subsections are named and chronologically ordered after the parts and the workflow of our proposed system described in Section 3 and its two applications.

The assembly we use in this example is an engine consisting of the following six sub-assemblies `bing`, `cylinder`, `head`, `lower_shell`, `upper_shell` and `wheel`. See Figure 4.3 for the virtual representation and a photograph of the engine. Note that the 3D representation was created using photogrammetry. While photogrammetry is straightforward to use, it is prone to create mesh artifacts. Hence it is preferable to use a virtual representation created with any *CAD* model software over photogrammetry to not have to deal with these mesh artifacts or acquire, if possible, the original *CAD* data of the assembly.

4.2.1 You Only Look Once

In Section 3.3 we described You Only Look Once (*YOLO*), how we set it up and how to create a synthetic dataset for training. See Figure 4.4 for the default configuration that is used when creating a synthetic dataset. We use 2000 randomly selected background images from the images of the Pascal *VOC* Challenges [10] of 2007 and 2012. Note that

(a) Photograph Engine



(b) Virtual Engine

Figure 4.3: While Figure 4.3a shows a photograph of the engine in the real world, Figure 4.3b shows its virtual representation created with photogrammetry. Note that both the photograph and the screenshot were taken from similar perspectives, making it easier to compare them.

each sampled image will have the resolution 960x540 pixels and the sub-assembly depicted by the sampled image will be annotated algorithmically. In Section 3.3.2 we described the scene setup with its six rotating and its one static cameras and the folder-based structure of a *YOLO* dataset. With the default configuration shown in Figure 4.4 this results in a total of

$$\frac{360}{45} * 60 * \text{rotating\_cameras} + 60 * \text{static\_cameras} = 2940$$

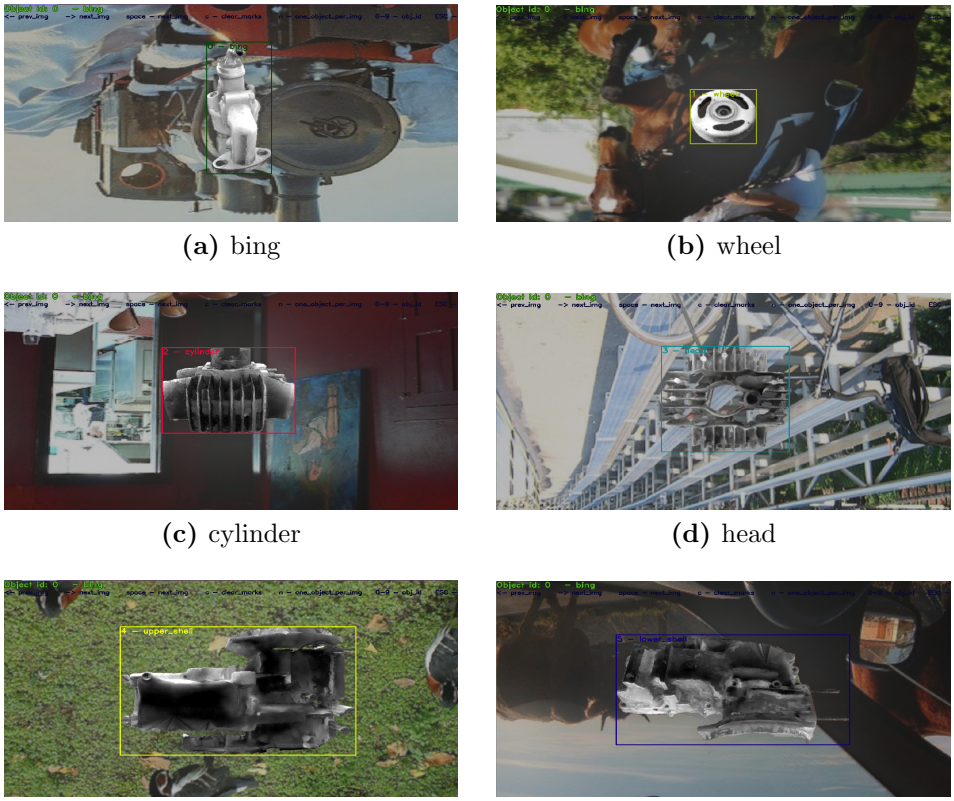
sampled images per sub-assembly which is sufficiently large for a dataset. Note that 20% of all the images in the dataset are used for validation. The actual images for validation are selected at random. When creating such a synthetic dataset, we do so by considering the folder-based structure of *YOLO* datasets. Therefore the created synthetic dataset can be used as is for training with *YOLO*.

Background Images Count	2000
Background Images Path	../BackgroundImages
Background Mat	Background
Pixel Height	540
Pixel Width	960
Rotations	60
Rotation Stepsize	45

Figure 4.4: The default configuration we use for creating a synthetic dataset. Following this configuration the sampled images will have a resolution of 960x540 pixels. The 6 rotating cameras described in the previous chapter will make 60 rotations with a stepsize of 45 degrees resulting in a total of 2940 sampled images per object class. Note that we randomly select 2000 background images from a pool of available background images.

Figure 4.5 shows images from a generated synthetic dataset. There is one example for each of the sub-assemblies. Note the algorithmically annotated bounding boxes which are visualized by YOLO Mark [2]. While YOLO Mark usually is used to manually annotate images, in our case we use the visualized bounding boxes to validate our annotation algorithm.

The latest synthetic dataset we trained contained six object classes corresponding to the six sub-assemblies with 2940 images per object class. In total that were 17640 images of which 3528 randomly selected images were selected as the validation set and the remaining 14112 images as the training set. We trained *YOLO* with this training set for 12000 iterations, 2000 for each object class.



(a) bing **(b)** wheel

(c) cylinder **(d)** head

(e) upper_shell **(f)** lower_shell

Figure 4.5: Annotated images taken from a synthetic dataset visualized by YOLO Mark [2]. One example for every object class in the synthetic dataset. Each object class corresponds to a sub-assembly of the engine. Note that the sub-assemblies were annotated algorithmically during the generation of the synthetic dataset.

	Iterations	mAP	IoU
1	11600	98.35	80.24
2	7800	96.87	77.98
3	10600	96.73	76.30
4	5200	96.39	75.38
5	8000	96.11	79.78
6	5400	95.57	73.85
7	11200	95.36	80.93
8	10500	95.32	76.76

Table 4.1: We present this table with the best weights, evaluating the validation of each trained weight of our latest synthetic dataset. The entries are ordered by mAP and IoU. Note that the training resulted in a high mAP relatively early looking, e.g., at the 5200th iteration with a mAP of 96.39%. That is why we conclude that later iterations are likely to be overfit. For a better overview of the validation and the progress of mAP and IoU through the iterations see Figure 4.6.

Following is a complete overview of the necessary steps that result in detecting an initial frame for each sub-assembly, required for the initialization of PWP3D:

1. Create the synthetic dataset
2. Calculate the anchors of the synthetic dataset
3. Training
4. Validation
5. Detection on the assembly video with the best weight
6. Filter initial frames

Note that all of the above steps are automatized except for the manual selection of the best weight for detection on the assembly video.

We have shown samples from and presented the latest synthetic dataset we trained above and detailed the benefits of anchors in Section 3.3. The results of validating the training with *YOLO* of this synthetic dataset are illustrated by the graph in Figure 4.6. When looking at this graph one can argue that the training could have stopped earlier since a high mAP is achieved quite early, but we set a maximum number of iterations as our convergence criteria as described in the previous chapter. That is why we pick the best weight for detection solely based on the results of validating each weight. Drawing conclusion from Figure 4.6 and Table 4.1 we chose the weight of the 5200th iteration as weight for detection on the assembly video, since weights of later iterations are likely overfit. Note that Table 4.1 shows the best validated weights ordered by mAP and IoU of this synthetic dataset trained with *YOLO*.

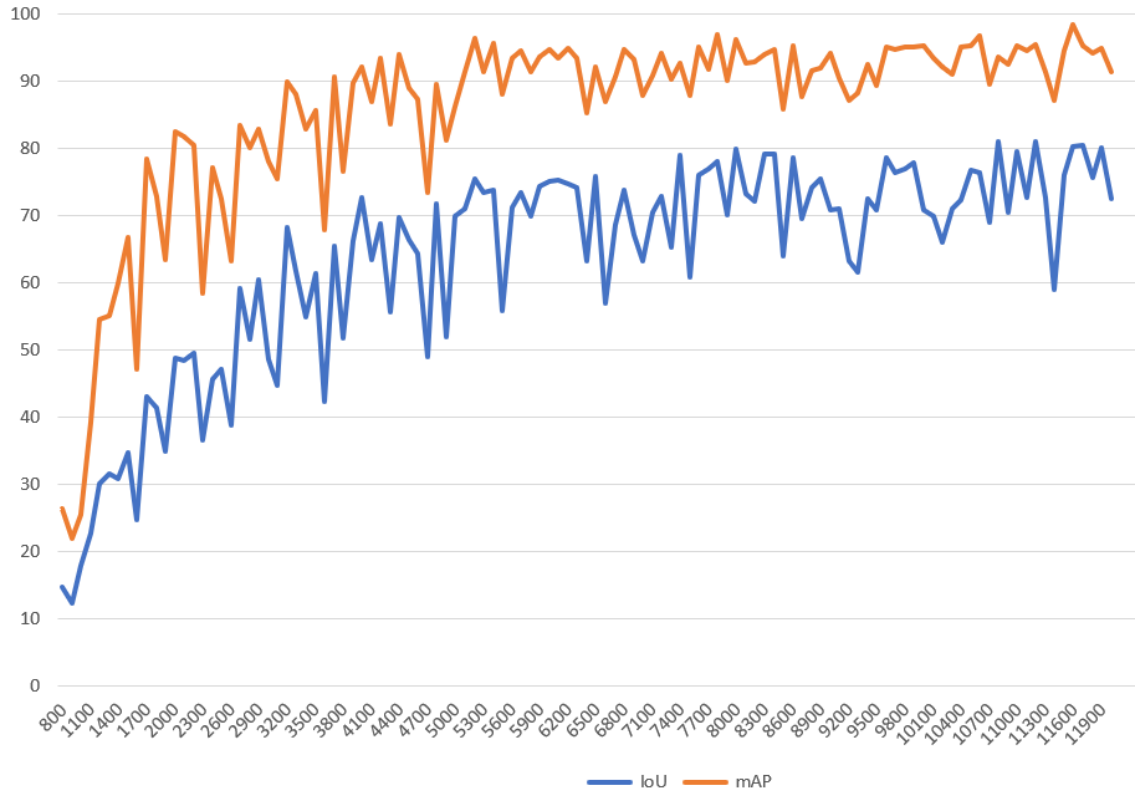


Figure 4.6: This graph depicts the results of validating each trained weight. Note that a high *mAP* is achieved quite early at around the 5300th iteration indicating that the training could have stopped at this point, since later iterations do not improve the *mAP* significantly. This is slightly different for the *IoU* which keeps improving steadily right to the last validated weight. Note that we prioritize *mAP* over *IoU* when picking a weight for detection. We also present an ordered view with the actual values of the validation in Table 4.1.

