ENPM808 - Independent Study
Final Report

Challenges with Object Handling in Industrial Environments

By
Kiran S Patil
kpatil27@umd.edu

Under the guidance of
Dr. Zeid Kootbally
zeidk@umd.edu

Abstract:
This project investigates the challenges associated with object handling by robots in industrial settings. In the context of the increasing use of automation in industrial settings, We focus on three key factors that hinder robust object detection and grasping: lighting conditions, occlusion, and best-grasping surface identification. A noteworthy challenge involves accurately identifying objects of interest, particularly when they are obstructed. Obstructions may arise from undesired components, human interference, or objects located in unfavorable positions. The primary objective of this independent study project is to design and implement an advanced deep-learning model capable of detecting and segmenting occluded objects in industrial environments. The focus will be on overcoming challenges posed by partial occlusion and improving the robustness of object recognition systems. Implementation is available at *https://github.com/kirangit27/Challenges-with-Object-Handling-in-Industrial-Environments.git*

Objects of interest:
This project employs parts from the ARIAC industrial environment, which consists of four different types of parts [Fig.1], in five colors each [Fig.2].

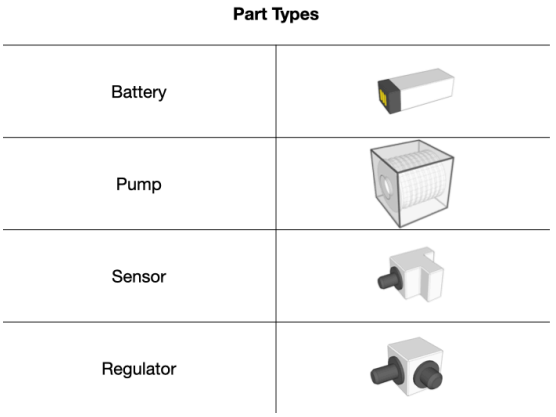

Fig. 1

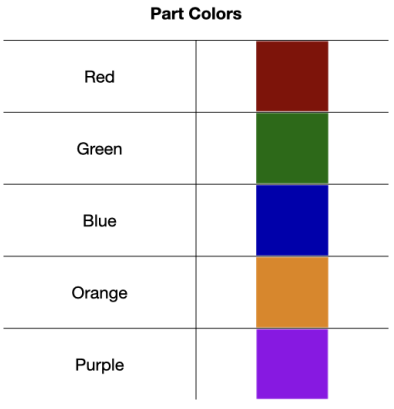

Fig. 2

1. Object detection

For the basic object detection task, the YOLOv8 object detection framework is employed. To train this deep learning model, the project utilizes a dataset that includes all available parts and their corresponding bounding box labels. This dataset is generated by creating a scene in Blender, adjusting lighting conditions and camera angles, and rendering the scene to produce synthetic images. The resulting dataset is then used to train the YOLOv8 model. Once trained, the model can accurately predict objects in scenes it has not encountered before.

1.1. Blender scene setup

To generate synthetic data for training the object recognizer, the Blender 3D software is employed to represent the objects of interest. A scene is meticulously crafted to closely resemble real-life environments where these objects are commonly found. This process involves setting up lighting conditions, camera angles, and object placements to create a scene that mirrors reality. By faithfully replicating real-world scenarios, the synthetic data generated becomes invaluable for training the object recognizer to accurately identify objects in various environments.

As mentioned earlier, the project relies on parts sourced from the ARIAC environment. To utilize these parts, the initial step involves installing the ARIAC 2024 package. Installation instructions for the package can be accessed here. Once the package is installed, the required .stl files can be found within the /ariac/ariac_gazebo/models folder. These .stl files for individual parts can be found under their respective folders.

Next, we proceed to launch Blender to initiate the import process for each .stl object earmarked for recognition. In the File window, we navigate to Import and opt for the Stl(.stl) option to begin importing the models as shown in [Fig.3]. It's crucial to specify a scale of 0.001 during the STL import process since Blender operates in meters. This ensures that the units of our model remain appropriately scaled within the Blender reference system, facilitating later stages where the camera's size is proportionate to the imported model.

.stl files are a specific type of file format used to represent the surface geometry of 3D models. Unlike some other 3D file formats, .stl files don't store information about color, texture, or other detailed characteristics of a model. They focus purely on describing the outer shell of the 3D object using triangles.

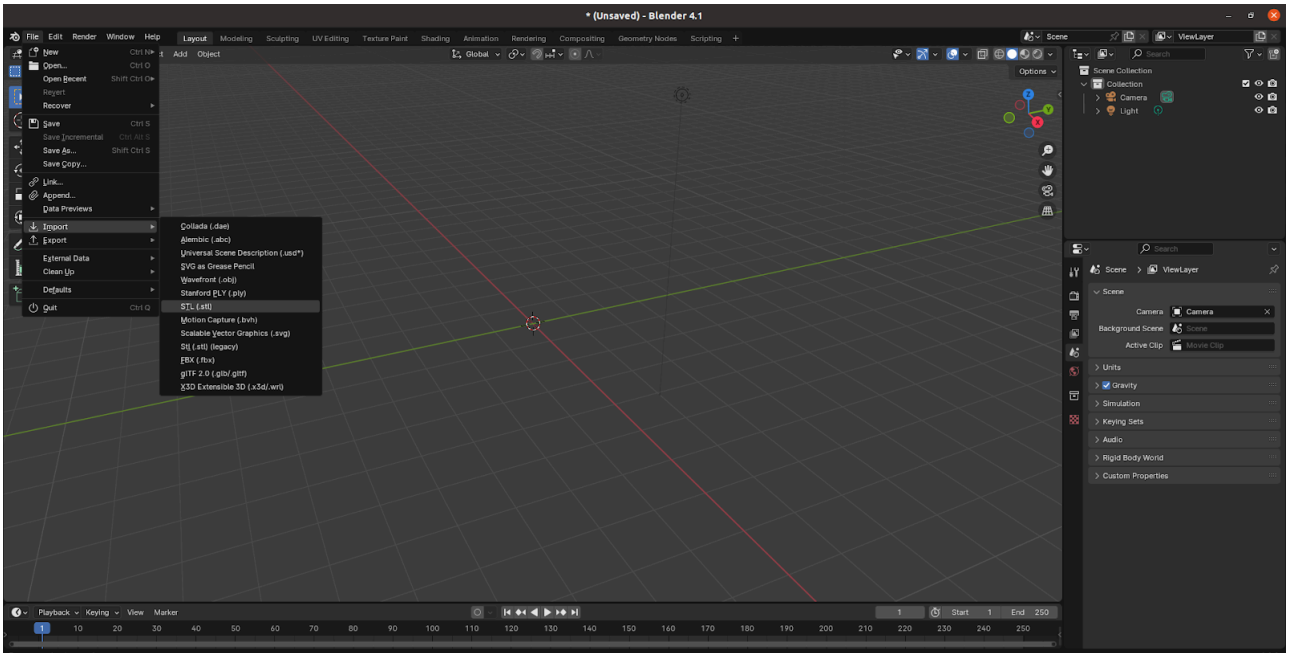

Fig. 3

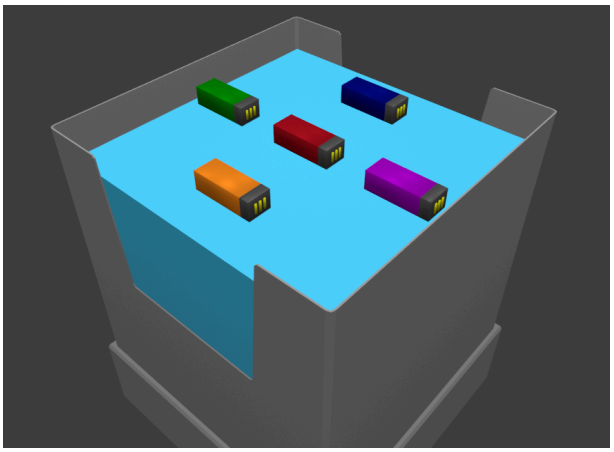

Fig. 4 (a)

