Error Management for Augmented Reality Assembly Instructions

Ana Stanescu [1*] Peter Mohr [1] Franz Thaler [1,2] Mateusz Kozinski [1] Lucchas Ribeiro Skreinig [1]
Dieter Schmalstieg [1,4] Denis Kalkofen [1,3 †]
[1] Graz University of Technology [2] Medical University of Graz [3] Flinders University [4] University of Stuttgart

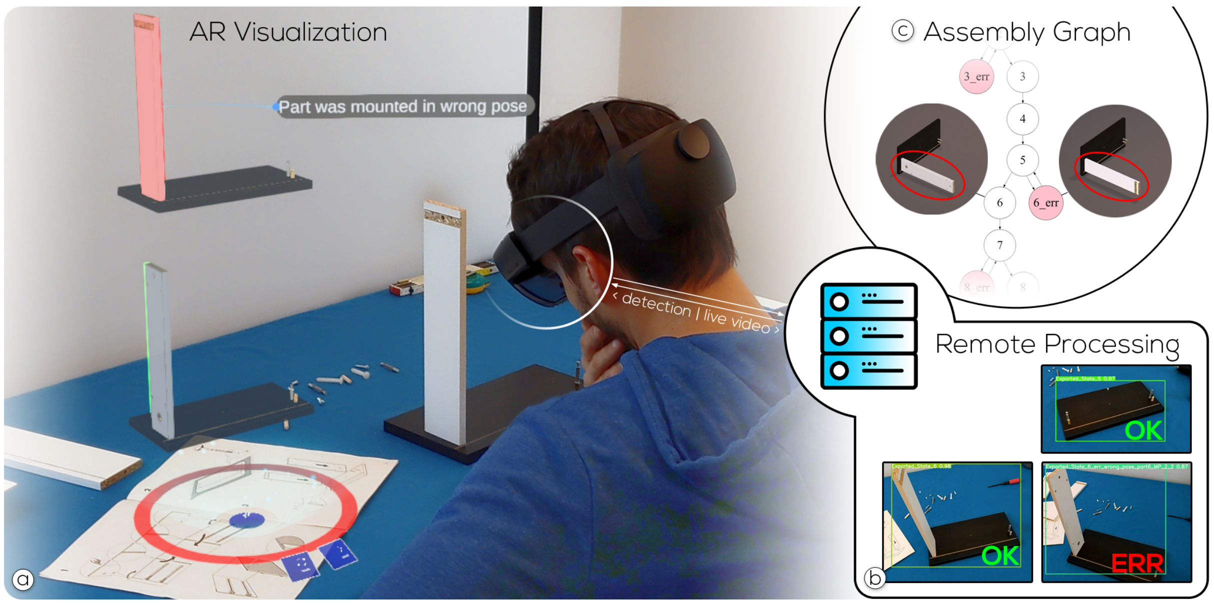

Figure 1: When a user makes an error while following an AR tutorial (a), our system is able to recognize the error state. (b) The state detector receives a video stream from the HMD and communicates the results back over a wireless network. (c) Once the error has been detected, the system guides the user back on the correct path in the assembly graph.

Abstract

Augmented reality (AR) lends itself to presenting visual instructions on how to assemble or disassemble an object. Splitting the assembly procedure into shorter steps and presenting the corresponding instructions in AR supports their comprehension. However, one can still misinterpret instructions and make errors while manipulating the object. While previous work supports detecting the occurrence of errors, we investigate handling such errors. This requires knowledge of the error at runtime of the application. Starting from a categorization of the errors, we investigate how to automatically derive common error states to generate training data. We introduce an extension to a state-of-the-art deep-learning-based object detector for supporting the detection of assembly states at real-time update rates, based on contrastive learning. We evaluated the proposed detector, showing that it outperforms the state-of-the-art, and we demonstrate our work with an AR application that alerts the user if errors occur and provides visual help to correct the error.

Index Terms: Human-centered computing—Human computer interaction (HCI)—Interaction paradigms—Mixed / augmented reality;

1 Introduction

Augmented Reality (AR) instructions provide visualizations to communicate the actions required to assemble or disassemble an object [39, chapter 7]. An assembly procedure is commonly split into several steps such that each step is simple to comprehend. Only one single step is presented at a time as a static or animated glyph. Showing instructions in AR [37], that is, directly within the user's field of view, can reduce the mental effort required to follow instructions [34, 16, 31]. Yet, even with AR, a certain probability remains that the user misinterprets the instructions and, as a consequence, introduces errors in the assembly.

Such errors can occur in various forms; for example, instructions are more likely to be misinterpreted when an object has many similar parts. Users may skip a step, confuse the order of steps, confuse object parts, place existing parts in wrong positions or poses, omit parts altogether, or repeat an action too often [36].

While some errors can be spotted immediately, many errors become evident only at the end of an assembly procedure, for example, when the final assembly does not match the intended outcome. Progress observation and discrepancy detection have been proposed to prevent erroneous assemblies. Progress observation validates every new assembly state [57, 50, 42]. Discrepancy checking additionally highlights the area of an assembly that differs from the expected object configuration [43, 55].

With progress observation and discrepancy checking, an AR application can notify the user of an erroneous assembly configuration. However, all these methods can only detect *that* a state is invalid, but not *why* it is invalid. Without precise information on the type of error, the feedback cannot include the means to correct the error. Consequently, error management is commonly restricted to either restarting the assembly procedure or waiting in error mode until a correct state is recognized. A method that provides instructions on how to backtrack and repair the error would be preferred.

Explicit error detection for AR has significant benefits, but its

*e-mail: stanescu@tugraz.at
†e-mail: kalkofen@tugraz.at

implementation is challenging. Errors need to reliably be detected in real-time, among a multitude of very similar correct and erroneous configurations, and under real-world conditions, such as varying lighting and partial occlusion. Enumerating the errors manually does not scale beyond trivial examples, so an automatic method is required to model the errors.

Since modeling all possible errors can become excessive, we focus on the most common ones. We start from an assembly graph expressing the sequence of valid states and extend it with additional states representing errors. We introduce a procedural error model that enumerates all common errors for a given assembly state and error category. This can easily be extended with additional error types. For assembly state detection at runtime, we build on an object detector specialized for assemblies, StateYOLO [42]. In our work, we introduce a novel approach based on contrastive learning for synthetic or mixed datasets, which improves the separation of features extracted from different states in our training data.

The extended assembly graph is essential for generating the training database for our new object detector. Obtaining training images for large assemblies that include all kinds of errors would be extremely tedious. Therefore, we generate synthetic images for training and mix them with real images.

In summary, we introduces the following contribution:

- A method for assembly graph augmentation to provide training examples for common user errors without the need to manually model them.
- A novel assembly state detector, which is based on contrastive learning from synthetic or mixed datasets and which has been integrated in YOLOv9, a current state-of-the-art real-time object detector.
- A new synthetic dataset that includes images of erroneous assembly states, with an accompanying test set consisting of captured real images, and a extension to the IndustReal dataset containing synthetic assembly error states.
- A proof of concept application for error detection for illustrating the real-time capabilities of our system running in AR.

2 Related work

Our work relates to AR assembly instructions with an automatic progression of steps based on user observation and error detection. Therefore, we review work on state and error detection and discuss literature investigating types of error as well as data structures to model the assembly procedure.

2.1 State detection

Several approaches for detecting the object's state have been investigated in the context of a real-time AR application. Wu et al. [57] propose an approach based on markerless tracking, while Wang et al. [50] track parts of an object based on template matching to check the completeness of the state. GBOT [28] approaches state detection by tracking each individual part of the object. Deriving detection from pose tracking has the advantage of knowing the pose of every component and their relation to each other, therefore, being able to detect wrongly mounted parts. However, in the case of small parts with heavy occlusions, tracking-based methods face challenges from lost objects. Thus, such methods are often limited to a low number of object parts and states.