Feeding the assembly video as input to *YOLO* results in predictions for each frame. In Section 3.3.4 we described Algorithm 3 that finds suitable frames for the initialization of PWP3D. Note that in Algorithm 3 we set

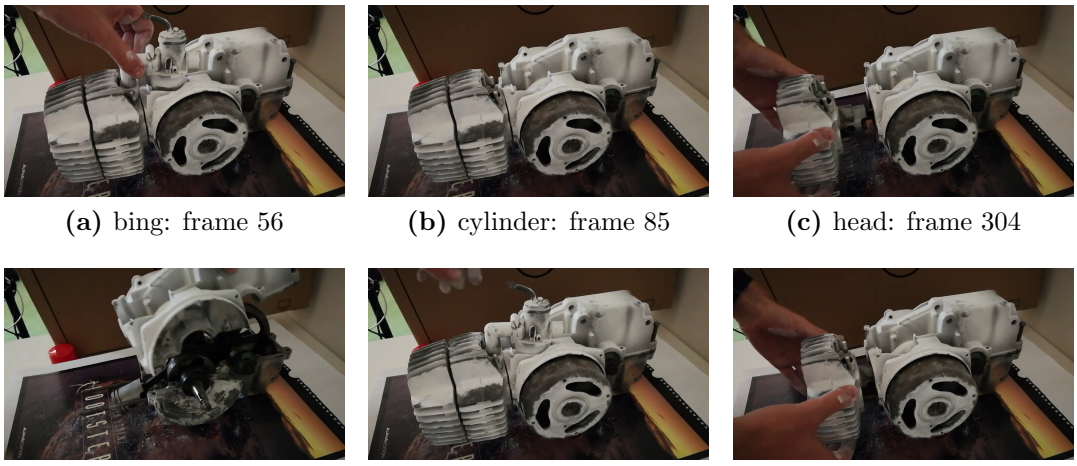
$$\begin{aligned} \text{predictionThreshold} &= 60.00 \\ \text{lookupRange} &= 8 \\ \text{framesRangeThreshold} &= 80 \end{aligned}$$

per default. In Table 4.2 we show the predictions of *YOLO* for the 25th frame of the assembly video. Note that `upper_shell` and `lower_shell` were detected multiple times in this single frame, while `cylinder`, `wheel` and `bing` were detected only once. Algorithm 3 deals with multiple predictions for the same object class by only considering the prediction

upper_shell	lower_shell	cylinder	wheel	bing
99.05%	86.92%	92.49%	79.53%	74.58%
39.54%	32.89%			
95.63%				

Table 4.2: We show an excerpt of the results of the predictions of *YOLO* over the assembly video. Specifically, we show the predictions for the 25th frame of assembly video. Note that `upper_shell` and `lower_shell` were detected multiple times in this single frame, while `cylinder`, `wheel` and `bing` were detected only once. We evaluate all the predictions of all frames by applying Algorithm 3. For the original excerpt see Listing B.5.

with the highest mAP. After iterating over every frame and evaluating the gathered data, one initial frame for each sub-assembly is chosen. The initial frames for the current synthetic dataset after applying Algorithm 3 on the results of the detection of *YOLO* over the assembly video are presented in Figure 4.7.



(a) bing: frame 56 **(b)** cylinder: frame 85 **(c)** head: frame 304

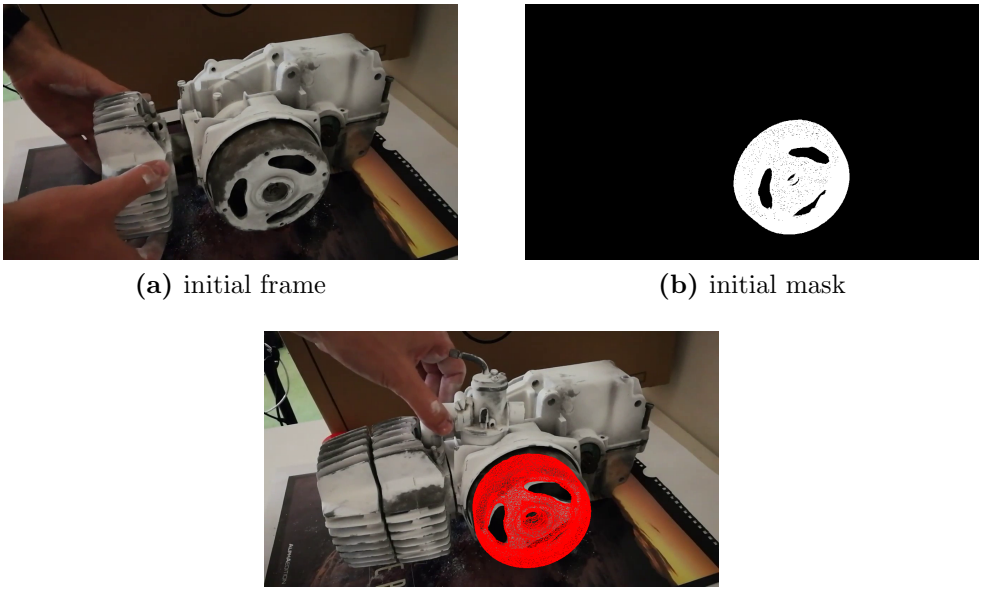
(d) lower_shell: frame 516 **(e)** upper_shell: frame 22 **(f)** wheel: frame 292

Figure 4.7: We present the results of evaluating the predictions of *YOLO* over the assembly video by applying Algorithm 3. The initial frames shown here are the initial frames for 2D-3D pose tracking with PWP3D in the next step. There is one initial frame for each of the sub-assemblies of the engine. The frame number indicates the position of the frame in the assembly video.

4.2.2 PWP3D

In the previous section, we describe how to use *YOLO* to acquire initial frames for each sub-assembly. Using the initial frame as a starting point, we set an initial pose and automatically create a mask for the sub-assembly we want to track throughout the assembly video. Note that our proposed system for creating author assembly graphs in its current

state is not able to estimate an initial 6 *DoF* pose automatically. See Figure 4.8 for the initial frame, initial mask and initial 6 *DoF* pose of the wheel sub-assembly. Note the initial pose visualized by the wireframe of the associated wheel 3D model projected onto the initial frame in Figure 4.8c.



(a) initial frame **(b)** initial mask

(c) initial pose

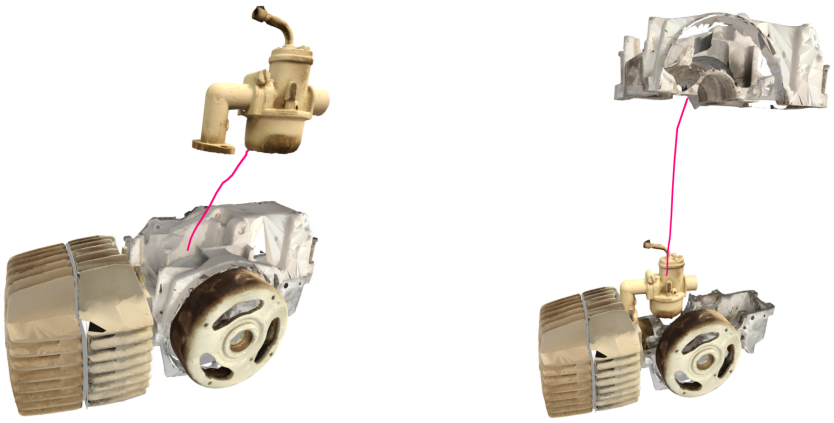
Figure 4.8: The initial frame, initial mask and initial 6 *DoF* pose required of the `wheel` sub-assembly. All sub-assemblies require their own for 2D-3D pose tracking with PWP3D.

See Section 3.4 for a more in-depth explanation of 2D-3D pose tracking with PWP3D. Using the initial frame as a starting point, we track each sub-assembly forward and backward through the assembly video. That results in a 6 *DoF* pose for each frame of the assembly video for the currently tracked sub-assembly. We 2D-3D pose track one sub-assembly at a time. In Figure 4.9 we visualize the results of 2D-3D pose tracking the `bing` and `upper_shell` sub-assemblies. Note that Listing B.6 shows a snippet of the serialized results of 2D-3D pose tracking the wheel sub-assembly. We use the following format to describe the `position` and `rotation` of the tracked object in the current `frame`. Following is a description of the tuple we defined describing the results of 2D-3D pose tracking with PWP3D.

$$\text{frame} : (\text{position.x}, \text{position}, \text{position.z}, \text{rotation}, \text{rotation.y}, \text{rotation.z})$$

We opted to represent a rotation as a three component vector given in degrees. This allows us to more easily validate results due to the superior readability compared to a quaternion representation. Note that the implementation of PWP3D we use [28] also

features a Nvidia® CUDA® implementation. Enabling *GPU* acceleration allows PWP3D to run in real-time as stated by Prisacariu and Reid [29]. Keeping this in mind we are able to calculate 6 *DoF* poses for the whole assembly video very fast, although we process one sub-assembly at a time.



(a) Result Pose Tracking `bing` **(b)** Result Pose Tracking `upper_shell`

Figure 4.9: In Figure 4.9a and Figure 4.9b we visualize the results of 2D-3D pose tracking with PWP3D. We visualize paths between a starting point and a final position where the corresponding sub-assembly is placed.