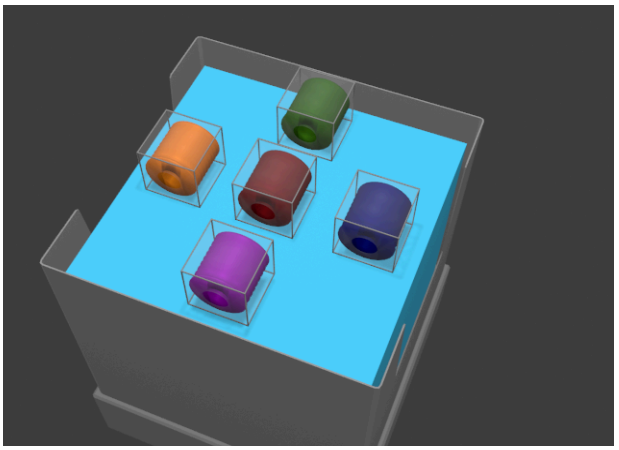

Fig. 4 (b)

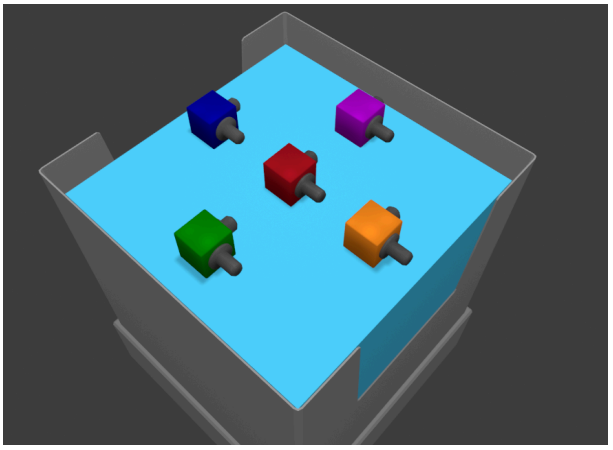

Fig. 4 (c)

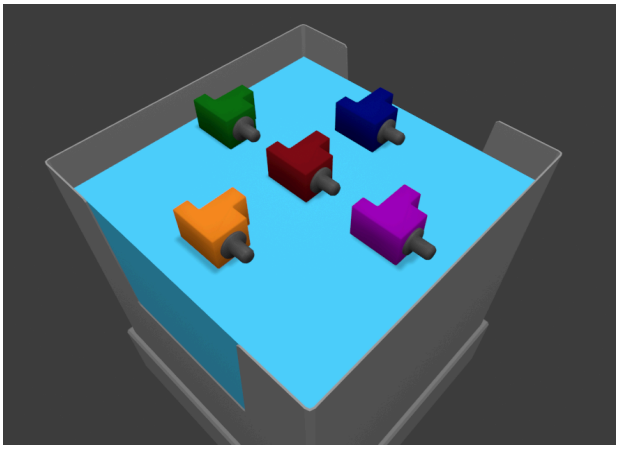

Fig. 4 (d)

Once each object is imported into the Blender environment, it's essential to create a plane to represent the surface of the scene. Additionally, ensure to name every object in the Scene Collection menu, accessible from the right panel. Moving on to defining the entire scene, realism becomes paramount for enhancing the quality of our training data and optimizing our algorithm's object recognition capabilities.

To achieve this, we begin by defining the materials of each object. This involves selecting each object and accessing the Material option in the right panel. Here, the Base Color, Specular, and Roughness parameters play pivotal roles in fine-tuning the appearance of the objects. Adjustments to Specular and Roughness determine whether the object absorbs or reflects light and whether it appears shiny or matte. If required, textures can be added to the objects in the scene. These textures, imported from images, can impose the Base Color from the image onto the objects. In our project, textures were applied to the Bin holding the objects, further enhancing the scene's realism. [Fig.4] shows the end result obtained for the (a) Battery, (b) Pump, (c) regulator, and (d) sensor respectively.

Following the material definition, the lighting setup is crucial. The second light can be effortlessly created by duplicating the first light, which is automatically generated when the Blender scene is initiated. Similar to objects, it's important to assign names to these lights, such as "light1" and "light2," for ease of reference during scripting.

The primary parameter to adjust for these lights is the Power parameter, enabling manipulation of light intensity during the scripting phase. In the "Light Properties" panel, located at the bottom right in the provided picture, the Power parameter is visible, set to 16W. In the final scene, four spotlights are strategically positioned to focus on the objects within the bin. Additionally, a single point light is positioned directly above the bin, providing illumination from above. [Fig.5] shows the final lighting setup.

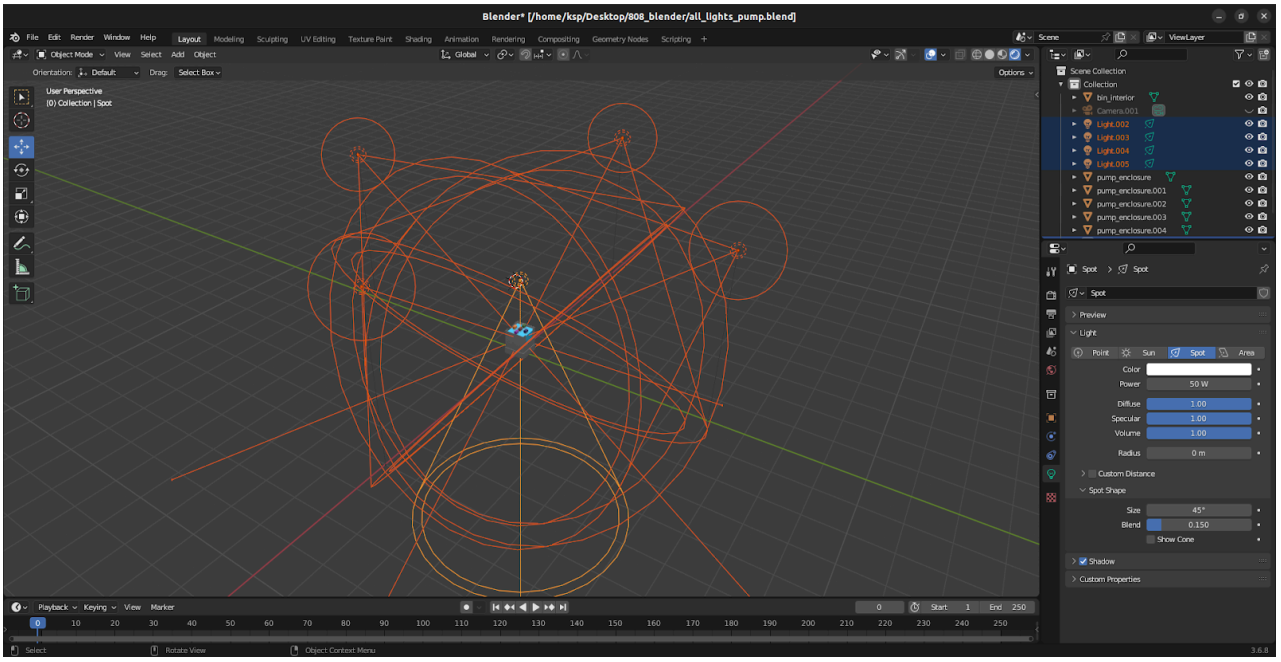

Fig. 5

In the final step of setting up the Blender scene for scripting, attention turns to configuring the camera. This step holds significant importance as it enables seamless control of the camera's movement through scripting, facilitating the capturing of images from various angles. To achieve this, a method of camera movement akin to orbiting around the objects was devised.

Considering the complexities of defining a list of (x, y, z) points to describe spherical movement, a simpler approach was adopted. An axis was established at the center of the scene, serving as a fixed point to which the camera would adhere when moved. This setup allows for rotational movement around the axis, akin to rotating your fist around your elbow. Just as the fist revolves around the elbow, the camera orbits around the axis, simplifying the process of capturing images from different perspectives, as shown in [Fig.6].

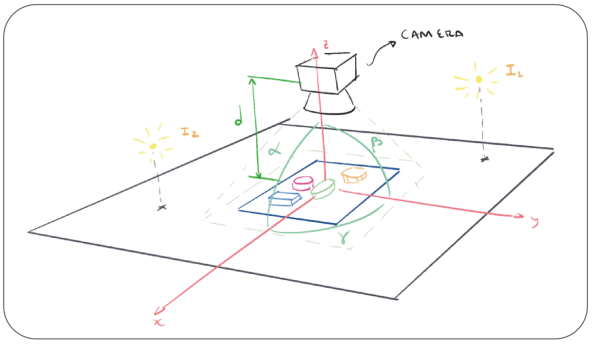

Fig. 6

To establish this relationship, the camera is selected first, followed by the axis, while holding the Shift key. Subsequently, pressing Ctrl + P and selecting "Object (Keep Transform)" sets the axis as the parent of the camera.