Yamaguchi et al. [58] overcome the issue of tracking small parts. Their system incorporates the assembly sequence, which allows for identifying the removal of small parts by detecting the removal of larger tracked parts that are blocked. Although this approach reduces the limitations of object tracking-based configuration detectors, it requires a model that can be used for tracking at runtime and suffers from the challenges of template-based tracking.

Image-based detectors have been developed to learn detection from a set of images. For example, deep-learning-based approaches have been investigated to detect the states of an assembly directly from images showing the different configurations of an object [30, 42], or by first extracting regions of interest of the object to subsequently re-identify and classify their states [59]. Since AR devices are commonly equipped with one or more cameras, an image-based approach can be easily integrated into existing applications. However, none of the previous approaches for deep learning of assembly configurations has investigated error modeling and error detection so far.

2.2 Error management

Related engineering literature considers human-made errors using quantitative methods for assessing errors such as THERP, HEART or JHEDI [25]. While our focus is not on developing such guidelines, we make use of some considerations from the field to generate common errors. These types of error can be categorized similarly to the work of Riedel et al. [36] or the VDI guidelines [49], since their categories are extracted from observed human errors in manual assembly tasks, which is exactly what we want to investigate.

Riedel et al. [36] classify the types of errors that users make in manual assembly: errors by omission, execution errors, errors by confusion, and execution errors. They show that detecting the parts that are mounted and guiding the user with video instructions and LED beacons mounted at the parts' locations can decrease errors.

2.3 Error detection

Khuong et al. [22] study the effectiveness of AR instruction systems. They use an error detection measure by computing occupied voxels in a Lego assembly. The results indicate that having an error detection system can reduce the number of user errors during assembly. Furthermore, the use of wireframe overlays shows promising results [23]. Wasenmuller et al. [55] introduce a method that can highlight discrepancies using color. Their approach heavily relies on dense 3D data, accurate 3D registration, and precise tracking at runtime, which is demanding in terms of sensor performance and computational load. We largely avoid these costs by considering the problem at a more abstract level of states without needing precise alignment of all parts, thus avoiding the need for 3D reconstruction.

The work of Gupta et al. [14] handles Duplo pieces in a step-by-step assembly by checking their geometry. The system can detect errors by checking the occupancy in a voxelized space populated by parts of fixed voxel sizes. Similarly, the approach introduced by Stanescu et al. [43] performs error checking by comparing the point clouds of extracted part models. The approaches of Gupta et al. and Stanescu et al. are suitable for detecting state differences caused by more significant parts, but not where only subtle differences occur, such as the presence of a fold or the installation of a screw. Also, both approaches rely a 3D scene representation, which requires either a carefully prepared 3D model or stable and accurate 3D reconstruction in real time, which may not be easily available.

Argus [6] is an AR assistant toolkit that unified a series of machine learning tools, including the detection of objects and actions to perform error correction. This is proposed by detecting broad object categories learned by large vision models, for example, in a cooking scenario. We focus on more subtle errors in the context of assembly and their detection. Another machine learning approach [29] uses a variant of YOLOv3 to detect regions on a plate and compare the regions with known templates. The success is determined by the confidence of the prediction of the correct state, combined with the overlap of the bounding boxes in 2D space. If these are below a threshold, the labels are incorrect. For this approach to work, capture coverage needs to be ensured so that incorrect states are not labeled. In addition, very recent work deals with detecting wrong steps in a sequence [32] by using vision transform-

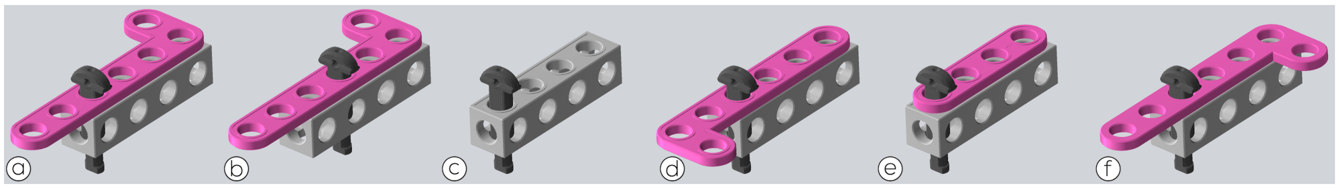

Figure 2: Example of types of errors that can occur during mounting: (a) correct state, (b) mounting screw at the wrong nearby position, (c) omission of the pink hinge, (d) mounting pink hinge in a 180° rotation, (e) mounting an incorrect (but visually similar to the pink hinge) part, (f) mounting the pink hinge in a flipped pose.

ers. This category of approaches focuses on tasks over time with more broadly defined steps, such as cooking or a medical procedure. This slightly differs from our goal, which is aimed at maintenance and training with small object differences.

Ghoddoosian et al. [12] deals with error detection in procedural videos, with a focus on anomaly detection, where anomalous actions are defined as skipping a step, wrongly repeating a step, and performing a wrong order of known steps. Their error detection method is based on a variant of the Viterbi algorithm, which can tell if the current step is likely anomalous or not.

In contrast to existing approaches, we identify the object states in a potentially complex assembly graph. Knowing how far the user has traversed can provide clues on the part that has wrongly been added to plan a recovery strategy. Out-of-distribution detection, in general, cannot identify what is wrong, only that a procedure deviates to a certain extent from a fixed linear procedure.

2.4 Assembly graphs

The literature provides several approaches to modeling an object's assembly. These approaches come from different research fields, such as multiple sub-domains of computer science, industrial engineering, and mechanical engineering. Similarly to previous work on assembly management in AR [42, 20, 35, 19], we choose to develop our approach based on assembly graphs. An assembly graph (sometimes also called *state graph*) is a directed graph that makes all possible (or permissible) actions explicit. It can be used directly as a state machine, where a single active node fully represents the current state of the assembly. Since a state-based approach aptly supports state detection, we make use of an assembly graph. In contrast to existing work, we extend assembly graphs with knowledge about possible error states because they directly provide classes to detect within their nodes. Also, given a start node, the currently active node in an assembly graph partially encodes the assembly history as the set of possible paths from the start node to the active node. This information is helpful in determining possible correct or incorrect actions that the user may take in the current state. More details on graphs introduced for the disassembly planning of objects can be found in the surveys of Zhou et al. [61] and Guo et al. [13].

2.5 Datasets of assembly procedures

The Epic Tent [17] dataset focuses on tent assembly and captures data from a first-person perspective, labeling video frames according to stages in the assembly of the tent, along with gaze data and user expertise data. The authors include nine error categories that were observed by 24 users. These cover motor errors, equipment misuse, out-of-order steps, equipment failure, omission of a step, searching for an item, correction of a prior error, slow movement, and repetition. The categories we focus on partially overlap with these. However, we focus on the error types that were empirically deduced by observing users in industrial assembly settings.

Assembly101 [41] is a recent dataset that captures humans assembling and disassembling toys. It includes mistakes made in the assembly as labels of coarse actions over time. The labels are *correct*, *correcting*, and *incorrect*. The approach of Ding et al. [9] follows up with the Assembly101 dataset, focusing on detecting errors. They investigate the problem as an action detection task and focus on step-order errors. Another recent dataset, HoloAssist, looks at first-person videos captured on a Microsoft Hololens and also includes errors [53]. The mentioned datasets do not contain object state labels, but instead focus on video procedures.