4.2.3 Author Assembly Graph

So far we presented the results of the different parts of our proposed system leading up to building an author assembly graph for the engine assembly. In this section, we present the results of the final step, the actual creation of an author assembly graph, by applying the algorithms described in Section 3.5.

See Table 4.3 for a full definition of the constraint assembly graph for the engine assembly given by a system based adapted from Li et al. [18]. The format of the graph is a variant of Directed Graph Markup Language (*DGML*) as mentioned in Section 3.5. In Figure 4.10 a visualization of the same graph is depicted. Note the leaves `bing`, `head` and `wheel` which are unconstrained by any other sub-assemblies. By removing these leaves along their `unconstrainedDirection` other parts become unconstrained and can be removed in turn. Following the default setting of defined by Li et al. [18], the `unconstrainedDirection` of each sub-assembly is given as a vector parallel to the coordinate frame axes of the assembly.

Nodes	
Id	Unconstrained Direction
lower_shell	(0,0,0)
upper_shell	(0,1,0)
cylinder	(1,0,0)
head	(1,0,0)
wheel	(0,0,1)
bing	(0,1,0)

Links	
Source Node Id	Target Node Id
lower_shell	upper_shell
lower_shell	cylinder
lower_shell	wheel
upper_shell	bing
upper_shell	cylinder
upper_shell	wheel
cylinder	head

Table 4.3: The full definition of the constraint assembly graph of the engine. Each node defined by an Id and its Unconstrained Direction and each Link linking together two nodes by their Id. Note that the graph structure of constraint assembly graphs is based on *DGML* and similarly to this structure we present a table with nodes and a table with links. In Listing B.7 we show the serialized file of the same constraint assembly of the engine.

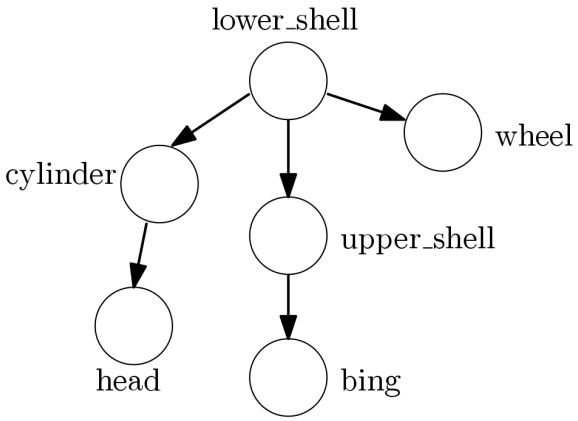


Figure 4.10: An illustration of the constraint assembly graph corresponding to the engine assembly. The dependencies between the individual nodes represented as directed edges between them.

The first step in building the author assembly graph is the filtering of the start-of-movement-candidates with Algorithm 4. For this reason, we first calculated the distances of each sub-assembly between the estimated poses of each frame. See Figure 4.11 for a visualization of the distances for the sub-assemblies `bing` and `cylinder`. We first evaluated the distances for all sub-assemblies to determine the following parameters for the Z-Score algorithm by van Brakel [43]:

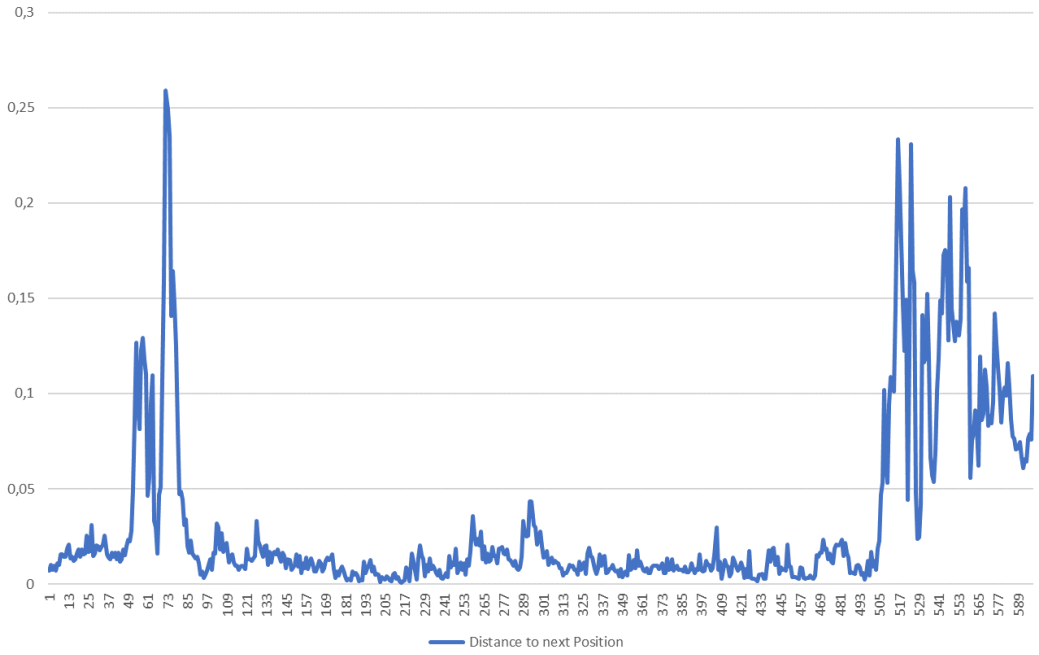
$$\begin{aligned} \text{lag} &= 15 \\ \text{influence} &= 0.0 \\ \text{threshold} &= 7.0 \end{aligned}$$

Based on the results of the 2D-3D pose tracking with PWP3D and after applying Algorithm 4, we get the following start-of-movement-candidates for each sub-assembly. Note that this list is already ordered by the first start-of-movement-candidate frame.

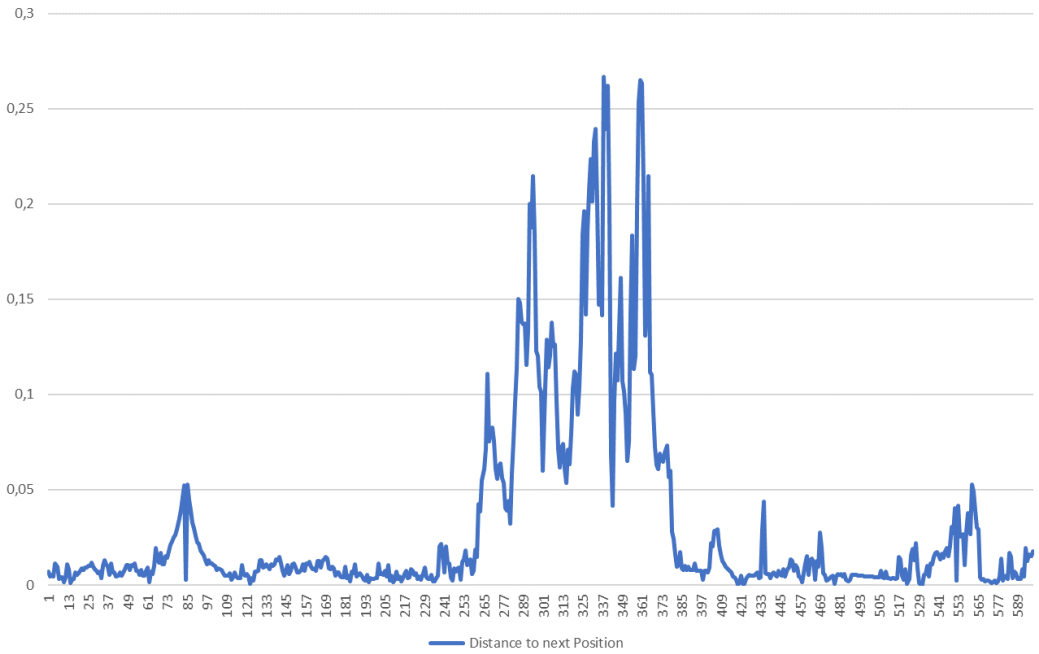
$$\begin{aligned}
\text{bing} &= \{[52, 60], 406\} \\
\text{lower_shell} &= \{160, 225, 265\} \\
\text{cylinder} &= \{238, 239, [403, 408], 434, 435, 506, 509, 517, 518, 521, [524, 527]\} \\
\text{head} &= \{[399, 402], 467, [518, 598]\} \\
\text{wheel} &= \{446, 574\} \\
\text{upper_shell} &= \{[505, 598]\}
\end{aligned}$$

Next, we validate this list against the engine constraint assembly graph illustrated in Figure 4.10. In Table 4.4 we show the steps of Algorithm 5, the list above the starting point for the first iteration. We use the first letter of each sub-assembly to denote them in this table and the following paragraph. The iteration indicates which sub-assembly is validated against the constraint assembly graph. We validate by asserting if the current sub-assembly is unconstrained. If so this sub-assembly can be added as instruction and iteration is incremented. In case the current sub-assembly is constrained, we remove the start-of-movement-candidate of this sub-assembly and sort all candidates by their start-of-movement-candidates again, establishing a new order of sub-assemblies. By ordering, different sub-assemblies will be validated against the constraint assembly graph, since iteration is only incremented if a sub-assembly can be added as instruction. Note that in the table the sorted candidates in the table also contain the current start-of-movement-candidate for this sub-assembly.

The first sub-assembly in the sorted candidates list is `bing`, which is a leaf in the constraint assembly graph, hence it can be added as instruction and iteration is incremented. In the next two steps two of the start-of-movement-candidates of the `lower_shell` sub-assembly are crossed out, since `lower_shell` is the root of the engine constraint assembly graph and is constrained by every other sub-assembly. After sorting, we obtain the following new sorted candidates $\{b = 52, c = 238, l = 265, h = 399, w = 446, u = 505\}$, where `cylinder` is now validated against the constraint assembly graph. Since the `cylinder` is constrained by `head`, two start-of-movement-candidates of `cylinder` are eliminated. The new order of sorted candidates has the last start-of-movement-candidate of `lower_shell` as the next sub-assembly to validate the constraint assembly graph against. This last start-of-movement-candidate is crossed out as well, since `lower_shell` is still constrained. Again the candidates are sorted, resulting in the order $\{b = 52, h = 399, c = 403, w = 446, u = 505, l\}$ giving us `head` as the next sub-assembly for validation against the constraint assembly graph. We can add `head` as instruction, since it is a leaf in the constraint assembly graph. Subsequently all remaining sub-assemblies are added as instructions, starting with `cylinder`, which is not constrained anymore because `head` has previously



(a) Distances `bing`



(b) Distances `cylinder`

Figure 4.11: In Figure 4.11a and Figure 4.11b we visualize the distances between the estimated poses of each frame. By applying Algorithm 4 we aim to detect the peaks in these graphs as peaks indicate a moving sub-assembly.