Verifying this setup involves selecting the axis, entering the Camera view, and adjusting the rotation coordinates of the axis in the Transform window. As the axis rotates, the camera orbits around the objects, demonstrating successful coordination between the two entities.

1.2. Synthetic Dataset generation with Blender Scripting

With the complete scene setup, scripting in Blender becomes the next focus. This functionality empowers the automation of render generation, enabling the creation of tens of thousands of images and labels, essential for training our object recognition algorithm.

Accessing the scripting functionality involves navigating to the Scripting window. Here, three key elements are presented: the Blender Console, the Scripting environment, and the Command Tracker. The Command Tracker monitors the scripting commands executed when modifying parameters in the Blender environment. Within the Scripting environment, Python scripts can be imported, created, and saved to an external location. Once the script is prepared, it can

be executed by clicking the "Run Script" button. Finally, to monitor the output of the script, toggle the System Console by navigating to Window/Toggle System Console. This console displays the output of the code execution, providing insights into the script's functionality.

Accessing object information is crucial for modifying various parameters such as light brightness and camera position. bpy.data.objects serve as a collection of all scene objects, with two methods to call objects: using indices (bpy.data.objects[x]) or object names (bpy.data.objects['Name']).

For instance, light1.data.energy allows adjustment of light brightness, camera.location modifies the camera position and axe.rotation_euler alters the rotation of the 'axe' object. To generate a large dataset for algorithm training, an algorithm is devised to capture images of objects from different angles by moving the camera around the scene. Light brightness is also varied to create a more realistic dataset.

The algorithm comprises three loops, each adjusting a camera angle, while simultaneously modifying light brightness.

```
## Define angle and height ranges
camera_d_limits = [dmin, dmax]
beta_limits = [min_beta, max_beta]
gamma_limits = [min_gamma, max_gamma]

## Initialize counters
render_counter = 0
rotation_step = rot_step

## Initialize camera
set_axis.rotation = (0,0,0)
set_camera.location = (0,0,0)

## Run algorithm
for d in range(dmin, dmax+1, 2) do:
   set_camera.location = (0,0,d)

   for beta in range(min_beta, max_beta+1,rotation_step) do:

      for gamma in range(min_gamma, max_gamma,rotation_step) do:
         render_counter = render_counter +1
         ## Update camera rotation
         axis_rotation = (beta, 0, gamma)
         set_axis.rotation = axis_rotation

         ## Configure lighting
         energy1 = random.randint(0,30)
         set_l_1.energy = energy1
         energy2 = random.randint(4,20)
         set_l_2.energy = energy2

         ## Generate render
         render_blender(render_counter)

         ## Output Labels
         text_file_name = labels_filepath+'/'+str(render_counter)+'.txt'
         text_coordinates = get_all_object_coordinates()
         text_file.write(text_coordinates)
```

For each camera position, an image of the scene is captured, and object information is recorded in a text file.

The algorithm described above generates multiple images paired with corresponding text files containing object locations and bounding box information. This dataset will be utilized for training a deep learning algorithm. To implement a program that incorporates this algorithm and accesses scene and object information, we've created a class named Render. The initialization of the Render class begins with importing relevant libraries, particularly bpy, which grants access to Blender elements.

The main_rendering_loop() function within the Render class executes the algorithm in Python. By accessing information about the camera, axis, and lights, this function moves the camera around the objects, captures images, and extracts labels. The self.calculate_n_renders(rot_step) function calculates the number of renders and labels based on the specified rotation step. This allows for adjustment of the rotation step to achieve the desired number of renders. The get_all_coordinates(resx, resy) function iterates through all objects in the scene, attempting to retrieve each object's coordinates using the self.find_bounding_box(obj) function, which identifies the bounding box coordinates of the object if it's within the camera's view.

1.3. YOLOv8 - Object Detection Model

YOLOv8 is a state-of-the-art deep learning model designed for real-time object detection. It achieves a balance between accuracy and speed, making it suitable for various computer vision tasks.

YOLOv8 Architecture [Fig.7] consists of two main parts: Backbone and Head.

Backbone: This is a modified version of the CSPDarknet53 architecture. It extracts features from the input image using convolutional layers. Notably, it employs cross-stage partial connections to improve information flow.

Head: This part is responsible for making predictions. It uses convolutional layers followed by fully connected layers to predict bounding boxes, object confidence scores, and class probabilities for detected objects.

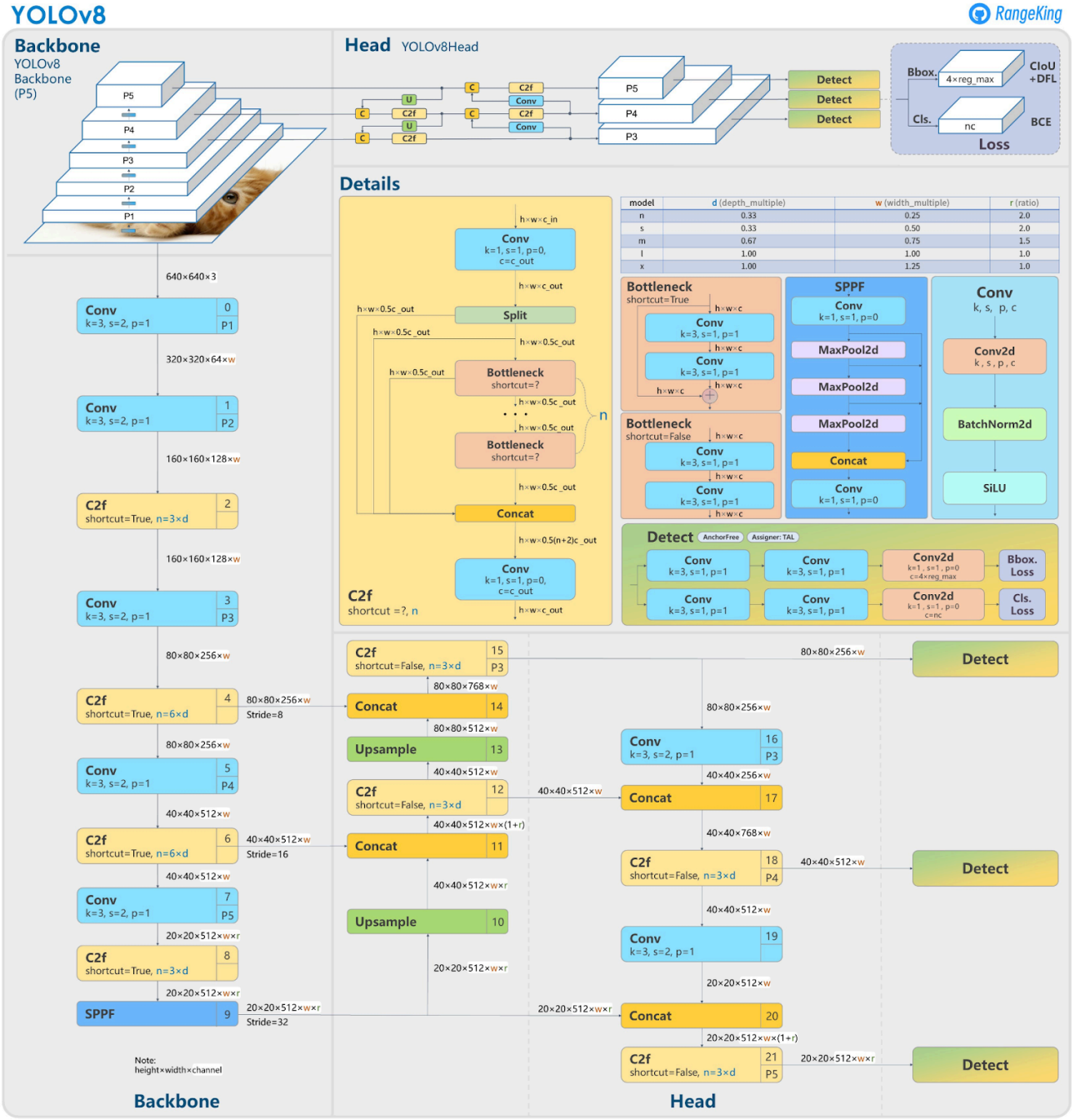

Fig. 7

C2f: CSP (Cross Stage Partial) bottleneck with 2 convolution layers. The C2f block refines and merges information from the raw features extracted by the backbone of the network. It essentially acts as a feature manipulation module before feeding the data to the head for final object detection predictions. Internally, it likely employs multiple convolutional layers, with techniques like BatchNorm and SiLU activation for normalization and non-linearity. By effectively merging features from different scales, C2f helps the model learn a richer representation of objects in the image. Previous versions had different feature scales for local and global features.