The IndustReal dataset [40] provides the assembly procedures of a 3D printable toy car model, including errors of different types, but all labeled with the same class. The authors propose several metrics for which the dataset can be used as a benchmark, including state detection and procedure step recognition. Leonardi et al. [26] generate synthetic datasets not of errors but of common object-hand interactions in industrial contexts as synthetic data. In contrast to their work, we want to generate common errors that could be made during object assembly of objects.

In addition to datasets that have been collected from physical observations, the generation of synthetic datasets is equally important [38], as frequently discussed in the literature [28, 30, 38]. Tools such as Nvidia's Onmiverse Replicator [33], BlenderProc [11] or Unity Perception [47] have been designed to generate large synthetic datasets (see Figure 3(b) for our generated example images).

3 Error model

In this work, we use assembly graphs, where each node represents an assembly configuration, and each edge represents the addition or removal of a part. The types of errors that a user can make while following an assembly procedure may fall into three categories:

1. **Invalid transition to a valid state.** An unexpected transition occurs to a valid state. For example, the user may mount two parts at once and, therefore, skips one state, or the user may go back in the assembly graph instead of progressing.
2. **Known error state.** A known error state is reached. This situation can be modeled if, for example, all common errors are modeled as states.
3. **Unknown error state.** This state is entered if an unknown invalid configuration is encountered.

We focus on managing errors of the second type, which are explicable given the assembly graph. In particular, we extend the graph of valid states with additional error-state nodes corresponding to invalid object configurations. While the causes for these types of errors are varied (see Table 1 for an overview), the actual errors can be combined into only two outcomes: Either an incorrect part is added, or a correct part is added in a wrong pose (translation and rotation). The condensation into just two outcomes makes it feasible to develop a procedural generator that can systematically enumerate common error states. We use the generator to extend the assembly graph with the most important erroneous states (see Figure 2). From these states, we synthesize images and annotations to train an object detector [51, 42]. We describe the procedural generator for creating these error states in more detail in Section 4.

3.1 Mounting points

We assume that objects are joined at fixed mounting points (e.g., holes for screws), as shown in Figure 3. The mounting points are

Error description	Category
Wrong location	Execution error
Wrong orientation	Execution error
Part forgotten	Omission error
Choosing a similar part	Confusion error
Wrong symmetric location	Confusion error

Table 1: Typical errors made by users during assembly and corresponding error categories.

3D locations with associated orientation (six degrees of freedom in total) placed on the surface of the object with their z-axis pointed outward in the mounting direction. We consider the forward mounting point, the z-axis, to be the mounting direction associated with a mounting point. The mounting points indicate where another part of the object with a matching mounting point can be attached. Every mounting point enables new state, which can be a valid state or an error state. Using the mounting points, we implement the error cases indicated in Table 1. Our error criteria are based previous work [36] where common errors made by users during assembly on a workbench are identified. In the conducted user study, five categories to model possible error states are identified. We chose to also use these categories since they were experimentally observed in a similar scenario to ours. To determine whether a part fits, we first check whether the mounting points are compatible. For example, a nut can be mounted on a pin. Another check determines whether the mounting point is occupied. We only generate common, plausible errors as a subset of all possible one-step errors. The number of generated states can be controlled by metaparameters such as the radius for mistakes around the correct mounting point.

A part can either occupy one or multiple mounting points in the case of an extended overlap. In Figure 3(a), the gray horizontal pin is mounted at only one point, but due to its elongated shape, the part occupies multiple mounting points at once, on both sides of the object, so no other part can be mounted at this location.

Each part is configured with a global transformation as well as a local offset from the origin in local coordinates to the mounting point used in a particular assembly step. For simplicity, we consider this offset to be unique for each part, and we apply the same offset whenever we mount the part.

3.2 Part similarity

To simulate part confusion, we implement a shape similarity check between parts. First, we implement a basic measure of shape similarity: The three dimensions of the parts' bounding boxes are compared, and the parts are accepted as similar if the difference of the largest dimension is not greater than 30%. Furthermore, we compare shapes by surface point registration, similar to Stanescu et al. [43]. We use Open3D [60] and Meshlab [7] to sample part meshes as point clouds. The point clouds are globally registered via iterative closest points, followed by a bidirectional point-to-point distance measurement. The distances are normalized, averaged, and thresholded, resulting in a value close to zero for similar parts. Other approaches could be considered, such as more advanced methods on 3D shape similarity and retrieval [4, 5, 2], but we found that the described approach suffices for our purposes.

3.3 Error state generation

With the preparations above, we can now express how error states should be generated.

Wrong location An execution error occurs if a part is incorrectly attached to a mounting point, typically near the correct one. To generate this case, we search for all mounting points within a radius of the correct point. The radius is empirically chosen to be 20% of the length of the object's bounding box. For each nearby mounting point, we place the part at the corresponding locations, orient it in the local coordinate system given by the mounting point, and add this configuration as an error state.

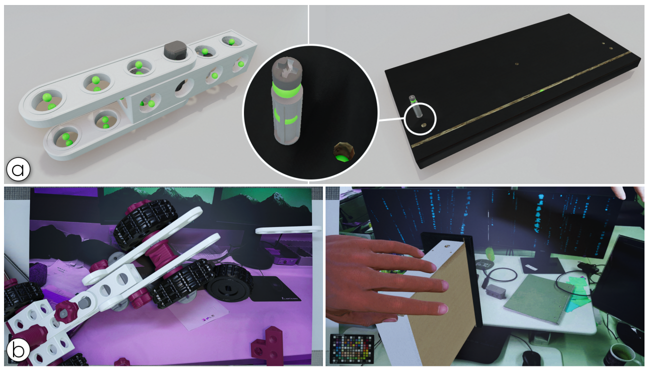

Figure 3: Mounting points and example data. (a) Mounting points (in green) on an object in a particular state. The mounting points can be in holes as well as, for example, on top of a pin, or on the object surface. (b) Samples from our synthetic datasets.

Wrong orientation In this type of execution error, a part is mounted at the correct mounting point, but in an incorrect pose. For machine assemblies, typical orientation errors are rotations by 90 degrees or flipping/rotation by 180 degrees. However, other pose errors can be considered as well, if necessary. To generate the error state, we start with the correct object transformation and rotate the object around the z-axis perpendicular to the mounting point. In our implementation, we only include a 180 degree rotation.

Part omission Here, the user skips a part and proceeds directly to the next part. This case is straightforward to implement; we simply create an assembly without the skipped part and a corresponding error state. For simplicity, we skip only one component, but more omitted parts could easily be considered.

Similar part An error by confusion happens when a visually similar part is mounted instead of the correct one. All spare parts that are similar to the current part according to the similarity metric described above replace it in the configuration. If multiple instances of the same part type are in the spare parts set, the configuration is generated only once per type. Duplicates of the correct part contained in the spare parts set are ignored.

Wrong symmetric location A second type of error by confusion involves mounting a part at an incorrect mounting point due to symmetry. We compute the symmetry of the current state of the object before mounting the next part by mirroring the model. If there is a mounting point in the flipped object very close to the original mounting point, we generate an error state with the part mounted to this new point.

4 Procedural error generator

We implemented the procedural error state generator as an application in the Unity game engine. Our aim is to automatically generate all valid states and the error states described in the last section and to render training images for deep learning.

4.1 Input data requirements

Our generator expects the input to meet the following requirements:

First, we need the geometry, which is usually derived from a CAD model, segmented into parts. In our implementation, we use object partitions [20]. A partition can consist of a single part or multiple parts that are added at once to transition to the next assembly state. For simplicity, we will use the term 'part' in the rest of this paper, even if it refers to a multi-part partition. Errors within a partition are not supported; they can be achieved by defining more intermediate states, as in Figure 5 for the toy car.