Iteration	Sorted Candidates	Note	Instruction List
0	$\{b = 52, l = 160, c = 238,$ $h = 399, w = 446, u = 505\}$	`bing` is a leaf, adding it as instruction	$\{\}$
1	$\{b = 52, l = 160, c = 238,$ $h = 399, w = 446, u = 505\}$	`lower_shell` is constrained	$\{b\}$
1	$\{b = 52, l = 225, c = 238,$ $h = 399, w = 446, u = 505\}$	`lower_shell` is constrained	$\{b\}$
1	$\{b = 52, c = 238, l = 265,$ $h = 399, w = 446, u = 505\}$	`cylinder` is constrained	$\{b\}$
1	$\{b = 52, c = 239, l = 265,$ $h = 399, w = 446, u = 505\}$	`cylinder` is constrained	$\{b\}$
1	$\{b = 52, l = 265, h = 399,$ $c = 403, w = 446, u = 505\}$	`lower_shell` is constrained	$\{b\}$
2-5	$\{b = 52, h = 399, c = 403,$ $w = 446, u = 505, l\}$	all sub-assemblies are added as instruction subsequently	$\{b, h, c,$ $w, u, l\}$

Table 4.4: An overview of the validation of the start-of-movement-candidates against the engine constraint assembly graph with Algorithm 5. The sub-assemblies are denoted by their first letter, the Sorted Candidates displaying the current start-of-movement-candidate of this sub-assembly. The Iteration indicating which sub-assembly is validated against the constraint assembly graph. Only if validation succeeds the current sub-assembly is added as instruction and Iteration is incremented. In case it does not succeed the current start-of-movement-candidate is eliminated and we sort the candidates again. By following this, subsequently all sub-assemblies are added as instructions encoding the disassembly sequence depicted in the assembly video.

been added as instruction, over to `wheel`, `upper_shell` and `lower_shell`. The order of removal encoded in the instruction list $\{b, h, c, w, u, l\}$ corresponds to the disassembly sequence depicted in the assembly video. This order of removal is the author assembly graph we are looking for.

In Section 3.5.3 we outlined how to further enhance the author assembly graph by detecting constraining components. See Figure 4.12 for the tools that are currently contained in our pre-defined tool database. The related tool constraints definition is shown in Table 3.1. By applying Algorithm 6 on the engine sub-assemblies, we detect the following constraining components:

bing	= {nut,nut}	head	= { }
lower_shell	= { }	cylinder	= { }
wheel	= { }	upper_shell	= {screw, screw}

Based on these results we now are able to encode additional information in the author assembly graph. Since two `nut` constraining components in `bing` and two `screw` constraining components in `upper_shell` are detected, we now can enhance each related

(a) Tool Screwdriver **(b)** Tool Wrench

Figure 4.12: The tools `Screwdriver` and `Wrench` of our pre-defined tool database. We use these 3D representations to indicate the required tool at each instruction.

#	Sub-Assembly	Unconstrained Direction Vector	Tool Required + Position
1	`bing`	$(0, 1, 0)$	`Wrench` $(3.77, 2.13, 1.39)$, `Wrench` $(3.91, 2.13, -0.62)$
2	`head`	$(1, 0, 0)$	-
3	`cylinder`	$(1, 0, 0)$	-
4	`wheel`	$(0, 0, 1)$	-
5	`upper_shell`	$(0, 1, 0)$	`Screwdriver` $(-9.63, 0.1, 0.13)$, `Screwdriver` $(-9.63, 0.1, -2.68)$
6	`lower_shell`	$(0, 0, 0)$	-

Table 4.5: The author assembly graph of the engine corresponding to our assembly video. Each row corresponds to one instruction of the author assembly graph. The row entries are sorted by order of removal of the sub-assemblies. Every instruction not only contains the unconstrained direction of the related sub-assembly but also information about required tools and the positions of the corresponding constraining components.

instruction. Specifically, according to our tool constraints definition in Table 3.1, the tool `Wrench` is required to remove two constraining components, before one is able to remove `bing` along its unconstrained direction and proceed to the next instruction. The same applies to `upper_shell` where the tool `Screwdriver` is required.

We conclude the presentation of all these results of our system by showing the author assembly graph of the engine corresponding to our assembly video in Table 4.5. The sub-assemblies are sorted by their order of removal. Note the unconstrained direction of each sub-assembly derived from the related constraint assembly graph and the encoded information about required tools and the positions of the corresponding constraining components. In Listing 4.1 we present a serialization of the same author assembly graph.

We talked about the motivation of this thesis in Chapter 1 and want to reiterate further that Listing 4.1 can be used as a basis to create tutorials, as the author assembly graph encodes step by step descriptions as instructions.

```
<ArrayOfInstruction>
  <Instruction id="bing">
    <unconstrainedDirection>
      <x>0</x><y>1</y><z>0</z>
    </unconstrainedDirection>
    <ToolConstraints>
      <ToolConstraint>
        <Tool>Wrench</Tool>
        <ConstraintPosition>
          <x>3.77127647</x><y>2.12999988</y<z>1.39133453</z>
        </ConstraintPosition>
      </ToolConstraint>
      <ToolConstraint>
        <Tool>Wrench</Tool>
        <ConstraintPosition>
          <x>3.91409445</x><y>2.12999988</y><z>-0.6169067</z>
        </ConstraintPosition>
      </ToolConstraint>
    </ToolConstraints>
  </Instruction>
  <Instruction id="head">
    <unconstrainedDirection>
      <x>1</x><y>0</y><z>0</z>
    </unconstrainedDirection>
    <ToolConstraints />
  </Instruction>
  <Instruction id="cylinder">
    <unconstrainedDirection>
      <x>1</x><y>0</y><z>0</z>
    </unconstrainedDirection>
    <ToolConstraints />
  </Instruction>
  <Instruction id="wheel">
    <unconstrainedDirection>
      <x>0</x><y>0</y><z>1</z>
    </unconstrainedDirection>
```

```
    <ToolConstraints />
  </Instruction>
  <Instruction id="upper_shell">
    <unconstrainedDirection>
      <x>0</x><y>1</y><z>0</z>
    </unconstrainedDirection>
    <ToolConstraints>
      <ToolConstraint>
        <Tool>Screwdriver</Tool>
        <ConstraintPosition>
          <x>-9.625</x><y>0.100000009</y><z>0.125</z>
        </ConstraintPosition>
      </ToolConstraint>
      <ToolConstraint>
        <Tool>Screwdriver</Tool>
        <ConstraintPosition>
          <x>-9.625</x><y>0.100000009</y><z>-2.6825</z>
        </ConstraintPosition>
      </ToolConstraint>
    </ToolConstraints>
  </Instruction>
  <Instruction id="lower_shell">
    <unconstrainedDirection>
      <x>0</x><y>0</y><z>0</z>
    </unconstrainedDirection>
    <ToolConstraints />
  </Instruction>
</ArrayOfInstruction>
```

Listing 4.1: The serialized author assembly graph of the engine corresponding to our assembly video. The encoded instructions are sorted by their order of removal. Each instruction also encodes the unconstrained direction of the related sub-assembly and information about required tools and the positions of the corresponding constraining components

4.2.4 2D Print Tutorial

In the previous section we detailed the author assembly graph of our engine assembly – for a representation see Table 4.5, for a serialization see Listing 4.1 – and how it was build. This section presents the first example application which makes use of such an author assembly graph to create a tutorial. Specifically, we present an application which uses the basic tutorial elements described in Section 4.1 to build a 2D Print Tutorial based on the author assembly graph of our engine assembly. Note that this example application can automatically build a 2D Print Tutorial given an author assembly graph based on any assembly.

For the 2D Print Tutorial, we created a setup with three cameras that differentiate themselves through the layers they render.

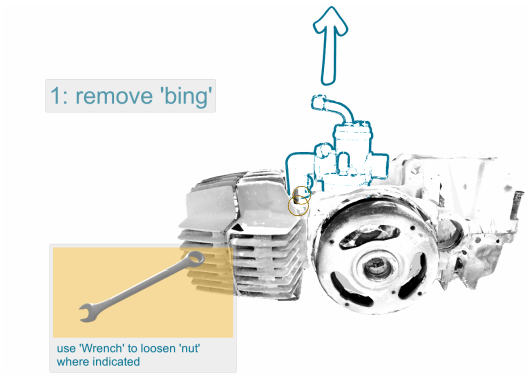
- **Main Camera.** Renders everything, which also includes all the elements of the user interface. This camera also applies the black-and-white color grading.
- **Instruction Camera.** This camera is used to render all the highlighted objects. That is the case for the relevant sub-assembly of the current instruction and its related glyph.
- **Tool Indicator Camera.** We use this camera to render the billboard tool indicators on top of everything else.

We build the 2D Print Tutorial by iterating through every instruction of the author assembly graph. During each instruction, we set up the correct position and direction of the glyph, place tool indicators if required, outline the current sub-assembly and finally set the correct layers s.t. each camera renders the right objects. There are five steps necessary to get to the `lower_shell` object, each described by one instruction. See Figure 4.13 which illustrates these instructions and the final 2D Print Tutorial for the engine assembly. We also create a PDF version of this 2D Print Tutorial where each page features one of the images shown in Figure 4.13.

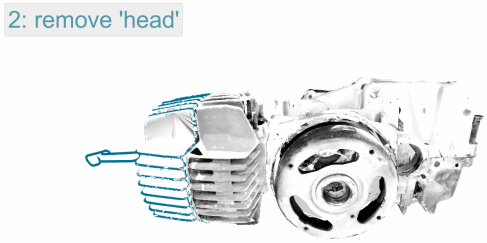
4.2.5 Augmented Reality Tutorial

The second example application we present is an Augmented Reality Tutorial running on Android smartphones. The basic tutorial elements are the same we described in Section 4.1. Similar to the 2D Print Tutorial, we base this Augmented Reality Tutorial on the same author assembly graph of the engine assembly. We present this author assembly graph in Table 4.5 and a serialization in Listing 4.1. Note that this second example application can create an Augmented Reality Tutorial given an author assembly graph based on any assembly.