Anchor-free Detection:

Anchor-free detection is a technique used in object detection models to predict bounding boxes for objects directly, without relying on predefined anchors. Most traditional object detection models, like earlier versions of YOLO (pre-v8) and Faster R-CNN, employ predefined anchor boxes. These anchors represent

boxes of various sizes and aspect ratios placed across an image.

Anchor-free detectors eliminate the need for predefined anchors. Instead, they directly predict bounding boxes for objects. This prediction typically involves:

- Object center coordinates: The model predicts the x and y coordinates of the object's center point relative to the image grid.
- Object dimensions: The model predicts the width and height of the bounding box.
- Confidence score: The model outputs a score indicating the probability of an object being present at the predicted location.

IOU - Intersection over Union:

IOU is a ratio between the area of intersection (overlap) between the predicted box and the ground truth box, divided by the area of their union (combined area). Ranges from 0 to 1. It essentially measures the accuracy of the model's localization predictions.

In practice, a specific IOU threshold is often set to determine whether a prediction is considered a true positive (correct detection). Limitations: IOU doesn't account for certain factors like object orientation or keypoint localization. It purely focuses on the bounding box overlap.

Loss Functions:

- **Bbox Loss:** It quantifies the difference between the predicted bounding boxes and the corresponding ground truth bounding boxes for objects in an image. This loss value guides the model during training to refine its predictions and improve its ability to localize objects accurately.
  Generalized Intersection over Union (GIOU Loss): This loss function not only considers the area of overlap (intersection) between predicted and ground truth boxes (like IOU) but also incorporates penalties for boxes with large areas of non-overlap or incorrect aspect ratios. It encourages the model to predict boxes that are closer in size and shape to the ground truth boxes.
- **Cls Loss:** It measures the discrepancy between the model's predicted class probabilities for objects and the actual object classes present in the image (ground truth). It guides the model during training to learn and distinguish between different object classes effectively. A loss function, typically the cross-entropy loss, is used to quantify the difference between the predicted probabilities and the ground truth labels. This loss value helps the model understand how far off its predictions are from the correct classes.
- **Cross-Entropy Loss:** It penalizes the model more for significant misclassifications and less for minor errors. It works by comparing the predicted probability distribution (how likely the model thinks each class is) with the one-hot encoded ground truth label (only one class is truly present).

1.4. Results Obtained

The following image [Fig.8] shows the number of instances each class has appeared in the training dataset, followed by the average occurrence and (height, width) of their bounding boxes.

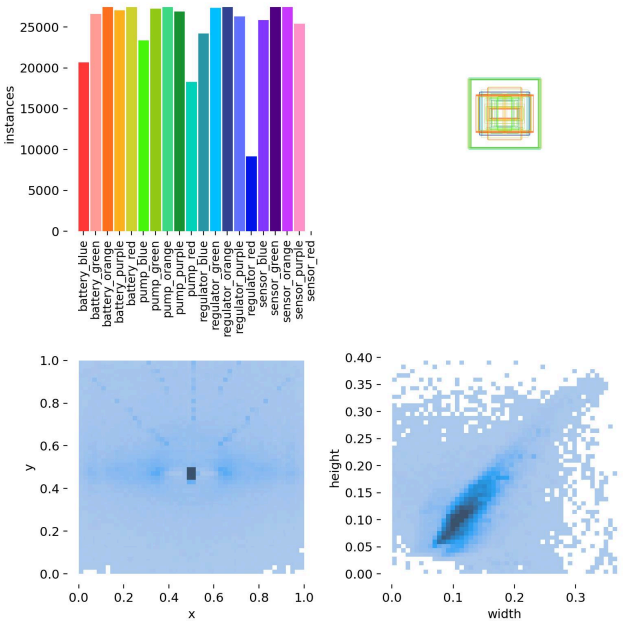

Fig. 8

1.4.1 F1 Confidence Curve: [Fig.9] is a visualization tool used to understand the performance of a YOLOv8 model, particularly in object detection. It helps analyze the trade-off between two crucial metrics: precision and recall, across different confidence thresholds.

- **F1 Score:** This metric combines precision (percentage of correctly identified objects) and recall (percentage of all actual objects identified) into a single score.
- **Confidence Threshold:** YOLOv8 assigns a confidence score to each detection it makes. This score represents the model's certainty about the detection being a true object. The F1 Confidence Curve analyzes the F1 score at various confidence thresholds.
- **Trade-off between Precision and Recall:** As you increase the confidence threshold, you ensure a higher percentage of detections are true positives

(precision goes up), but you might miss some actual objects (recall goes down). The curve depicts this interplay.

The curve helps you identify the confidence threshold that offers the best balance between precision and recall for your specific application. By analyzing the curve for specific object classes, you can identify classes where the model struggles and focus training efforts on those. The model attains a peak F1 score of 0.97 at a confidence of 0.410 (IOU - 41%).