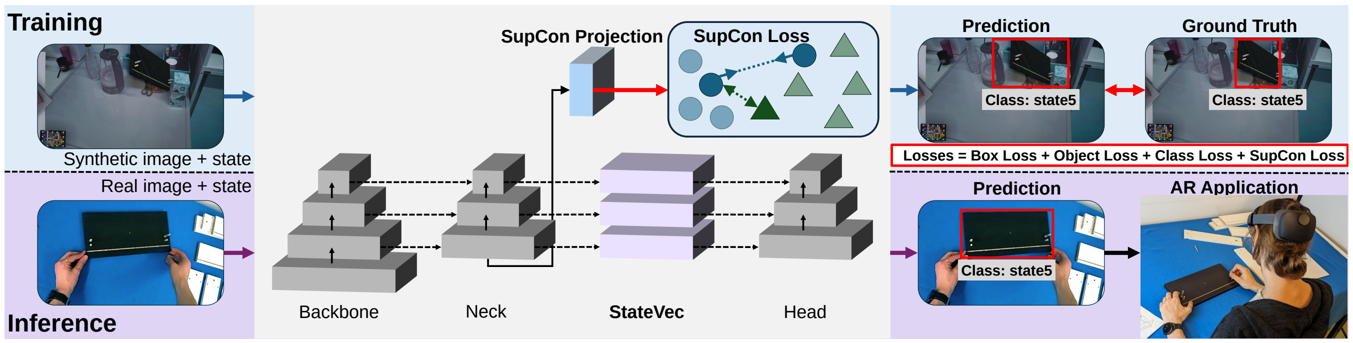

Figure 4: Architecture overview of YOLOv9 (GELAN), split into the training pipeline on the top and the inference pipeline on the bottom. The supervised contrastive (SupCon) loss, which is built on the highest resolution branch of the neck, is used only during training. The state-aware layer StateVec is placed between the network neck and head, and receives the previous state as input.

Second, we require the assembly graph of the correct states. The assembly graph does not need to cover all the possible states. It may be restricted to a specific order or preferred sequence. In each state transition, only one part can be added. The part is taken from the set of spare parts available, which initially consists of all parts. When a part is added to a step, it is removed from the spare parts.

Third, we assume that mounting points are defined as annotations to the part geometry, together with mounting point compatibility and mounting pose per part. In a practical scenario, this information would be created by geometric disassembly planning [20, 54, 10], deep learning [56], or manual authoring. We place only one mounting point between any pair of parts. For simplicity, we do not consider multiple mounting points or mounting edges.

4.2 Generation workflow

The generator outputs a set of Unity prefabs corresponding to the correct assembly states. For each state, a set of errors of various types is created. Each prefab has a corresponding label that specifies which parts are present and at which part an error is introduced. This representation is forwarded to the dataset renderer, which generates training images and training labels for each state.

Generation progresses along a selected path through the assembly graph. To generate the valid states, we perform a depth-first traversal of the assembly graph and place the parts in their correct position at the correct mounting points For each incoming edge of a node in the assembly graph, all errors are generated according to the heuristics. After exporting all prefabs belonging to the state (correct as well as errors), the next state in the selected path is processed.

4.3 Rendering

For rendering the objects, we rely on the *Perception* framework [18, 3] which specializes in the generation of synthetic data using the Unity engine. For domain randomization, we vary the pose of the object state, as well as the backgrounds and the light color, and render with Unity's HDRP rendering pipeline. The camera parameters are set to resemble those of a HoloLens device. For the backgrounds, we collect our own images of various messy desks that simulate an AR workbench environment.

We exclude images where the next part is completely occluded and cannot be observed in the rendered frame, since we assume that the user typically looks at the part while mounting it. An image mask of the latest added part is used to identify and discard frames where the part is visible in fewer than a given number of pixels.

In addition, we use measures to further increase the realism of the data. For example, we render models with a shader that approximates the appearance of the 3D printed parts we use in AR experiments as closely as possible by simulating the filament surface structure using a normal map. For the 'drawer' dataset we use scanned textures to closely resemble the appearance of the object. The images show the assembly in the current state as defined in the prefab, while spare parts are placed at random locations in the scene. Furthermore, we introduce hands as distractors and occluders in random poses to simulate the presence of the user (Fig. 3b).

5 STATE DETECTION NETWORK

Similarly to previous work on state detection [40, 42] we use YOLO, a leading real-time object detector in its latest version [52]. We experiment with two extensions, state-aware detection and contrastive learning, applied to the GELAN-C variant of YOLOv9.

For state detection with a predefined assembly graph, we implement the state-aware detection module proposed by Stanescu et al. [42], which was shown to improve YOLOv7 detection by exploiting an assembly graph. This module takes the previously detected state as input to a module *StateVec*. An internal representation of that state is added to the features of the network. The module is located between the neck and head of YOLOv7, and we place it in the same position in YOLOv9, right before the *DDetect* module.

Due to the visual similarity of our classes to each other, sometimes with only very subtle differences, we employ supervised contrastive learning [21] to further improve the state detection performance. Figure 4 shows these enhancements to the network architecture. Contrastive learning has been successfully applied in YOLOv5 to enhance the performance on thermal images [46]. However, it has not yet been explored in the context of state detection and when models are trained with synthetic images. The main idea behind contrastive learning is to achieve class-based clustering in feature space by attracting the feature representation of images of the same class to one another, while repelling the feature representation of images of different classes. This is implemented by obtaining the feature representation r of an image in a hidden layer of the network during training. Due to the high dimensionality of the intermediate feature representation, we follow related work [15, 21] and use a projection head $H(\cdot)$ to project the feature representation r to a low-dimensional projected feature vector $z = H(r)$. The contrastive loss is then computed for pairs of projected feature vectors, which are attracted to or repelled on the basis of their class by minimizing or maximizing their distance in terms of cosine similarity. We apply the contrastive loss to the entire batch to ensure that a sufficient number of same- and different-class samples are available for each weight update. In this way, *each* sample is considered as the anchor that attracts or repels *all* other samples in the same batch. The supervised contrastive loss is defined as in the work of Tian et al. [45]. Applied to YOLOv9, it is added as an additional term to the original three loss terms L_{box}, L_{cls} and L_{dfl}, resulting in the final optimization target:

$$L = w_1 L_{box} + w_2 L_{cls} + w_3 L_{dfl} + w_4 L_{supcon}, \tag{1}$$

where $w_1 \dots w_4$ are used as weights for the individual loss terms.

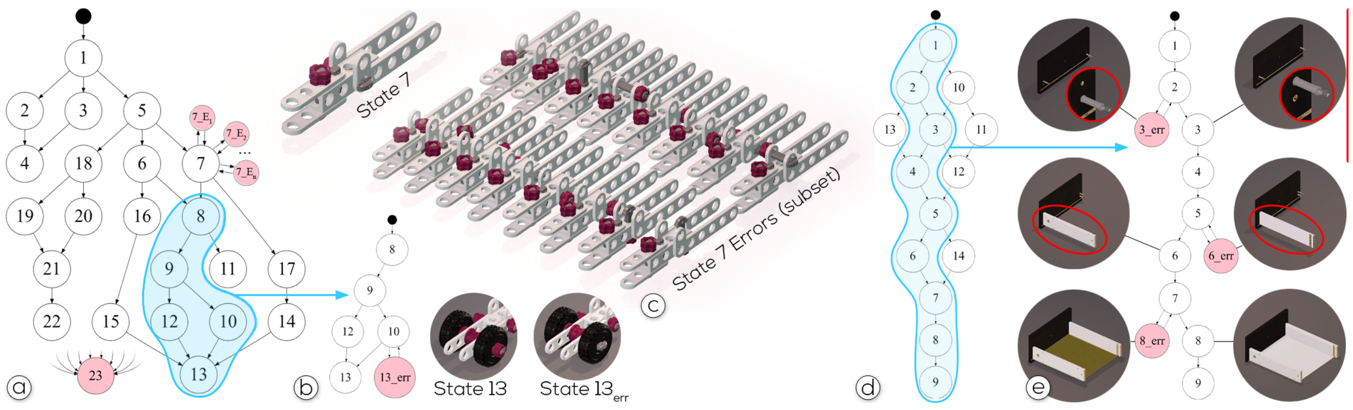

Figure 5: Error states. (a) Original assembly graph of the IndustReal dataset shown in black. (b) Subgraph of the IndustReal dataset used for evaluation, with added error states. (c) Example subset of error states generated by our system for State 7 of the IndustReal dataset. (d) Complete assembly graph of the IKEA drawer. (e). Subset of the drawer dataset used for evaluation, with generated error states.