We use Vuforia™ Model Targets [44] as the underlying framework to integrate augmented reality capabilities into our application. Note that Vuforia™ is built into Unity3D making it comfortable to use. When working with Vuforia™ Model Targets, it is necessary



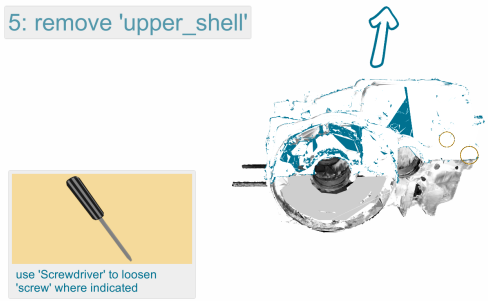
(a) Step 1



(b) Step 2

(c) Step 3

(d) Step 4



(e) Step 5

(f) Step 6

Figure 4.13: The instructions of the 2D Print Tutorial based on the author assembly graph of the engine assembly. The actual print version features each step on its own page.

to define *views* of the target 3D object in an external application provided by Vuforia™ in a pre-processing step. Those *views* are defined as specific snapshots of the target 3D object from one or more different perspectives by the user. For every *view* Vuforia™ detects features and saves this feature information into a database, which can be exported from this external application and imported into a Unity3D project. Now when setting up the actual augmented reality application and running it, the real object is detected when viewing it from a perspective similar to a pre-processed *view*. Note that this additional pre-processing cannot be automatized. Hence, when using another author assembly graph based on a different assembly, a new Model Target database has to be created s.t. this new assembly can be tracked by Vuforia™ in our example application.

In contrast to the setup of the 2D Print Tutorial, we only use one camera, which is for augmented reality purposes. The user interface, glyphs, and tool indicators are rendered on top of the incoming video stream after registering the Model Target. After building and installing the application on a smartphone, we have the following workflow.

When first starting the Augmented Reality Tutorial app on a smartphone, the Model Target has to be registered. That is done by aligning the outline of the pre-processed *view* with the real object in the camera stream. After this alignment and the subsequent registration of the real object, the user taps the *Start* button to start displaying augmented instructions. Note that these instructions are based on the author assembly graph of the engine assembly. Now for every instruction we display, we set up the correct position and direction of the glyph and place tool indicators if required. Figure 4.14 shows a filmstrip of a screen capture of a smartphone running this Augmented Reality Tutorial app. After applying tools if necessary and then removing a sub-assembly, a tap on the screen displays the next instruction. This way we iterate through every instruction of the author assembly graph of the engine assembly, finally getting to the `lower_shell` sub-assembly after five steps.

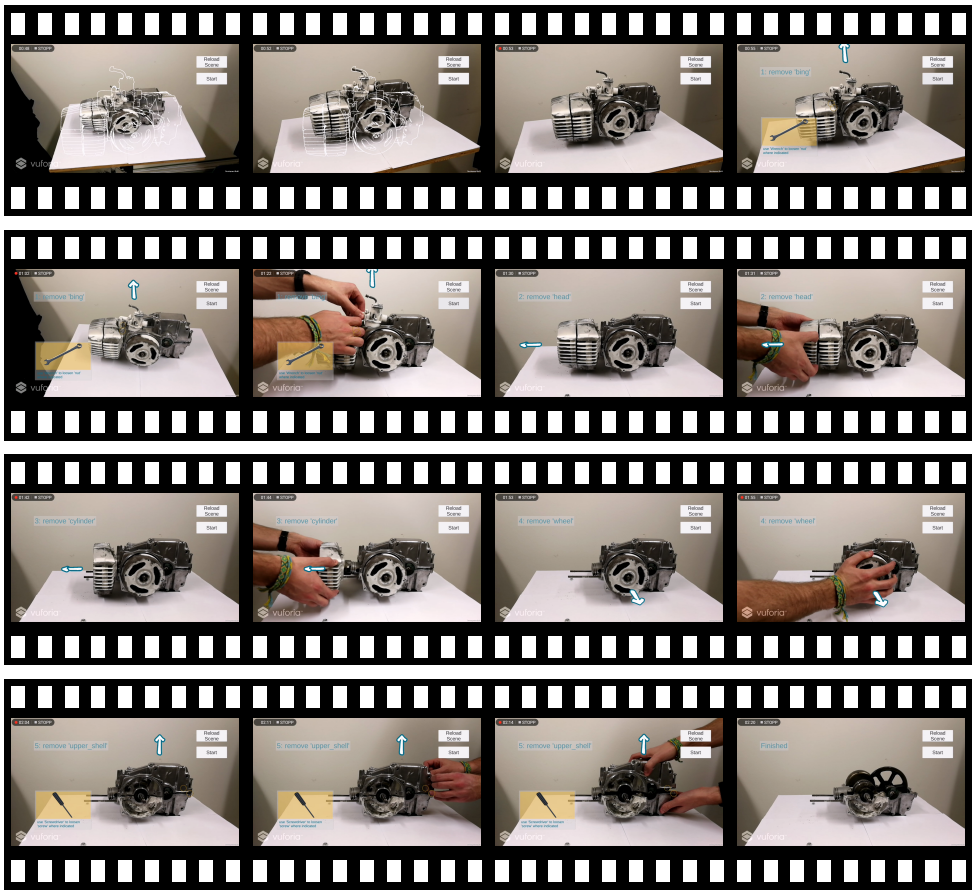


Figure 4.14: This filmstrip shows a screen capture of a smartphone running the Augmented Reality Tutorial. Before starting the actual tutorial, first the pre-processed Vuforia™ *view* has to be aligned with the real object s.t. it is registered. Note that this Vuforia™ *view* is visualized by white edges based on the virtual 3D object. After registration the user taps the *Start* button to launch the actual tutorial to start displaying augmented instructions. Subsequently for each instruction we setup the correct position and direction of the glyph and place tool indicators if required. The next instruction is displayed as soon as the user taps the screen. Iterating through every instruction of the author assembly graph of the engine assembly, we reach the `lower_shell` after five steps.

5 Evaluation

We started with an idea of a system that creates XML based construction manuals and detailed its specifics in Chapter 3. In Chapter 4 we showed how we practically built such an XML based construction manual and illustrated how the same XML based construction manual can be used to target different media types of tutorials. In the following chapter, we further evaluate specific parts of this system and discuss possible limitations. For an overview of the used terminology see Section 3.1.

5.1 Quality of Input

The system we propose requires two inputs to be able to create an author assembly graph. Namely, that is a 3D representation of the real object and a video depicting the (dis)assembly of this object by an expert.

We argue that the quality of these inputs can have a significant impact on the quality of the 2D object detection with You Only Look Once (*YOLO*) and the 2D-3D pose estimation with PWP3D. Hence following is an overview of our key findings regarding the impact on tracking and pose estimation with these frameworks based on our experience using them.

Light and exposure. Note that both *YOLO* and PWP3D operate on the pixels of images. Therefore it is crucial that the captured assembly video is neither over- nor underexposed. In our experiments, both frameworks did not produce as good results when processing assembly videos with too much or too little lighting. We argue that this is in line for systems that operate on pixels and rely on color information for 2D object detection and 2D-3D pose estimation.

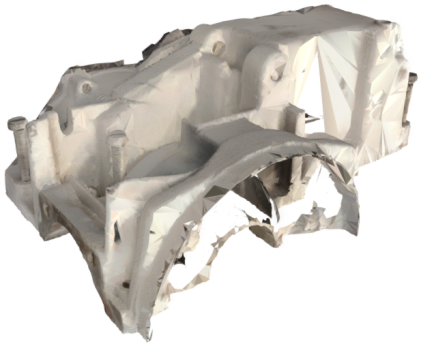
Specular reflections. We discussed over- and underexposure and want to take special note of specular reflections that can occur on the assembly when capturing an assembly video. Specular reflections occur when an object reflects rays of light. The intensity of these reflections depends on the intensity of the incoming light and the complexion of the surface this incoming light hits. Be aware of these specular reflections when capturing

an assembly video as this effect has an impact on the quality of the 2D object detection and 2D-3D pose estimation of the assembly for the same reason as light and exposure in general.

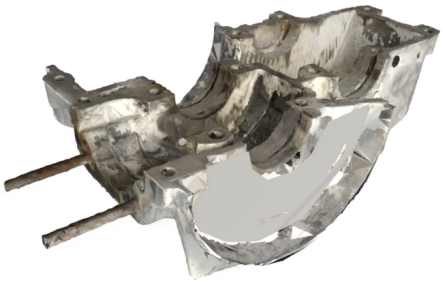
Matching virtual representation of the real object. In Section 4.2 we mentioned that the engine assembly we presented was created with photogrammetry. The resulting mesh artifacts impact the quality of the synthetic dataset we use for training *YOLO*. Therefore it is preferable to use the original *CAD* data, if available, or create a virtual representation of a real object from the ground up with a 3D modeling software. See Figure 5.1 for images that were sampled by our application, which we detailed in Section 3.3.2, that creates such synthetic datasets. In the following section, we further argue the impact of mesh artifacts on 2D object detection with *YOLO*.

(a) Synthetic Dataset Sample 1

(b) Synthetic Dataset Sample 2



(c) Photogrammetry Mesh Artifacts



(d) Photogrammetry Mesh Artifacts Repaired

Figure 5.1: The 3D objects were build using photogrammetry, which is prone to create mesh artifacts. In Figure 5.1a and Figure 5.1b we show sampled images from our synthetic dataset with respect to mesh artifacts. The same 3D objects are shown in Figure 5.1c and Figure 5.1d where it is easier to distinguish these mesh artifacts. In Figure 5.1d some of these artifacts were covered up by filling holes.

5.2 Detection and Tracking Quality

We described the quality standard for the inputs to our system in the previous section. This section focuses on the impact of subpar input on You Only Look Once (*YOLO*) and PWP3D. Both frameworks rely on different characteristics concerning their input for satisfactory results.