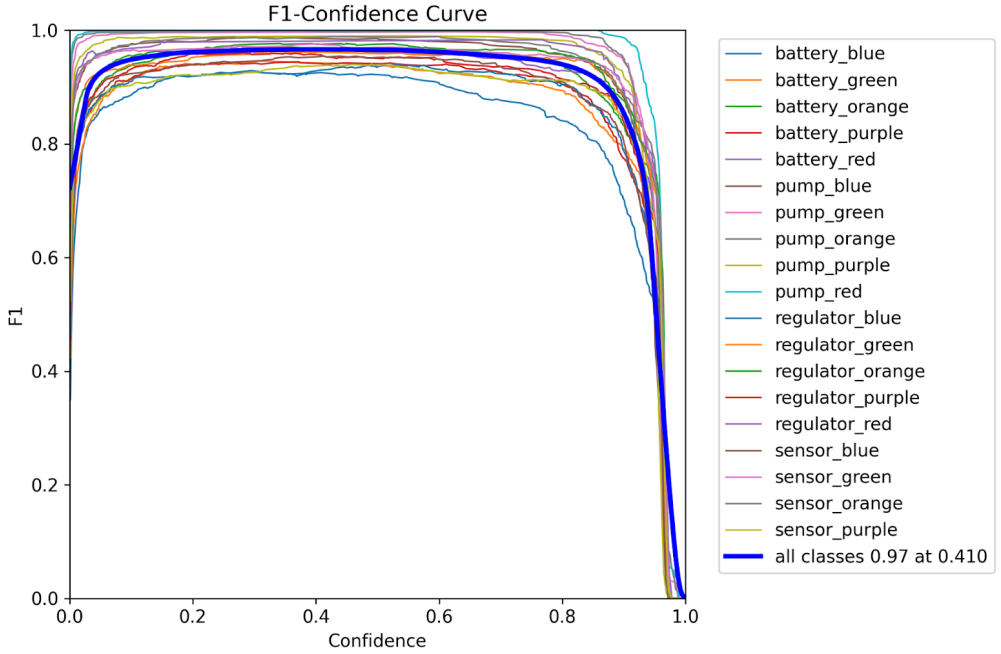

Fig. 9

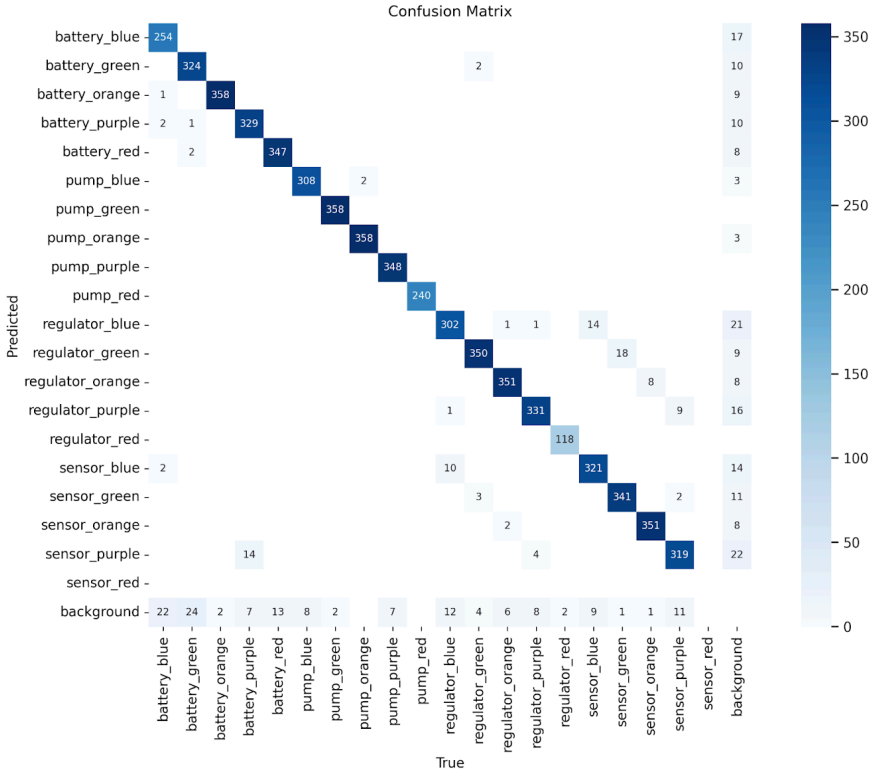

Fig. 10

1.4.2 Confusion Matrix: [Fig.10] is a table that visually summarizes the performance of a model on a dataset. It shows how many times the model correctly classified objects (true positives, true negatives) and how often it made mistakes (false positives, false negatives).

- **Focus on Bounding Boxes:** YOLOv8's confusion matrix deals with bounding boxes, not pixel-level masks. It evaluates how well the model predicts bounding boxes around objects.
- **Thresholds for Classification:** YOLOv8 employs two thresholds (confidence and Intersection over Union (IoU)) to categorize predictions into the confusion matrix.
  Confidence: This score indicates the model's certainty that a bounding box contains an object.
  IoU: This metric measures the overlap between the predicted bounding box and the ground truth (actual) bounding box.

Deriving the Confusion Matrix

YOLOv8's confusion matrix generation process involves these steps;

- **Ground Truth and Predictions:** The model's predictions (bounding boxes with confidence scores) are compared to the ground truth annotations (actual object locations) for the image.
- **Matching Predictions:** Predictions are matched to ground truth annotations based on the IoU metric. A prediction is considered a match if its IoU with a ground truth box exceeds a certain threshold.
- **Classification based on Thresholds:** Predictions are then classified into the confusion matrix categories using the confidence threshold:

True Positive (TP): A predicted bounding box with high enough confidence (above the threshold) correctly matches a ground truth box.

True Negative (TN): A prediction with high enough confidence correctly identifies the absence of an object in a region.

False Positive (FP): A predicted bounding box with high enough confidence doesn't correspond to any actual object (it's a mistake).

False Negative (FN): The model misses an object that exists in the image (no prediction with sufficient confidence for that object).

1.4.3 Loss Curves: [Fig.11] consists of six loss curves. A loss curve is a graph that tracks the model's error (or loss) as it learns from training data. The goal is for the loss value to decrease over time, indicating the model is making fewer errors in its predictions.

There are typically two types of loss curves in YOLOv8's results;

Training Loss Curves show the model's loss on the data it's specifically being trained on.

- **train/box_loss:** This curve represents the bounding box localization loss during training. Lower values indicate that the model is better at predicting the accurate bounding boxes for detected objects.
- **train/cls_loss:** This curve depicts the classification loss during training. Lower values signify that the model is better at distinguishing between different object classes.

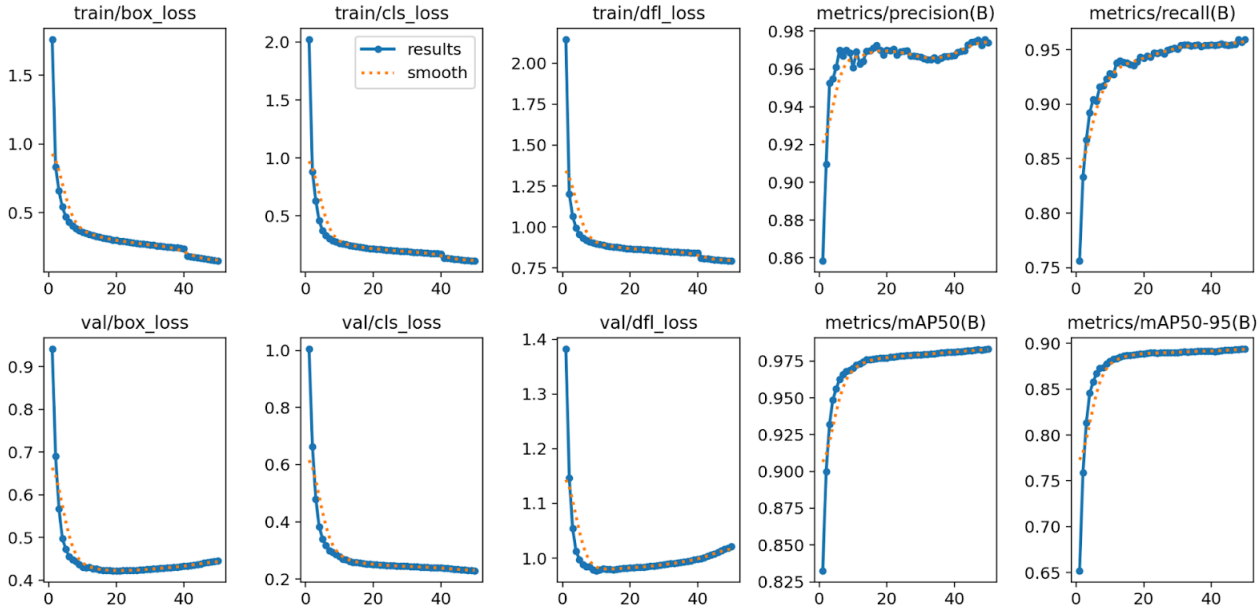

Fig. 11

- **train/dfl_loss:** This curve stands for the deformation loss during training. It measures how well the model predicts the deformation (perspective distortion) of objects in the training images. Lower values suggest better handling of object deformations.
- **val/box_loss:** This curve reflects the bounding box localization loss on the validation set. It helps assess how well the model generalizes to unseen data.
- **val/cls_loss:** This curve shows the classification loss on the validation set. It indicates how well the model performs on classifying objects in data it hasn't been explicitly trained on.
- **val/dfl_loss:** This curve represents the deformation loss on the validation set. It evaluates the model's ability to generalize to unseen object deformations.

1.4.4 Metric Curves: The remaining four curves in [Fig.11] are the matric curves. This curve represents the precision for the "box" metric, which refers to bounding boxes. Precision measures the accuracy of your model's predictions. Essentially, it tells you how many of the objects the model detected actually belong to the correct class. Higher values on this curve indicate better precision. In simpler terms, the model is making fewer false positives (detecting objects that aren't actually there or assigning them the wrong class).

- **metrics/precision(B):** This curve represents the precision for the "box" metric (bounding boxes). Precision measures how many of the predicted objects actually belong to the correct class. Higher values indicate better precision.
- **metrics/recall(B):** This curve depicts the recall for the "box" metric. Recall measures how many of the actual objects in the image were correctly detected by the model. Higher values indicate better recall.
- **metrics/mAP50(B):** This curve shows the Mean Average Precision (mAP) at an Intersection over Union (IoU) threshold of 0.5. mAP is a comprehensive metric that considers both precision and recall across different IoU thresholds. Higher values suggest better overall object detection performance.
- **metrics/mAP50-95(B):** This curve represents the mAP averaged across IoU thresholds ranging from 0.5 to 0.95. It provides a broader picture of the model's performance across varying degrees of overlap between predicted and ground truth bounding boxes. Higher values indicate more consistent detection accuracy at different levels of overlap.

[Fig.12] shows the (a) Ground truth labels for a batch of images in the validation set and their respective (b) Prediction labels.