Implementation Details Our implementation directly employs YOLOv9 as the encoder for the images from which we obtain the feature representation r from the neck part of the network, specifically, at the output of layer 15 as defined in the network configuration file. The projection head $H(\cdot)$ is implemented as an adaptive max-pooling with the output size 1, followed by a linear layer with the same feature size as the input. L2 normalization is applied [45] before computing the contrastive loss. Since the projection head is only required during training, it is not used after training finishes and, consequently, does not impose any additional computational cost during inference (86 ms per image).

6 Results

For our experiments, we focus on two datasets. The first dataset represents furniture, specifically a set of drawers. The second dataset is taken from the STEMFIE project [24, 40], with its parts modified to more closely resemble their 3D printed counterparts.

6.1 Enhanced assembly graphs

Our datasets are given as a hierarchy which implicitly contains the structure of the assembly graph.

Toy car We define the partitions of the IndustReal object in the same way as defined in the original paper [40], which allows us to combine the output of our procedural error generator with the original dataset. The original dataset contains 22 success states, to which we add the automatically generated error states. The original dataset also includes state 23, which is a collective label for all errors that were noticed during assembly. The partitions are defined as consisting of multiple parts, and multiple partitions can be added in one step. For a practical AR application, it would be possible to modify the dataset by adding intermediate states and further splitting the partitions before generating the synthetic training data.

In Figure 5b, we show an example of the error states generated for this object. The user can easily make the error of mounting the horizontal pin in State 7 at a nearby location. Our error generator creates configurations that reflect this problem.

Drawer The partitions of our second object, the drawer, are defined as shown in Figure 5. For the drawer, each state adds exactly one part to the assembly. We exemplify some generated errors in Figure 5e, namely the wrong mounting location for the screws, or the flipping of the side and bottom part.

6.2 Datasets used for comparison

We create a few variants of datasets for our experiments using subsets of our generated error states: IndustReal-small-err and Drawer-small-err. Their assembly graphs are shown in Figure 5.

While Drawer is a dataset introduced in this paper, IndustReal-small-err contains a subset of real data from the industReal dataset and synthetic data rendered using the output of our error generator. We render each class with roughly 1500 samples/class. IndustReal-small-err contains 6144 images in the training, 876 in the validation, and 1754 in the synthetic test set, and 258 in the real test set that comes directly from the IndustReal dataset, including the recorded error class. The Drawer-small-err contains 10016 images in the training, 1445 in the validation, and 2886 in the synthetic test set, as well as 484 in the real test set. The synthetic test sets were obtained by a 70, 10, 20 percent split for training, validation and testing. The test set is independent and based on real data. The validation set is only used to select the best model in terms of mAP.

Furthermore, we also create two variations of the IndustReal-small-err dataset by replacing a small set of samples with real data, 5% and 2%, respectively. We use these for the experiment of mixing real and synthetic data, since real training data is available in the public dataset. We only add real data to the success classes, not to the error classes, since in a real-world application it would be challenging to capture and observe errors as users commit them.

6.3 Detecting states and errors

Model comparison We evaluate the proposed state detector by comparing it with related work, as well as by performing a step-by-step ablation of our extensions to YOLOv9. As it is a standard measure for object detection, we evaluate methods using the mean average precision (mAP), specifically, the mAP50 and the mAP50-90 metrics, as well as a confidence of 0.001 and an IOU threshold of 0.45. Table 2 shows that the proposed YOLOv9 State Supcon method outperforms related work on both the IndustReal-small-err and the Drawer-small-err dataset. Closer inspection of the ablation results on the IndustReal-small-err dataset reveals that both YOLOv9 State and YOLOv9 Supcon result in improvements compared to the YOLOv9 baseline. However, in the Drawer-small-err data set, YOLOv9 Supcon by itself underperformed on the mAP50-90 metric compared to the YOLOv9 baseline, which we assume to be caused by some very minimal visual differences like the presence or position of small screws between individual states that are challenging to correctly identify in some cases. Nevertheless, the proposed YOLOv9 State Supcon method achieves the overall best performance when compared to the step-by-step ablations, YOLOv9 State and YOLOv9 Supcon. Furthermore, when observing the results on the Industreal-bboxed dataset, YOLOv9 Supcon also improves over related work as well as the YOLOv9 baseline, most notably on the mAP50 metric. Since all error states are defined as a single error class in the Industreal-bboxed dataset, the state-aware module is not directly applicable without separating

Table 2: Different variants of the state detectors (YOLOv7 and YOLOv9), with constrastive loss and state-aware module ehancements.

Dataset Train (synthetic)	Dataset Test (real)	#States	Approach	mAP50	mAP50-90
IndustReal-small-err	IndustReal-small-err	6	YOLOv7 [51]	0.718	0.378
			YOLOv7 State [42]	0.885	0.515
			YOLOv9 [52]	0.92	0.597
			YOLOv9 State only	0.981	0.637
			YOLOv9 Supcon only	0.957	0.616
			YOLOv9 State Supcon (ours)	0.982	**0.653**
Drawer-small-err	Drawer-small-err	11	YOLOv7 [51]	0.28	0.236
			YOLOv7 State [42]	0.582	0.49
			YOLOv9 [52]	0.486	0.44
			YOLOv9 State only	0.678	0.609
			YOLOv9 Supcon only	0.468	0.412
			YOLOv9 State Supcon (ours)	0.681	**0.611**
IndustReal-bboxed	IndustReal-bboxed	23	YOLOv8 [40]	0.573	-
			YOLOv9	0.59	0.324
			YOLOv9 Supcon only	0.611	**0.327**

Table 3: Experiments with different mixtures of synthetic and real data for training, added only to the success states.

Dataset Train	Dataset Test	#States	Approach	mAP50	mAP50-90
IndustReal-small-err	IndustReal-small-err	6	YOLOv9 [52]	0.92	0.597
			YOLOv9 State Supcon	0.982	0.653
IndustReal-small-err 98% synth 2% real	IndustReal-small-err	6	YOLOv9 [52]	0.995	0.79
			YOLOv9 State only	0.995	0.826
			YOLOv9 Supcon only	0.995	0.815
			YOLOv9 State Supcon	0.966	**0.829**
IndustReal-small-err 95% synth 5% real	IndustReal-small-err	6	YOLOv9 [52]	0.987	0.812
			YOLOv9 State only	0.995	0.834
			YOLOv9 Supcon only	0.995	0.837
			YOLOv9 State Supcon	0.994	**0.838**

different error states into different error classes and, consequently, cannot be evaluated on this dataset.