Robust 2D object detection through training dataset. The previous section described the mesh artifacts of our engine assembly. Although using 3D objects with mesh artifacts to create synthetic training datasets was not optimal, our method of filtering with Algorithm 3 still proved to be successful in finding good frames for initializing each sub-assembly for PWP3D. It also should be noted that the other 3D representations of the sub-assemblies contain negligible mesh artifacts in contrast to the two 3D sub-assemblies illustrated in Figure 5.1. Following is an overview of all the detections per object class over all the 600 frames of our assembly video, corresponding to the results we presented in Section 4.2.1. Note the possibility of multiple detections of the same object during one frame.

$$\begin{aligned} \text{bing} &= 131 \\ \text{head} &= 18 \\ \text{cylinder} &= 213 \\ \text{wheel} &= 357 \\ \text{lower\_shell} &= 1592 \\ \text{upper\_shell} &= 1897 \end{aligned}$$

It is very likely that the high number of detections of the objects `upper_shell` and `lower_shell` are caused by the mesh artifacts of these 3D objects resulting in *YOLO* training rather generic features for these two object classes. Note the order of removal for this disassembly sequence depicted by the assembly video: `bing`, `head`, `cylinder`, `wheel`, `upper_shell` and `lower_shell`. This aligns with the number of detections for the objects `bing`, `cylinder` and `wheel`. The earlier an object is removed, moved outside the frame of the video, the lower the number of its detections. Unfortunately `head` did not result in many detections. We reason that this is due to the similarity between `head` and `cylinder`, which is illustrated by Figure 5.2, leading *YOLO* to mistake one object for the other. Nonetheless Algorithm 3 successfully found two satisfactory initial frames for both sub-assemblies.

Relation between synthetic dataset and assembly video. It is essential to note the relation between the images of a synthetic dataset and the assembly video. *YOLO* scales every input image to the resolution defined by its configuration. For *YOLO* to yield satisfactory results, the aspect ratio of the input video has to be equal to the aspect ratio of the input images *YOLO* uses for training. Although per default we set the input images

(a) Sub-Assembly Head **(b)** Sub-Assembly Cylinder

Figure 5.2: The sub-assemblies `head` and `cylinder` share a similar structure. We argue that this can lead *YOLO* to mistake one object for the other.

to a different width and height as the assembly video, both have the same aspect ratio. That is in line with the architecture of *YOLO* and the findings of [15, chapter *Discussion*, pp 48].

Robust 2D-3D pose estimation. In Section 2.2.1 we outlined the robustness of PWP3D to its initialization as it can recover from initial misalignments. [29] also argue that PWP3D is robust to motion blur, noise, occlusion and approximated models. The last collaborates with our engine assembly, which is an approximated model as it contains mesh artifacts due to photogrammetry. Our experiments confirm this by resulting in accurate pose estimations over the frames of the assembly video for each sub-assembly.

5.3 Complexity of Constraint Assembly Graphs

The order of removal encoded by an author assembly graph is calculated by validating start-of-movement-candidates against a related constraint assembly graph. We outlined our methodology of this validation as part of Section 3.5 and designed Algorithm 5 for this purpose. Note though that the validation against a constraint assembly graph relies on Algorithm 4 to find suitable start-of-movement-candidates.

On the one hand we argue that the default values of Algorithm 4 presented in Section 4.2.3 can be used for any assembly, while still producing suitable start-of-movement-candidates. On the other hand, it is likely that further research on Algorithm 5 is necessary to extend its capabilities to consider the bigger picture of a more complex constraint assembly graph. In Section 6.1 we further address this in more detail.

Despite this Algorithm 5 performed well for our task of calculating the order of removal for the engine assembly as the primary goal of this thesis is to show the feasibility of

building an author assembly graph solely based on a 3D object and a video as input to our proposed system.

5.4 Showcasing the Tool Database

In addition to the order of removal of the sub-assemblies, author assembly graphs produced by our proposed system also encode information about constraining components. We first discussed constraining components, tool databases and tool constraint definitions in Section 3.5.3 and presented practical results in Section 4.2.3.

While the extent of this part of our system is minor compared to others, it has significant impact on the outcome. Especially when looking at the example applications, we presented in Chapter 4, it is beneficial being able to indicate constraining components to a user. The calculation of the position of these constraining components is straightforward as we use the position of each constraining component in each sub-assembly to place the tool indicators illustrated in the example applications. The actual challenge is the detection of such constraining components.

That is why we believe that further research in this field to have great impact on other 3D tutorial related systems or future versions of our proposed system. We further discuss possible scenarios in Section 6.1.

5.5 Requirements when building upon an Author Assembly Graph

We have described in great detail how we arrive at an author assembly graph in previous chapters. In this section, we discuss the modularity of our approach and requirements when building a tutorial based on an author assembly graph.

Assembly. So far we have been detailing the author assembly graph. Note though that when building an illustrative tutorial, also the related 3D assembly to this author assembly graph is required.

Software framework. The example applications we presented in Chapter 4 were created using Unity3D. Note that also any other similar software can be used to build a tutorial based on an author assembly graph. From this software, it is only required that it can read in the author assembly graph and handle 3D objects, namely the related assembly.

Relation between instruction and sub-assembly. One part of the encoded information in the author assembly graph is the `instruction-id`. Using this `instruction-id` we can relate an instruction to its sub-assembly. For this reason, we encode the name of the sub-assembly corresponding to the instruction as `instruction-id`. When creating a tutorial based on an author assembly graph a data structure containing this relation is required.

Generic descriptions of tools. As part of the example applications we show 3D representations of necessary tools corresponding to the constraining components of the current instruction. Although we encode the required tools in the author assembly graph, the same 3D representations of these tools as shown in our examples do not need to be provided. We argue that depending on the domain of the assembly the look and handling of these tools can vary considerably. That is why we define a generic description, as is encoded in the author assembly graph, for each tool to be sufficient to identify and illustrate it in tutorials.

Benefits of this modular approach. In Chapter 3 we detailed the different parts of our proposed system – see Figure 3.1 for a visualization. As soon as an author assembly graph is created, together with the related assembly, it can be used as a basis to create any media type of tutorial. We believe this to be an aid to tutorial creators because the requirements to build a tutorial of an author assembly graph are kept low, while the author assembly graph itself still encodes all relevant information about a particular (dis)assembly sequence. The many different practical applications based on such author assembly graph are reserved for future work.

5.6 Extent of Automatization

We have outlined the goals of this thesis in Chapter 1 and proposed a system to achieve these goals in Chapter 3. In both chapters, we highlighted the set goal of an automatized system. While we may not have reached this goal at this time, we achieved promising partial results. Following an overview of the extent of automatization of each part of the system.

You Only Look Once. In Section 4.2.1 we gave an overview of the necessary steps that result in initial frames for each sub-assembly for the initialization of PWP3D. By re-iterating these steps we evaluate the extent of automatization of this part of the system. The creation of the synthetic dataset is fully automatized, meaning when starting the corresponding application a new synthetic dataset for the 3D object you fed into it is created. When it comes to You Only Look Once ($YOLO$) itself, we created a script which re-calculates the anchors, invokes training with $YOLO$, the validation of the resulting weights, starts detection with the best weight on the assembly video and filters initial frames for the initialization of PWP3D. One downside to this is that we always chose the best weight for detection based on the highest validated mean Average Precision (mAP) – see Section 4.2.1 for more details. We are aware that this approach likely picks an overfit weight as our convergence criteria for stopping training is the number of trained iterations, and later weights will have a higher mAP than earlier weights. That is why we chose the best weight by manually looking at charts like Figure 4.6 and tables like Table 4.1 and manually invoke detection and the subsequent filtering of initial frames at this time.

PWP3D. We outlined the requirements and involved steps resulting in a 6 DoF pose for each sub-assembly for every frame of the assembly video by using PWP3D in

Section 3.4 and presented its results in Section 4.2.2. At the time of writing, we manually set the initial 6 *DoF* pose in the initial frame and invoke 2D-3D pose tracking on the assembly video for every sub-assembly. Note that the process itself is automatized in the PWP3D framework, meaning you start PWP3D once for each sub-assembly. This manual step, unfortunately, restricts the use of our system. We hope to amend this in future versions and outline possible approaches in the following chapter.

Building the Author Assembly Graph. Given a constraint assembly graph and the results of 2D-3D pose tracking over the assembly video, we can automatically build an author assembly graph at this time. Note possible limitations due to more complex constraints assembly graphs which we discussed in Section 5.3.

The same example tutorials with new underlying content. The example applications presented in Chapter 4 further illustrate the benefit of the modular approach of our proposed system, which we described in Section 5.5. Taking the 2D Print Tutorial as a basis, the underlying author assembly graph and its related assembly can be seamlessly exchanged for an author assembly graph based on a different assembly or assembly video. Without any adjustments necessary a new 2D Print Tutorial based on this new author assembly graph and new assembly with the same tutorial characteristics can be created. We can achieve the same with the Augmented Reality Tutorial. However, one additional step is required. In Section 4.2.5 we detailed the Vuforia™ Model Target tracking, which we use to track and augment assemblies in the real world. Since it is not possible to automatize the generation of the database that Vuforia™ uses to model-track objects, this additional pre-processing step has to be handled manually. After importing this pre-processed database and adding the new author assembly graph with its new assembly, a new smartphone application featuring this new content can be built.

6 Conclusion

In this thesis, we set out to build a system which is capable of automatically processing videos and related 3D objects to create XML based construction manuals. We further investigated practical applications of XML based construction manuals, i.e. build tutorials using such an XML based construction manual and its related 3D object. Our work is evidence of the feasibility of such a system, and our results illustrate the various possibilities for its applications.

In contrast to current systems for the automatized creation of tutorials which focus on retargeting one type of media to another, we developed a conceptional workflow that interacts through interfaces between its various parts. As a result, we are able to retarget a video tutorial to tutorials of any type of media, if also the virtual 3D representation of the real world object depicted in this video is given.

While we show the feasibility of such a system, we are aware that this system we proposed is not yet fully automatic. Specifically parts of the initialization of the 2D-3D pose estimation offer an area for future research. During our work on practical applications of XML based construction manuals, we found the area of augmented reality tutorials to be a promising field of research in and of itself.