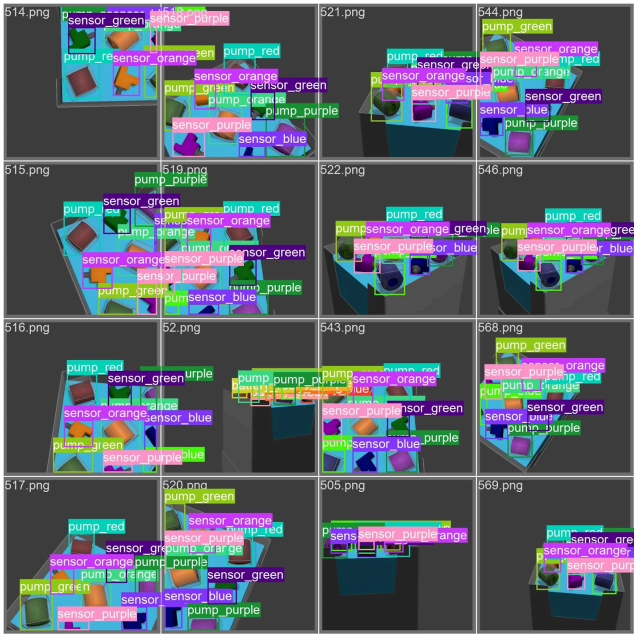

Fig. 12 (a)

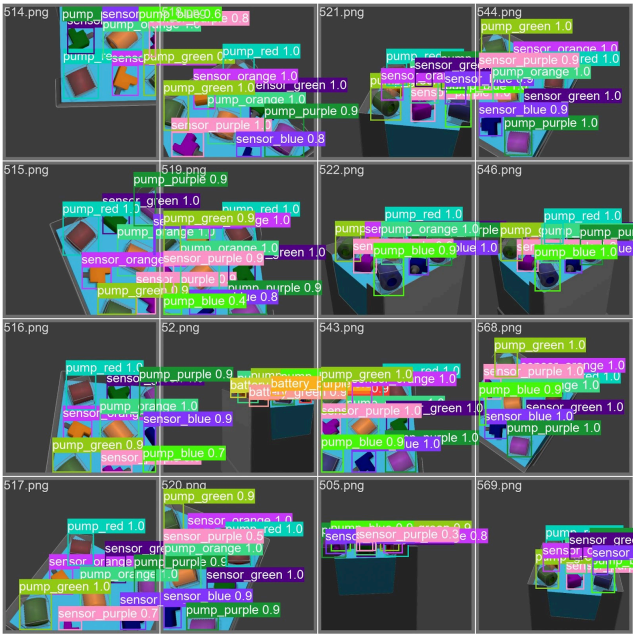

Fig. 12 (b)

2. Object detection - With Occlusions

Occlusion refers to the phenomenon where an object of interest is partially or fully obscured by another object in the scene. In object detection, occlusion poses significant challenges as it can hinder the accurate localization and recognition of objects, leading to errors in automated systems, particularly in industrial automation settings.

Here are some specific effects of occlusion in object detection and how they can hinder industrial automation:

- **Incomplete Object Detection**: Occlusion can cause only parts of an object to be visible, making it challenging for object detection algorithms to accurately detect and classify the object. This can lead to incomplete or inaccurate detection results, impacting the reliability of automated systems in industrial settings.

- **Misclassification:** Partially occluded objects may be misclassified or confused with other objects in the scene due to the limited information available to the detection algorithm. This can result in incorrect decisions being made by automated systems, affecting the overall performance and efficiency of industrial processes.

- **Tracking Errors:** Occlusion can disrupt object tracking algorithms, especially in scenarios where objects move dynamically within the scene. When objects become occluded, their trajectory may be lost or incorrectly predicted, leading to tracking errors and potentially causing collisions.

- **Increased False Positives/Negatives:** Occlusion can increase the likelihood of false positives (incorrectly detected objects) or false negatives (missed detections) in object detection systems. False positives can lead to unnecessary interventions by automated systems, while false negatives can result in critical objects being overlooked, both of which can disrupt industrial processes and compromise safety/efficiency.

- **Reduced Robustness:** Occlusion introduces additional variability and complexity into the detection task, reducing the robustness of object detection algorithms to handle diverse real-world scenarios. This can necessitate more sophisticated and computationally intensive algorithms or the integration of additional sensor modalities to mitigate the effects of occlusion, increasing the cost and complexity.

Overall, occlusion poses significant challenges for object detection in industrial automation, impacting the reliability, accuracy, and efficiency of automated systems. Addressing these challenges requires the development of robust detection algorithms capable of handling occluded objects effectively, as well as the integration of complementary sensing and tracking technologies to enhance the performance of industrial automation systems in complex environments.

2.1 Dataset generation

In the context of object detection with occlusions, the dataset utilized for basic object detection in section 1 remains unchanged. However, a new dataset is employed for validation, generated in Blender. This new dataset comprises scenes containing occluded objects. The pipeline for dataset generation and model training remains consistent with the previous section. [Fig.13] depicts a Blender scene featuring occluded objects, which served as the basis for validating the trained object detection model.

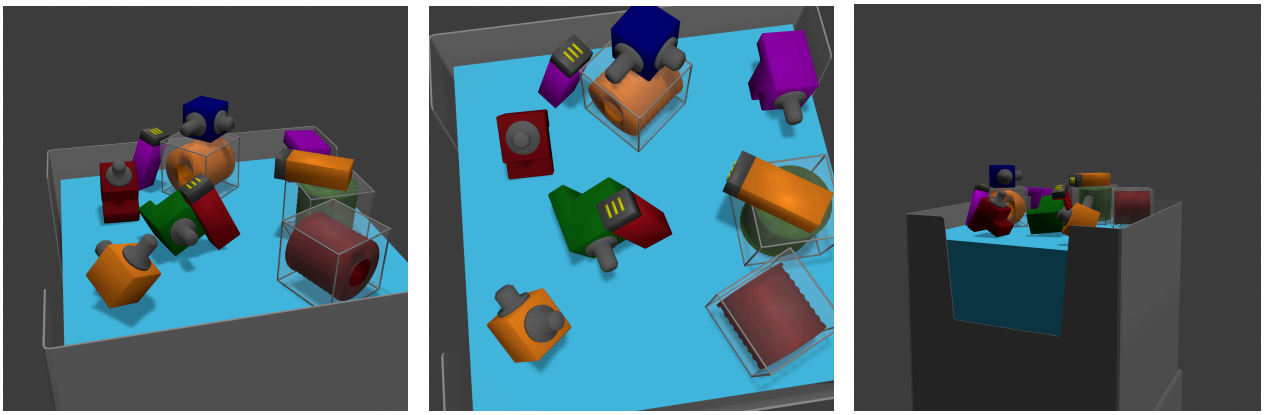

Fig. 13

2.2 Results - Comparison between Occlusion and Occlusion-Free Object Detection

The subsequent subsection focuses on comparing the results obtained between occlusion-free object detection and detection under occluded conditions.

2.2.1 F1 Confidence Curve: [Fig.14] (a) 5 epochs (b) 50 epochs illustrates a significant drop in the peak F1 score for occluded scenes compared to [Fig.9]. Specifically, the F1 score decreases from 0.97 to 0.63. In contrast to the previous section, where the F1 scores for all classes ranged from 0.8 to 1.0, the variability spectrum for occlusion scenarios is notably broader, spanning from 0.2 to 0.9.

2.2.2 Confusion Matrix: In [Fig.15], the confusion matrix illustrates the performance of the object detection model in an occluded scene. Similar to the observed drop in the F1 score, a decline in model performance is evident here too. The matrix reveals numerous misclassifications, with many objects erroneously categorized as background. Contrasting this with [Fig.10], where the majority of values align along the diagonal representing accurate classification, in [Fig.15], values appear more scattered, indicating a higher incidence of misclassifications.