Furthermore, in Table 3 we experiment with mixing real and synthetic data when training the model on the IndustReal-small-err dataset. Specifically, results are provided for models trained 98% of synthetic and 2% of real data as well as 95% of synthetic and 5% of real data. Compared to the results trained exclusively on synthetic data, it can be observed that using only very small proportions like 2% of real data already greatly improves the performance of the state detector. In addition, an improvement can be achieved by increasing the proportion of real data from 2% to 5% for all experiments. Similarly to before, the YOLOv9 State and Supcon variants both lead to an improvement compared to the baseline YOLOv9, while the proposed YOLOv9 State Supcon method achieves the overall best results on the mAP50-90 metric.

We also show the performance per class for the mixed data. As shown in Table 4, adding a small proportion of real data only to the success classes of the synthetic training set leads to an increase in performance of all classes, including the error class. So, in this case, error state 13 also benefits in terms of mAP from adding real data to the other states, although it is synthetic in both training rounds.

Experimental Setup For contrastive loss, we used parameters 0.1 for temperature τ, and the parameter λ to weigh the loss term was experimentally chosen to be 0.1 (YOLOv7) or 0.01 (YOLOv9). The experiments are trained for 150 epochs, at a learning rate of 0.01 in YOLOv9 and 0.01-0.1 in YOLOv7, as in the public repositories with the original implementations of these networks. We used the SGD optimizer, batch size 32, and an image size of 512×512.

Since the StateYOLO training might process the same image twice, just with different previous states, the experiments with the state-aware networks are stopped after reaching the same number of iterations as the ones without the states. For the StateVec module, we use the StateYOLO-mlp architecture [42] in YOLOv9. The IndustReal comparison experiment was run with the ADAM optimizer for 50 epochs, with the original hyperparameters [40].

Table 4: Performance per class for synthetic-only data trained on the baseline model YOLOv9, and our best model YOLOv9 State Supcon with the 95/5 data mix.

Class (State)	mAP50-90 Synth only training	Class (State)	mAP50-90 Mixed training
S 8	0.439	S 8 (95% synth 5% real)	0.754
S 9	0.578	S 9 (95% synth 5% real)	0.821
S 12	0.715	S 12 (95% synth 5% real)	0.889
S 10	0.751	S 10 (95% synth 5% real)	0.769
S 13	0.806	S 13 (95% synth 5% real)	0.906
S 13_err	0.631	S 13_err (100% synth)	0.889

Data Visualizations We visualize the clusters of our data embeddings using t-SNE [48], a non-linear dimensionality reduction algorithm that preserves the spatial relationship between samples. We run t-SNE on the test sets of IndustReal-small-err, and compute it at the network position where the contrastive loss is enforced. The t-SNE results in Figure 6 show that contrastive loss has an impact on hidden features, clustering them in a meaningful way, and showing the effect of clustering similar classes. In both plots, but especially in the synthetic data images on the left side, it can be observed that the consecutive classes are in each other's vicinity. This reflects the geometric difference of the states in the assembly graph, where neighboring states are more similar.

6.4 AR application

We implemented an AR instructions application using our state detector in the Unity game engine. The application and the imple-

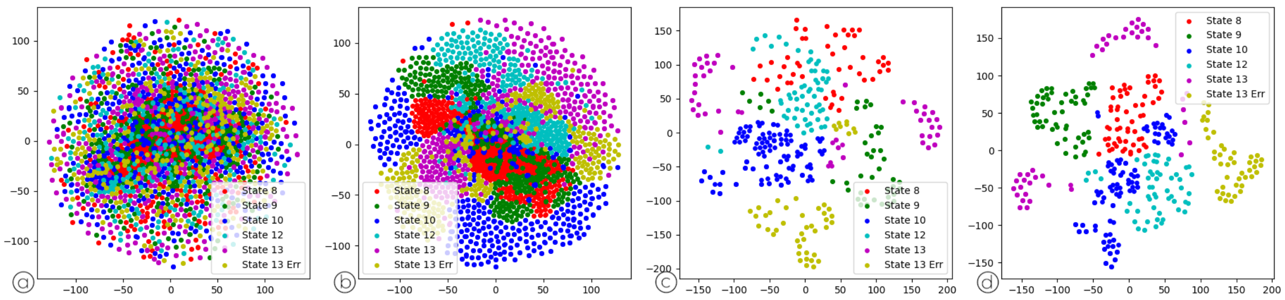

Figure 6: Embeddings of the states from the IndustReal-small-err dataset, visualized with t-SNE in (a) YOLOv9 on synthetic data, (b) YOLOv9 with contrastive loss on synthetic data, (c) YOLOv9 on real data, and (d) YOLOv9 with contrastive loss on real data.

mented workflow are illustrated in Figure 1. The instruction application guides the user through an assembly procedure, continuously detecting and proposing corrections to errors. It shows a 3D model of the object next to the physical work area, indicating which part needs to be mounted next by highlighting the part and showing a magnified copy. We believe that showing the part separately and in an unmounted state is important to let the user clearly see the relevant part, especially if it is small and disappears in the unfinished assembly after mounting. Showing only assembled states would force the user to compare the states, a task which can be difficult due to change blindness [1]. For the same reason, we show an animation of the repair process if an error state is detected. If parts are different, the wrong part is highlighted in red in the error display area, while the correct part is shown in the instruction for the current step as can be seen in Figure 7.

The AR application runs on the HoloLens 2, and our state detector runs on a server on the local network. We use the HLSS project for live streaming the images [8]. The detector aggregates votes over a small window of consecutive frames to robustly confirm the detection of a new assembly state. If a transition to the new state is found in the assembly graph, the state progresses to the confirmed state, and the visualization is updated.

7 Discussion

When generating error states, we consider single errors, but we do not handle multiple linked errors encountered after progressing through multiple steps. Our rationale is based on the expectation that the user makes only one error at a time, provided the AR application can immediately detect the error and suggest its rectification. Studying chained errors represents an interesting direction for future work, but we believe that detecting individual errors represents a worthy contribution in itself: In many applications feedback should be provided instantly if the first user error is detected. A different task would be detecting defects, like fractures or corrosion. Our method can be extended to detect such defects provided the corresponding training data can be collected or generated.

Furthermore, the error state generator could be improved to cover a broader range of error criteria, handle collision detection, and support more complex forms of mounting. The error state configurations could be generated using a neural network instead of a heuristic classification. A model could be trained to generate likely errors made by humans, given either example error configurations, example videos, or text descriptions as input.

The need for assembly graphs is not specific to our error detection method [30, 57]. Our method does require extending the graph with states representing errors, but in all critical applications, maintenance and assembly procedures are analyzed for possible errors. Therefore, adding the corresponding states to the assembly graph does not incur a significant cost. On the contrary, our procedural error generator automates this process.

The increased number of states leads to a larger training set, as images need to be generated for each error state. However, the

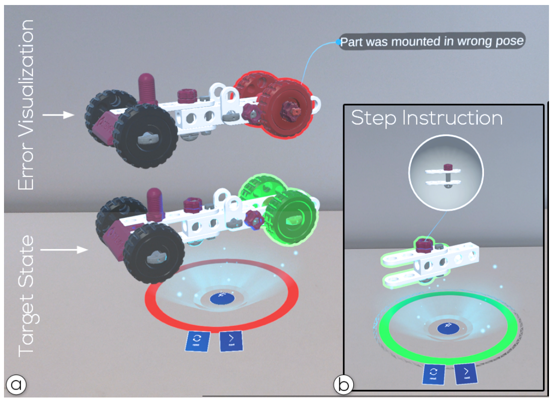

Figure 7: Error visualization. (a) Our proof of concept application shows a red highlight of the error, namely the wrongfully flipped pin holding the wheels, and suggests the correct configuration in green. (b) The next instruction is shown once a step was completed.

required training time does not grow linearly with the number of states, because images of the error states are visually similar to the images of the correct states. Since the error states contain the same parts as the correct states, distinguishing them does not require recognizing new visual cues, but rather differentiating compositions of already known elements. Investigating the scale of the effects described above represents an interesting direction for future work.