We believe our system and its conceptual idea to be a promising field for future research and want to conclude our thesis with final remarks on possible improvements and new ideas for our system.

6.1 Future Work

6.1.1 Initialization of 2D-3D Pose Tracking

With further research in this area, we hope to automatize the whole process of 2D-3D pose tracking with PWP3D. We believe that an approach similar to the external application Vuforia™ uses to set up their *views* for model tracking, which we detailed in Section 4.2.5, to be worthwhile looking into. That is, creating snapshots of a 3D object from different perspectives and matching this to a pose. Following this approach, it is crucial to note the influence of lightning on the detection quality. Therefore we believe that a combination of this Vuforia™ application and a system like that of Mandl et al. [19], which learns lightprobes for mixed reality illumination, to be a promising starting point to estimate a 6 *DoF* pose.

6.1.2 Extending the Concept of Tool Databases

We showcased the general concept of tool databases in this thesis. Further work could treat this part of our system, as a system of its own. A more comprehensive framework that, e.g., can be used as a plugin in tutorial related software looks to be an aid to such systems. In this context, we refer to more advanced algorithms for matching constraining components and a survey of *CAD* data to be able to pre-define the most common terms for constraining components in a generic way with, e.g., regular expressions.

6.1.3 Improving the Validation against Constraint Assembly Graphs

In Section 5.3 we evaluated Algorithm 5 in the context of this thesis. For future versions, this algorithm might need to consider the bigger picture of more complex constraint assembly graphs. We propose to extend this algorithm by looking at multiple start-of-movement-candidates at a time and also validate their related 3D paths. In future versions, a matching of these 3D paths against the unconstrained directions to establish an order of removal of sub-assemblies might be worthwhile.

6.1.4 Custom Glyph Paths

We adopt the unconstrained directions encoded in constraint assembly graphs in resulting author assembly graphs. Note that these unconstrained directions are vectors parallel to the coordinate frame axes of the assembly and as such not necessarily are the actual directions of removal depicted in the assembly video. Therefore we suggest extracting these actual directions of removal from the results of the 2D-3D pose tracking in future versions.

6.1.5 Applications of XML Based Construction Manuals

Future work likely will also explore new fields of application or take a deeper dive on the type of tutorials we presented as examples. Surveys about the current state of 2D Print Tutorials and Augmented Reality Tutorials might be a place to start when further pursuing this topic. Using these surveys as a basis our example applications can surely be enhanced.

Unity3D Asset Bundles. Assuming that future tutorials are also build using Unity3D, it is worthwhile looking at the possibilities of Unity3D itself, when trying to enhance our existing example applications. As such, e.g., Unity3D Asset Bundles [41] are a compelling concept which allows the import of assets into an existing application – no rebuilding of the application as a whole required. Pursuing this, future versions could deal with new author assembly graphs and their related assemblies in a more dynamic way resulting in a more automatized workflow.

Augmented Reality. We mentioned that tutorials enhanced with the capabilities of augmented reality to be a promising area of research in and of itself. The main benefits of such interactive Augmented Reality Tutorials being that, e.g., a user can view the real object from all directions, while the instructions and additional information are still anchored on the real object. As such, existing visualization techniques as, e.g., presented by Kalkofen et al. [16] would level up our existing example of an Augmented Reality Tutorial.

Tutorial Tool Case. The present findings regarding the creation of tutorials for different types of media based on author assembly graphs suggests that, e.g., dedicated applications focused on the aided creation of tutorials are also an area of future research. Another way of referring to this would probably be calling it *power point for tutorial creation based on a pre-defined instruction order*. That might lead to the development of a whole tool case to be applicable at each instruction of an author assembly graph.

A

List of Acronyms

CAD	Computer Aided Design
CNN	Convolutional Neural Network
DGML	Directed Graph Markup Language
DoF	Degrees of Freedom
FPS	frames per second
GPU	Graphics Processing Unit
IoU	Intersection-over-Union
mAP	mean Average Precision
VOC	Visual Object Classes
YOLO	You Only Look Once

B

Supplemental Material

```
   ...
 7 subdivisions=8
 8 height=416
 9 width=416
   ...
603 filters=255
   ...
609 anchors=10,13  ...  373,326
610 classes=80
   ...
689 filters=255
   ...
695 anchors=10,13  ...  373,326
696 classes=80
   ...
776 filters=255
   ...
782 anchors=10,13  ...  373,326
783 classes=80
   ...
```

Listing B.1: This shows the relevant lines of the original configuration of *YOLO*.

```
   ...
 7 subdivisions=32
 8 width=608
 9 height=608
   ...
603 filters=...
   ...
609 anchors=...
610 classes=...
   ...
689 filters=...
   ...
695 anchors=...
696 classes=...
   ...
776 filters=...
   ...
782 anchors=...
783 classes=...
   ...
```

Listing B.2: We present the adjusted lines of the configuration of *YOLO*.

```
<?xml version="1.0" encoding="utf-8"?>
<DirectedGraph Title="DrivingTest" Background="Blue" xmlns="http
    ://schemas.microsoft.com/vs/2009/dgml">
    <Nodes>
        ...
    </Nodes>
    <Links>
        ...
    </Links>
    <Categories>
        ...
    </Categories>
    <Properties>
        ...
    </Properties>
</DirectedGraph>
```

Listing B.3: A serialized example of the Directed Graph Markup Language (*DGML*) by Microsoft [21].

```
<ArrayOfConstraintDefinition >
  <ConstraintDefinition>
    <Pattern>
      <string>nut</string>
    </Pattern>
    <tool>Wrench</tool>
  </ConstraintDefinition>
  <ConstraintDefinition>
    <Pattern>
      <string>screw</string>
    </Pattern>
    <tool>Screwdriver</tool>
  </ConstraintDefinition>
</ArrayOfConstraintDefinition>
```

Listing B.4: We use a tool constraints definition to map tools of our tool database to patterns. This allows us to infer a related tool, if a pattern is matched in a sub-assembly. Here we show a serialization of such a tool constraints definition.

```
...
upper_shell: 99.05%; f 25; 493x 325y; w: 908, h: 277
upper_shell: 39.54%; f 25; 537x 436y; w: 1023, h: 364
lower_shell: 86.92%; f 25; 531x 545y; w: 761, h: 278
upper_shell: 95.63%; f 25; 845x 323y; w: 749, h: 411
cylinder: 92.49%; f 25; 342x 535y; w: 630, h: 368
wheel: 79.53%; f 25; 766x 546y; w: 500, h: 333
bing: 74.58%; f 25; 474x 242y; w: 496, h: 342
lower_shell: 32.89%; f 25; 1005x 255y; w: 476, h: 348
...
```

Listing B.5: We show an excerpt of the results of the predictions of *YOLO* over the assembly video. Specifically, we show the predictions for the 25th frame of assembly video. Note that `upper_shell` and `lower_shell` were detected multiple times in this single frame, while `cylinder`, `wheel` and `bing` were detected only once. We evaluate all the predictions of all frames by applying Algorithm 3.

```
...
35: (0.808423, 0.045217, 21.299782, -132.738297, 6.859068,
    -24.283009)
36: (0.808423, 0.045217, 21.299782, -132.738297, 6.859068,
    -24.283009)
37: (0.808423, 0.045217, 21.299782, -132.738297, 6.859068,
    -24.283009)
...
236: (0.842073, -0.605555, 19.083961, -119.048218, 19.910362,
    -27.806824)
237: (0.842073, -0.605555, 19.083961, -119.048218, 19.910362,
    -27.806824)
238: (0.842073, -0.605555, 19.083961, -119.048218, 19.910362,
    -27.806824)
...
492: (0.418461, -0.192556, 23.527088, -128.202835, 7.023273,
    -29.541023)
493: (0.418461, -0.192556, 23.527088, -128.202835, 7.023273,
    -29.541023)
494: (0.418461, -0.192556, 23.527088, -128.202835, 7.023273,
    -29.541023)
...
```

Listing B.6: We present an excerpt of the results of 2D-3D pose tracking the wheel subassembly with PWP3D. Each result is given as a tuple of `(frame: position.x, position.y, position.z, rotation.x, rotation.y, rotation.z)`.

```
<DirectedGraph>
  <links>
    <Link sourceNodeId="lower_shell" targetNodeId="upper_shell"
       />
    <Link sourceNodeId="lower_shell" targetNodeId="cylinder" />
    <Link sourceNodeId="lower_shell" targetNodeId="wheel" />
    <Link sourceNodeId="upper_shell" targetNodeId="bing" />
    <Link sourceNodeId="upper_shell" targetNodeId="cylinder" />
    <Link sourceNodeId="upper_shell" targetNodeId="wheel" />
    <Link sourceNodeId="cylinder" targetNodeId="head" />
  </links>
  <nodes>
```

```
    <Node id="lower_shell">
      <unconstrainedDirection>
        <x>0</x><y>0</y><z>0</z>
      </unconstrainedDirection>
    </Node>
    <Node id="upper_shell">
      <unconstrainedDirection>
        <x>0</x><y>1</y><z>0</z>
      </unconstrainedDirection>
    </Node>
    <Node id="cylinder">
      <unconstrainedDirection>
        <x>1</x><y>0</y><z>0</z>
      </unconstrainedDirection>
    </Node>
    <Node id="head">
      <unconstrainedDirection>
        <x>1</x><y>0</y><z>0</z>
      </unconstrainedDirection>
    </Node>
    <Node id="wheel">
      <unconstrainedDirection>
        <x>0</x><y>0</y><z>1</z>
      </unconstrainedDirection>
    </Node>
    <Node id="bing">
      <unconstrainedDirection>
        <x>0</x><y>1</y><z>0</z>
      </unconstrainedDirection>
    </Node>
  </nodes>
</DirectedGraph>
```

Listing B.7: The serialized constraint assembly graph of the engine assembly. Note the *DGML* based graph structure featuring nodes and links. Each node also encoding an unconstrained direction of removal of the related sub-assembly.