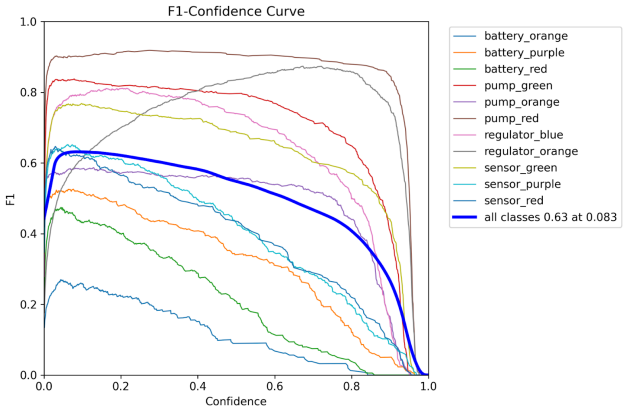

Fig. 14 (a)

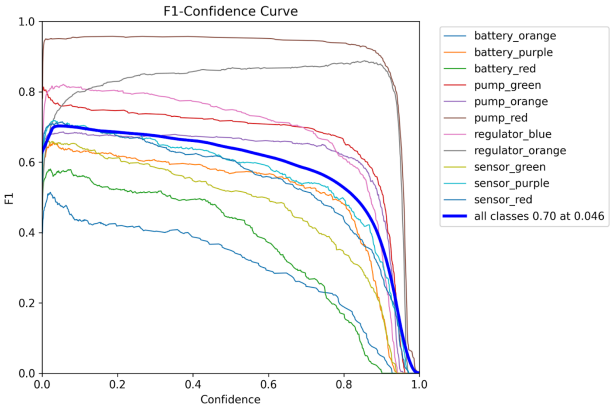

Fig. 14 (b)

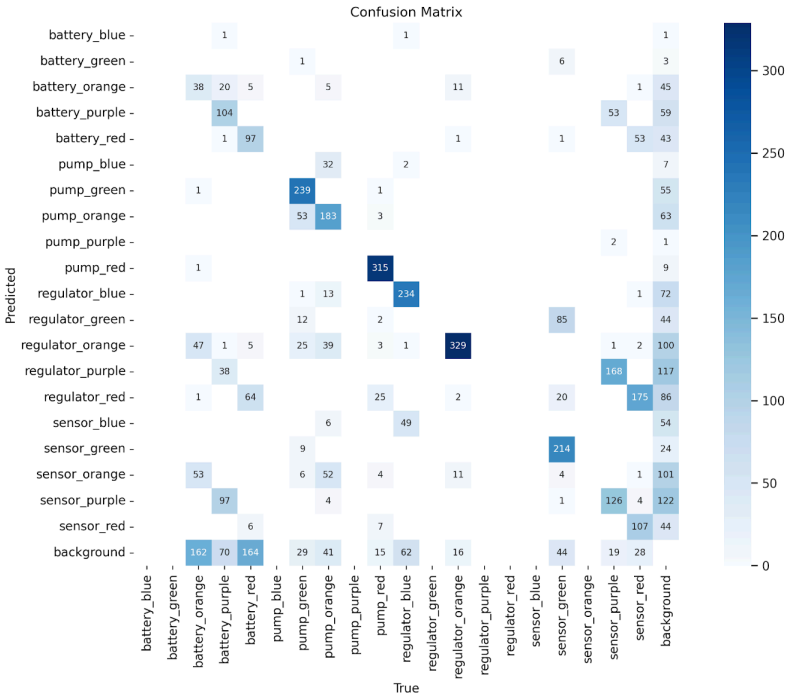

Fig. 15

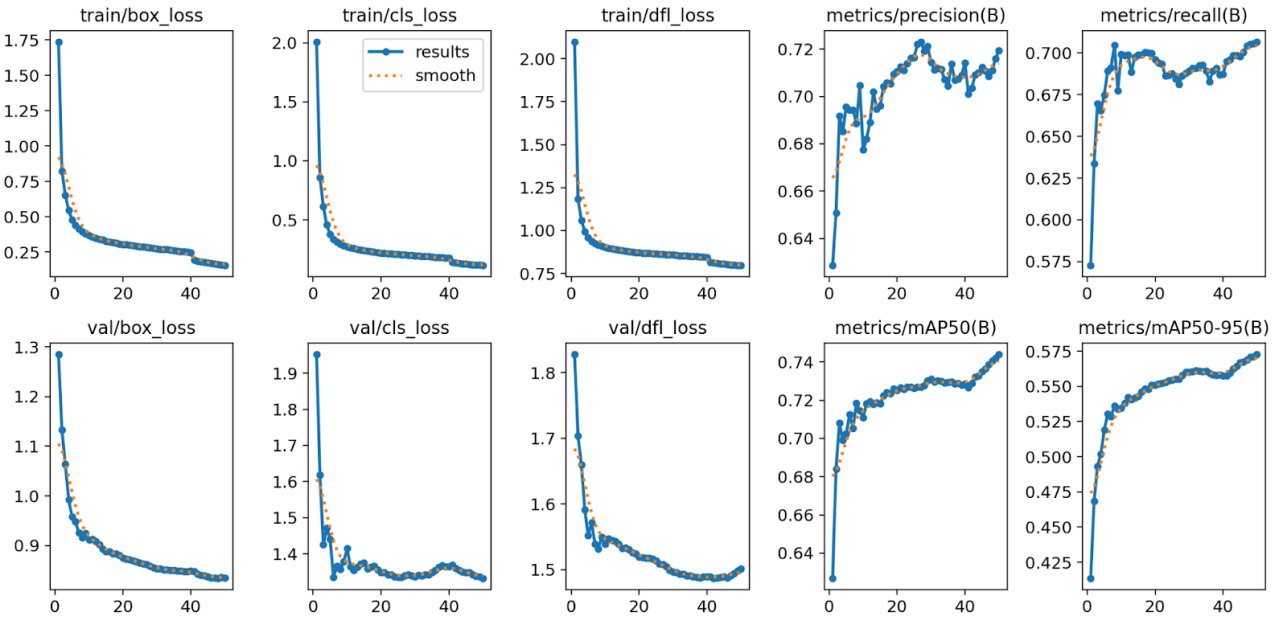

Fig. 16

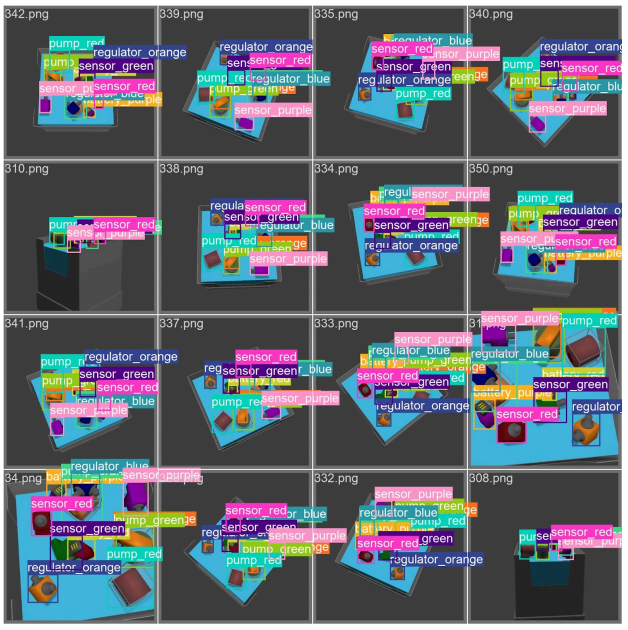

Fig. 17 (a)

Fig. 17 (b)

2.2.3 Loss/Matric Curves: The graphs present in [Fig.16] represent the Loss and Metric curves for Object detection with the presence of occlusion in the scene. A direct comparison can be drawn with [Fig.11].

In [Fig.11], it can be observed that the loss curves, irrespective of the type, drop rapidly within the first 10 epochs itself, but in the case of occlusion [Fig.16], the loss curves tend to drop much slower and take around 20-plus epochs to attain a stable curve. Also, the minimum loss attended is good but not as great as in [Fig.11] as expected.

Even for the metric curves, the peak precision/recall has dropped significantly from around 0.9 in [Fig.11], to around 0.6 in [Fig.16].

[Fig.17] shows the (a) Ground truth labels for a batch of images in the validation set and their respective (b) Prediction labels.

3. Best Grassping Point

Determining the best grasping point for an object in an industrial environment can indeed be challenging, especially when the object is not in an ideal position. Here are some common challenges;

- **Variability in Object Positioning:** The inconsistent placement of objects in industrial environments can lead to difficulties in predicting their exact location. This variability hampers automation efforts as robotic systems rely on precise positioning for effective grasping and manipulation tasks.

- **Complex Object Shapes:** Industrial objects often exhibit intricate shapes and contours, posing a challenge for automated systems to identify suitable grasping points. Without a clear understanding of the object's geometry, robots may struggle to grasp them securely, resulting in inefficiencies or errors in production processes.