8 Conclusions and Future Work

We introduce an approach for state and error detection, including a method for generating synthetic data, and we provide a proof-of-concept AR guidance application. Our work demonstrates that joint state detection and error correction is technically feasible even for very similar assembly states. Although we believe that our current system is already beneficial for providing important information during AR guidance, we see several directions for future work.

We also plan to improve the realism of the synthetic examples. For example, GrabNet [44] or EgoGen [27] could be used to replace random hand poses with common mounting gestures. In addition, we plan to integrate object tracking. With an object tracking approach such as GBOT [28], the user interface of the AR application can be improved. For example, the next part to manipulate can be shown in 3D, registered on top of or next to the partial assembly. However, tracking introduces additional challenges in terms of stability and run-time.

Also, in the future we intend to perform user studies to evaluate how our error detection method can improve the user experience.

In summary, the improvements we introduced to the network are very well suited for our use case, while the inference remains real-time. We show that, even with small batches, contrastive learning improves the measured performance. To support future research, we publish our datasets online.

References

[1] M. Agrawala, D. Phan, J. Heiser, J. Haymaker, J. Klingner, P. Hanrahan, and B. Tversky. Designing effective step-by-step assembly instructions. *ACM Transactions on Graphics (TOG)*, 22(3):828–837, 2003. 8

[2] S. Biasotti, A. Cerri, A. Bronstein, and M. Bronstein. Recent trends, applications, and perspectives in 3d shape similarity assessment. In *Computer graphics forum*, vol. 35, pp. 87–119. Wiley Online Library, 2016. 4

[3] S. Borkman, A. Crespi, S. Dhakad, S. Ganguly, J. Hogins, Y.-C. Jhang, M. Kamalzadeh, B. Li, S. Leal, P. Parisi, et al. Unity perception: Generate synthetic data for computer vision. *arXiv preprint arXiv:2107.04259*, 2021. 5

[4] B. Bustos, D. A. Keim, D. Saupe, T. Schreck, and D. V. Vranic. Using entropy impurity for improved 3d object similarity search. In *2004 IEEE International Conference on Multimedia and Expo (ICME)(IEEE Cat. No. 04TH8763)*, vol. 2, pp. 1303–1306. IEEE, 2004. 4

[5] B. Bustos, D. A. Keim, D. Saupe, T. Schreck, and D. V. Vranić. Feature-based similarity search in 3d object databases. *ACM Computing Surveys (CSUR)*, 37(4):345–387, 2005. 4

[6] S. Castelo, J. Rulff, E. McGowan, B. Steers, G. Wu, S. Chen, I. Roman, R. Lopez, E. Brewer, C. Zhao, et al. Argus: Visualization of ai-assisted task guidance in ar. *IEEE Transactions on Visualization and Computer Graphics*, 2023. 2

[7] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In V. Scarano, R. D. Chiara, and U. Erra, eds., *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008. doi: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136 4

[8] J. C. Dibene and E. Dunn. Hololens 2 sensor streaming. *arXiv preprint arXiv:2211.02648*, 2022. 8

[9] G. Ding, F. Sener, S. Ma, and A. Yao. Every mistake counts in assembly. *arXiv preprint arXiv:2307.16453*, 2023. 3

[10] C.-W. Fu, P. Song, X. Yan, L. W. Yang, P. K. Jayaraman, and D. Cohen-Or. Computational interlocking furniture assembly. *ACM Transactions on Graphics (TOG)*, 34(4):1–11, 2015. 5

[11] German-Aerospace-Center. Blenderproc. Project website, 2024. https://dlr-rm.github.io/BlenderProc/. 3

[12] R. Ghoddoosian, I. Dwivedi, N. Agarwal, and B. Dariush. Weakly-supervised action segmentation and unseen error detection in anomalous instructional videos. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10128–10138, 2023. 3

[13] X. Guo, M. Zhou, A. Abusorrah, F. Alsokhiry, and K. Sedraoui. Disassembly sequence planning: a survey. *IEEE/CAA Journal of Automatica Sinica*, 8(7):1308–1324, 2020. 3

[14] A. Gupta, D. Fox, B. Curless, and M. Cohen. Duplotrack: a real-time system for authoring and guiding duplo block assembly. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pp. 389–402, 2012. doi: 10.1145/2380116.2380167 2

[15] M. Gutmann and A. Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 297–304. JMLR Workshop and Conference Proceedings, 2010. 5

[16] S. J. Henderson and S. K. Feiner. Augmented reality in the psychomotor phase of a procedural task. In *Proc. International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 191–200. IEEE, 2011. doi: 10.1109/ISMAR.2011.6092386 1

[17] Y. Jang, B. Sullivan, C. Ludwig, I. Gilchrist, D. Damen, and W. Mayol-Cuevas. Epic-tent: An egocentric video dataset for camping tent assembly. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pp. 0–0, 2019. 3

[18] Y.-C. Jhang, A. Palmar, B. Li, S. Dhakad, S. K. Vishwakarma, J. Hogins, A. Crespi, C. Kerr, S. Chockalingam, C. Romero, A. Thaman, and S. Ganguly. Training a performant object detection ML model on synthetic data using Unity Perception tools. https://blogs.unity3d.com/2020/09/17/training-a-performant-object-detection-ml-model-on-synthetic-data-using-unity-computer-vision-tools/, Sep 2020. 5

[19] D. Kalkofen, M. Tatzgern, and D. Schmalstieg. Explosion diagrams in augmented reality. In *Proc. IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 71–78, 2009. doi: 10.1109/VR.2009.4811001 3

[20] B. Kerbl, D. Kalkofen, M. Steinberger, and D. Schmalstieg. Interactive disassembly planning for complex objects. *Computer Graphics Forum*, 34(2):287–297, may 2015. doi: 10.1111/cgf.12560 3, 4, 5

[21] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673, 2020. 5

[22] B. M. Khuong, K. Kiyokawa, A. Miller, J. J. La Viola, T. Mashita, and H. Takemura. The effectiveness of an ar-based context-aware assembly support system in object assembly. In *2014 IEEE Virtual Reality (VR)*, pp. 57–62. IEEE, 2014. 2

[23] B. M. Khuong, K. Kiyokawa, A. Miller, J. J. LaViola Jr, T. Mashita, and H. Takemura. Context-related visualization modes of an ar-based context-aware assembly support system in object assembly (special issue mixed reality). *Transactions of the Virtual Reality Society of Japan*, 19(2):195–205, 2014. 2

[24] P. Kiefe. Stemfie. https://stemfie.org/sps-000001, 2022. 6

[25] B. Kirwan. The validation of three human reliability quantification techniques—THERP, HEART and JHEDI: Part 1—technique descriptions and validation issues. *Applied ergonomics*, 27(6):359–373, 1996. 2

[26] R. Leonardi, A. Furnari, F. Ragusa, and G. M. Farinella. Are synthetic data useful for egocentric hand-object interaction detection? an investigation and the hoi-synth domain adaptation benchmark. *arXiv preprint arXiv:2312.02672*, 2023. 3

[27] G. Li, K. Zhao, S. Zhang, X. Lyu, M. Dusmanu, Y. Zhang, M. Pollefeys, and S. Tang. EgoGen: An Egocentric Synthetic Data Generator. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. 8

[28] S. Li, H. Schieber, N. Corell, B. Egger, J. Kreimeier, and D. Roth. GBOT: Graph-Based 3D Object Tracking for Augmented Reality-Assisted Assembly Guidance. In *Proc. IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 513–523, 2024. doi: 10.1109/VR58804.2024.00072 2, 3, 8