Bibliography

[1] Aldoma, A., Vincze, M., Blodow, N., Gossow, D., Gedikli, S., Rusu, R. B., and Bradski, G. R. (2011). Cad-model recognition and 6dof pose estimation using 3d cues. In *IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain, November 6-13, 2011*, pages 585–592. IEEE Computer Society. (page 7)

[2] AlexeyAB (2019a). Yolo_mark. `https://github.com/AlexeyAB/Yolo_mark`. Accessed: 2019-03-25. (page 35)

[3] AlexeyAB (2019b). You only look once for windows. `https://github.com/AlexeyAB/darknet`. Accessed: 2019-03-12. (page 13, 14, 18)

[4] Blender Foundation (2019). Blender. `https://www.blender.org/`. Accessed: 2019-03-01. (page 4)

[5] Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. (page 20)

[6] Brox, T., Rosenhahn, B., and Weickert, J. (2005). Three-dimensional shape knowledge for joint image segmentation and pose estimation. In Kropatsch, W. G., Sablatnig, R., and Hanbury, A., editors, *Pattern Recognition, 27th DAGM Symposium, Vienna, Austria, August 31 - September 2, 2005, Proceedings*, volume 3663 of *Lecture Notes in Computer Science*, pages 109–116. Springer. (page 7)

[7] Chi, P., Ahn, S., Ren, A., Dontcheva, M., Li, W., and Hartmann, B. (2012). Mixt: automatic generation of step-by-step mixed media tutorials. In *The 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12, Cambridge, MA, USA, October 7-10, 2012*, pages 93–102. (page 1)

[8] Dambreville, S., Sandhu, R., Yezzi, A. J., and Tannenbaum, A. R. (2008). Robust 3d pose estimation and efficient 2d region-based segmentation from a 3d shape prior. In Forsyth, D. A., Torr, P. H. S., and Zisserman, A., editors, *Computer Vision - ECCV 2008, 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part II*, volume 5303 of *Lecture Notes in Computer Science*, pages 169–182. Springer. (page 7)

[9] Everingham, M., Eslami, S. M. A., Gool, L. J. V., Williams, C. K. I., Winn, J. M., and Zisserman, A. (2015). The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136. (page 5)

[10] Everingham, M., Gool, L. J. V., Williams, C. K. I., Winn, J. M., and Zisserman, A. (2010). The pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338. (page 4, 18, 33)

[11] Google (2019). Google image labeler. `https://crowdsource.google.com/`. Accessed: 2019-03-01. (page 4)

[12] Griffin, G., Holub, A., and Perona, P. (2007). Caltech-256 object category dataset. Technical Report 7694, California Institute of Technology. (page 4)

[13] Heikkilä, J. and Silvén, O. (1997). A four-step camera calibration procedure with implicit image correction. In *1997 Conference on Computer Vision and Pattern Recognition (CVPR '97), June 17-19, 1997, San Juan, Puerto Rico*, pages 1106–1112. IEEE Computer Society. (page 13)

[14] Hui, J. (2019). mAP (mean average precision) for object detection. `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`. Accessed: 2019-03-14. (page 19)

[15] Jensen, S. and Selvik, A. L. (2016). Using 3d graphics to train object detection systems. Master's thesis, Norwegian University of Science and Technology. (page 4, 14, 56)

[16] Kalkofen, D., Sandor, C., White, S., and Schmalstieg, D. (2011). Visualization techniques for augmented reality. In Furht, B., editor, *Handbook of Augmented Reality*, pages 65–98. Springer. (page 63)

[17] Kalkofen, D., Tatzgern, M., and Schmalstieg, D. (2009). Explosion diagrams in augmented reality. In *IEEE Virtual Reality Conference 2009 (VR 2009), 14-18 March 2009, Lafayette, Louisiana, USA, Proceedings*, pages 71–78. IEEE Computer Society. (page 8)

[18] Li, W., Agrawala, M., Curless, B., and Salesin, D. (2008). Automated generation of interactive 3d exploded view diagrams. *ACM Trans. Graph.*, 27(3):101:1–101:7. (page 9, 40)

[19] Mandl, D., Yi, K. M., Mohr, P., Roth, P. M., Fua, P., Lepetit, V., Schmalstieg, D., and Kalkofen, D. (2017). Learning lightprobes for mixed reality illumination. In Broll, W., Regenbrecht, H., and II, J. E. S., editors, *2017 IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2017, Nantes, France, October 9-13, 2017*, pages 82–89. IEEE Computer Society. (page 62)

[20] Marchand, É., Uchiyama, H., and Spindler, F. (2016). Pose estimation for augmented reality: A hands-on survey. *IEEE Trans. Vis. Comput. Graph.*, 22(12):2633–2651. (page 7)

[21] Microsoft (2019). Directed graph markup language (dgml). `https://docs.microsoft.com/en-us/visualstudio/modeling/directed-graph-markup-language-dgml-reference`. Accessed: 2019-03-15. (page 24, 68)

[22] Mohr, P., Kerbl, B., Donoser, M., Schmalstieg, D., and Kalkofen, D. (2015). Retargeting technical documentation to augmented reality. In Begole, B., Kim, J., Inkpen, K., and Woo, W., editors, *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*, pages 3337–3346. ACM. (page 1)

[23] Mohr, P., Mandl, D., Tatzgern, M., Veas, E. E., Schmalstieg, D., and Kalkofen, D. (2017). Retargeting video tutorials showing tools with surface contact to augmented reality. In Mark, G., Fussell, S. R., Lampe, C., m. c. schraefel, Hourcade, J. P., Appert, C., and Wigdor, D., editors, *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017.*, pages 6547–6558. ACM. (page 1)

[24] Padilla, R. (2019). Metrics for object detection. `https://github.com/rafaelpadilla/Object-Detection-Metrics`. Accessed: 2019-03-13. (page 18, 19)

[25] Pedoeem, J. and Huang, R. (2018). YOLO-LITE: A real-time object detection algorithm optimized for non-gpu computers. *CoRR*, abs/1811.05588. (page 6)

[26] Peng, X., Sun, B., Ali, K., and Saenko, K. (2014). Exploring invariances in deep convolutional neural networks using synthetic images. *CoRR*, abs/1412.7122. (page 4)

[27] Pollefeys, M., Koch, R., and Gool, L. J. V. (1999). Self-calibration and metric reconstruction inspite of varying and unknown intrinsic camera parameters. *International Journal of Computer Vision*, 32(1):7–25. (page 13, 21)

[28] Prisacariu, V. A. (2019). PWP3D code. `http://www.robots.ox.ac.uk/~victor/code.html`. Accessed: 2019-03-05. (page 8, 39)

[29] Prisacariu, V. A. and Reid, I. D. (2012). PWP3D: real-time segmentation and tracking of 3d objects. *International Journal of Computer Vision*, 98(3):335–354. (page 7, 8, 40, 56)

[30] Rajpura, P. S., Hegde, R. S., and Bojinov, H. (2017). Object detection using deep cnns trained on synthetic images. *CoRR*, abs/1706.06782. (page 4)

[31] Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. (2015). You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640. (page 5, 6)

[32] Redmon, J. and Farhadi, A. (2016). YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242. (page 6)

[33] Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *CoRR*, abs/1804.02767. (page 6, 14)

[34] Robot Perception & Navigation Group (2019). Android camera calibration. `https://github.com/rpng/android-camera-calibration`. Accessed: 2019-03-14. (page 21)

[35] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. S., Berg, A. C., and Li, F. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252. (page 4)

[36] Russell, B. C., Torralba, A., Murphy, K. P., and Freeman, W. T. (2008). Labelme: A database and web-based tool for image annotation. *International Journal of Computer Vision*, 77(1-3):157–173. (page 4)

[37] Schmaltz, C., Rosenhahn, B., Brox, T., Cremers, D., Weickert, J., Wietzke, L., and Sommer, G. (2007). Region-based pose tracking. In Martí, J., Benedí, J., Mendonça, A. M., and Serrat, J., editors, *Pattern Recognition and Image Analysis, Third Iberian Conference, IbPRIA 2007, Girona, Spain, June 6-8, 2007, Proceedings, Part II*, volume 4478 of *Lecture Notes in Computer Science*, pages 56–63. Springer. (page 7)

[38] Shafiee, M. J., Chywl, B., Li, F., and Wong, A. (2017). Fast YOLO: A fast you only look once system for real-time embedded object detection in video. *CoRR*, abs/1709.05943. (page 6)

[39] Tatzgern, M., Kalkofen, D., and Schmalstieg, D. (2010). Compact explosion diagrams. In McGuire, M. and Collomosse, J. P., editors, *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering 2010, Annecy, France, June 7-10, 2010*, pages 17–26. ACM. (page 9)

[40] Tian, Y., Li, X., Wang, K., and Wang, F. (2017). Training and testing object detectors with virtual images. *CoRR*, abs/1712.08470. (page 4)

[41] Unity Technologies (2019a). Unity3d asset bundles. `https://docs.unity3d.com/Manual/AssetBundles-Workflow.html`. Accessed: 2019-04-12. (page 63)

[42] Unity Technologies (2019b). Unity3d engine. `https://unity3d.com/`. Accessed: 2019-03-01. (page 4, 11)

[43] van Brakel, J.-P. (2019). Peak signal detection in realtime timeseries data. `https://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data/22640362#22640362`. Accessed: 2019-04-30. (page 25, 26, 41)

[44] Vuforia by PTC (2019). Vuforia model targets. `https://library.vuforia.com/features/objects/model-targets.html`. Accessed: 2019-04-08. (page 48)

[45] Wang, C., Chu, W., Chen, H., Hsu, C., and Chen, M. Y. (2014). Evertutor: automatically creating interactive guided tutorials on smartphones by user demonstration. In *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, pages 4027–4036. (page 1)

[46] Yao, B. Z., Yang, X., and Wu, T. (2009). Image parsing with stochastic grammar: The lotus hill dataset and inference scheme. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2009, Miami, FL, USA, 20-25 June, 2009*, page 8. IEEE Computer Society. (page 4)

[47] Ye, M., Wang, X., Yang, R., Ren, L., and Pollefeys, M. (2011). Accurate 3d pose estimation from a single depth image. In Metaxas, D. N., Quan, L., Sanfeliu, A., and Gool, L. J. V., editors, *IEEE International Conference on Computer Vision, ICCV 2011, Barcelona, Spain, November 6-13, 2011*, pages 731–738. IEEE Computer Society. (page 7)