[29] W. Li, X. Aibo, M. Wei, W. Zuo, and R. Li. Deep learning-based augmented reality work instruction assistance system for complex manual assembly. *Journal of Manufacturing Systems*, 73:307–319, 2024. 2

[30] H. Liu, Y. Su, J. Rambach, A. Pagani, and D. Stricker. Tga: Two-level group attention for assembly state detection. In *IEEE Int. Symp. on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, pp. 258–263. IEEE, 2020. doi: 10.1109/ISMAR-Adjunct51615.2020.00074 2, 3, 8

[31] E. Marino, L. Barbieri, F. Bruno, and M. Muzzupappa. Assessing user performance in augmented reality assembly guidance for industry 4.0 operators. *Computers in Industry*, 157:104085, 2024. 1

[32] M. Narasimhan, L. Yu, S. Bell, N. Zhang, and T. Darrell. Learning and verification of task structure in instructional videos. *arXiv preprint arXiv:2303.13519*, 2023. 2

[33] NVIDIA. Omniverse replicator. Online announcement, 2024. https://nvidianews.nvidia.com/news/nvidia-announces-omniverse-replicator-synthetic-data-generation-engine-for-training-ais. 3

[34] S. Pongnumkul, M. Dontcheva, W. Li, J. Wang, L. Bourdev, S. Avidan, and M. F. Cohen. Pause-and-play: automatically linking screencast video tutorials with applications. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pp. 135–144, 2011. doi: 10.1145/2047196.2047213 1

[35] V. Raghavan, J. Molineros, and R. Sharma. Interactive evaluation of assembly sequences using augmented reality. *IEEE Transactions on Robotics and Automation*, 15(3):435–449, 1999. 3

[36] A. Riedel, J. Gerlach, M. Dietsch, S. Herbst, F. Engelmann, N. Brehm, and T. Pfeifroth. A deep learning-based worker assistance system for error prevention: case study in a real-world manual assembly. *Advances in Production Engineering & Management*, 16(4):393–404, 2021. 1, 2, 4

[37] C. Rolim, D. Schmalstieg, D. Kalkofen, and V. Teichrieb. Design guidelines for generating augmented reality instructions. In *Proceed-*

ings of the 2015 IEEE International Symposium on Mixed and Augmented Reality, p. 120–123. IEEE Computer Society, USA, 2015. doi: 10.1109/ISMAR.2015.36 1

[38] H. Schieber, K. C. Demir, C. Kleinbeck, S. H. Yang, and D. Roth. Indoor synthetic data generation: A systematic review. *Computer Vision and Image Understanding*, p. 103907, 2024. 3

[39] D. Schmalstieg and T. Höllerer. *Augmented Reality - Principles and Practice*. Addison-Wesley Professional, June 2016. 1

[40] T. J. Schoonbeek, T. Houben, H. Onvlee, P. H. de With, and F. Van der Sommen. IndustReal: A Dataset for Procedure Step Recognition Handling Execution Errors in Egocentric Videos in an Industrial-Like Setting. In *2024 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pp. 4353–4362, 2024. doi: 10.1109/WACV57701.2024.00431 3, 5, 6, 7

[41] F. Sener, D. Chatterjee, D. Shelepov, K. He, D. Singhania, R. Wang, and A. Yao. Assembly101: A large-scale multi-view video dataset for understanding procedural activities. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21096–21106, 2022. 3

[42] A. Stanescu, P. Mohr, M. Kozinski, S. Mori, D. Schmalstieg, and D. Kalkofen. State-Aware Configuration Detection for Augmented Reality Step-by-Step Tutorials. In *Proc. International Symposium on Mixed and Augmented Reality (ISMAR)*, 2023. doi: 10.1109/ISMAR59233.2023.00030 1, 2, 3, 5, 7

[43] A. Stanescu, P. Mohr, D. Schmalstieg, and D. Kalkofen. Model-free authoring by demonstration of assembly instructions in augmented reality. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 28(11):3821–3831, 2022. doi: 10.1109/TVCG.2022.3203104 1, 2, 4

[44] O. Taheri, N. Ghorbani, M. J. Black, and D. Tzionas. GRAB: A dataset of whole-body human grasping of objects. In *European Conference on Computer Vision (ECCV)*, 2020. 8

[45] Y. Tian, L. Fan, P. Isola, H. Chang, and D. Krishnan. Stablerep: Synthetic images from text-to-image models make strong visual representation learners. *Advances in Neural Information Processing Systems*, 36, 2024. 5, 6

[46] X. Tu, Z. Yuan, B. Liu, J. Liu, Y. Hu, H. Hua, and L. Wei. An improved yolov5 for object detection in visible and thermal infrared images based on contrastive learning. *Frontiers in Physics*, 11:1193245, 2023. 5

[47] Unity Technologies. Unity Perception package. https://github.com/Unity-Technologies/com.unity.perception, 2020. 3

[48] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008. 7

[49] VDI-Handbuch Zuverlässigkeit. Human reliability – methods for quantitative assessment of human reliability, 11 2017. VDI 4006/F2. 2

[50] B. Wang, G. Wang, A. Sharf, Y. Li, F. Zhong, X. Qin, D. CohenOr, and B. Chen. Active assembly guidance with online video parsing. In *Proc. IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 459–466. IEEE, 2018. doi: 10.1109/VR.2018.8446602 1, 2

[51] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. In *Proc. Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7464–7475, June 2023. 3, 7

[52] C.-Y. Wang, I.-H. Yeh, and H.-Y. M. Liao. Yolov9: Learning what you want to learn using programmable gradient information. *arXiv preprint arXiv:2402.13616*, 2024. 5, 7

[53] X. Wang, T. Kwon, M. Rad, B. Pan, I. Chakraborty, S. Andrist, D. Bohus, A. Feniello, B. Tekin, F. V. Frujeri, et al. Holoassist: an egocentric human interaction dataset for interactive ai assistants in the real world. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 20270–20281, 2023. 3

[54] Z. Wang, P. Song, and M. Pauly. State of the art on computational design of assemblies with rigid parts. In *Computer graphics forum*, vol. 40, pp. 633–657. Wiley Online Library, 2021. 5

[55] O. Wasenmüller, M. Meyer, and D. Stricker. Augmented reality 3D discrepancy check in industrial applications. In *Proc. International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 125–134, 2016. doi: 10.1109/ISMAR.2016.15 1, 2

[56] K. D. Willis, P. K. Jayaraman, H. Chu, Y. Tian, Y. Li, D. Grandi, A. Sanghi, L. Tran, J. G. Lambourne, A. Solar-Lezama, et al. Joinable: Learning bottom-up assembly of parametric cad joints. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15849–15860, 2022. 5

[57] L.-C. Wu, I.-C. Lin, and M.-H. Tsai. Augmented reality instruction for object assembly based on markerless tracking. In *Proceedings ACM Symposium on Interactive 3D Graphics and Games*, pp. 95–102, 2016. doi: 10.1145/2856400.2856416 1, 2, 8

[58] M. Yamaguchi, S. Mori, P. Mohr, M. Tatzgern, A. Stanescu, H. Saito, and D. Kalkofen. Video-annotated augmented reality assembly tutorials. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pp. 1010–1022, 2020. doi: 10.1145/3379337.3415819 2

[59] B. Zhou and S. Güven. Fine-grained visual recognition in mobile augmented reality for technical support. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 26(12):3514–3523, 2020. doi: 10.1109/TVCG.2020.3023635 2

[60] Q.-Y. Zhou, J. Park, and V. Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018. 4

[61] Z. Zhou, J. Liu, D. T. Pham, W. Xu, F. J. Ramirez, C. Ji, and Q. Liu. Disassembly sequence planning: Recent developments and future trends. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 233(5):1450–1471, 2019. 3