- **Cluttered Environments:** The presence of obstacles and clutter in industrial settings complicates the task of locating and accessing objects for manipulation. Automated systems may encounter difficulties in navigating through cluttered spaces, leading to delays or failures in grasping tasks.

- **Sensor Limitations:** Traditional sensors may suffer from inaccuracies or limitations in certain industrial environments, such as low visibility or harsh operating conditions. This can impede the ability of robotic systems to gather reliable information about the object's position and orientation, hindering their grasping capabilities.

- **Dynamic Environment:** Industrial environments are often dynamic, with objects constantly being moved or replaced as part of manufacturing processes. This dynamic nature introduces uncertainty into automation systems, making it challenging to adapt and respond effectively to changes in object locations and conditions.

In the automation industry, these challenges can collectively impede the development and deployment of robotic systems for grasping and manipulation tasks. Without robust solutions to address these issues, automation efforts may struggle to achieve the level of reliability and efficiency required for widespread adoption in industrial settings.

3.1 Normal Vector Analysis:

This section explores a potential method for identifying the optimal grasping point for objects in a cluttered environment. It utilizes normal vector analysis to ascertain the flattest surface of the object relative to the world frame. Here's a brief explanation:

3.1.1 Surface Normal Analysis:

- Every point on a surface possesses a normal vector, perpendicular to the surface at that specific point. Visualize these vectors as arrows emanating directly away from each point [Fig.18].
- These normal vectors furnish information about the surface's orientation, indicating the direction that is 'up' or 'out' from each point.

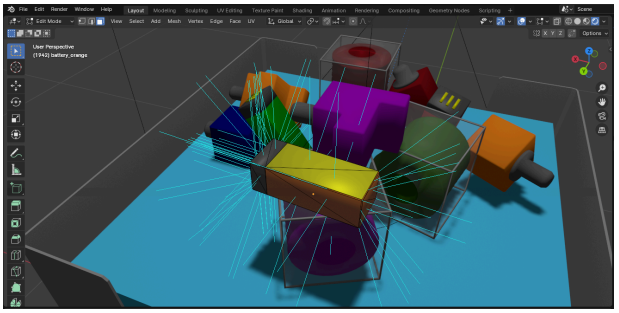

Fig. 18

3.1.2 Assessing Surface Flatness

- To gauge the flatness of a surface, we examine the uniformity of the normal vectors across it.
- A flat surface will exhibit consistent normal vectors, all pointing in the same direction. Conversely, a non-flat surface will showcase normal vectors pointing diversely, suggesting curvature or irregularity.

3.1.3 Comparison with Gravity:

- Gravity serves as a constant reference direction, typically directed downward, perpendicular to the ground.
- By contrasting the orientation of surface normals with the opposite of the gravity vector, the flattest surface [Fig.19] can be pinpointed.
- This comparison aids in identifying which surface is most adept at supporting weight or countering gravitational forces.

3.1.4 Determining the Grasping Point:

- Upon identifying the flattest surface, the task is to locate the optimal point for grasping the object.
- The preferred grasping point is typically positioned at the center of mass or centroid of the object. This choice ensures the most balanced weight distribution when handling or lifting the object.
- Placing the grasp at or near the center of mass minimizes the risk of tipping or instability during manipulation.

In essence, analyzing an object's surface normals enables the evaluation of its flatness and the determination of the most secure grasping point, aligning with the direction of gravity. This method ensures that the object can be safely lifted and handled with minimal risk of imbalance or instability.

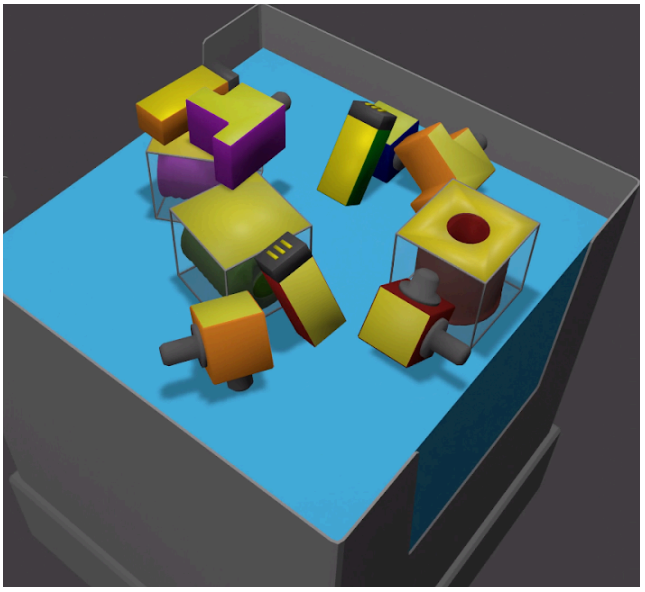

Fig. 19 (a)

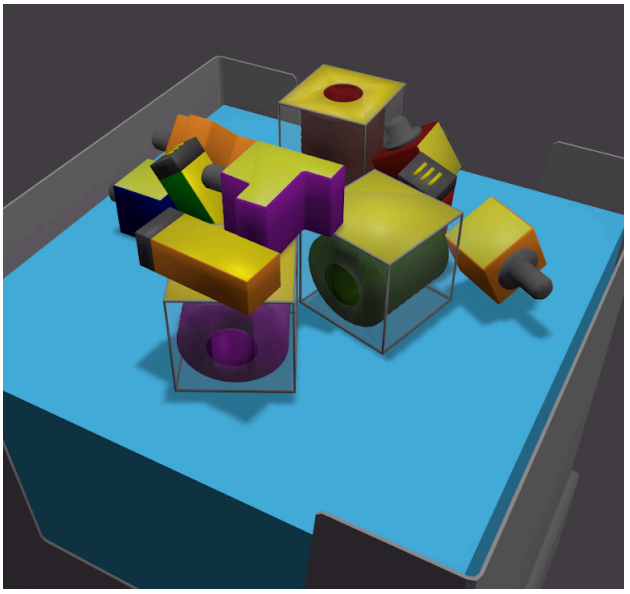

Fig. 19 (b)

4. Deploying the model into the ARIAC environment.

To integrate the trained YOLOv8 model into the ARIAC industrial Gazebo environment, the first step involves configuring the camera sensors within the environment to capture images. More about sensor configuration can be read here. In the ARIAC environment, nine RGBD camera sensors are utilized for the object detection task. Unlike the process used in Blender dataset generation, where a single camera could be moved around the bin, in this scenario, the cameras are positioned in a circular fashion around the bin [Fig.20]. This positioning enables capturing images from various angles without the need for camera movement.

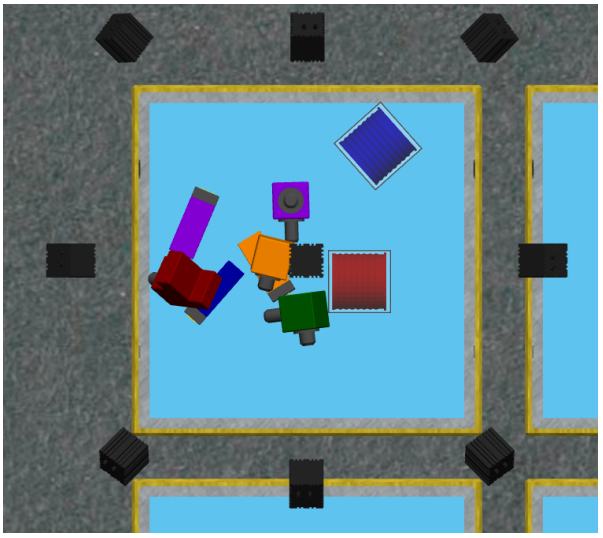

Fig. 20

4.1 Processing Data from an RGBD Camera

The RBG and RGBD cameras in the ARIAC environment publish data on various topics via plugins. One such topic, containing the image matrix, is located at */ariac/sensors/<SENSOR_NAME>/rgb_image*. More details on camera data processing can be found here.

These camera images are formatted as ROS messages, which are incompatible with OpenCV Matrix types. To bridge this gap, the ROS2 *cv_bridge* package facilitates efficient conversion between image messages and OpenCV matrices. The most recent image from each subscribed sensor is stored in an instance variable using the *imgmsg_to_cv2* function. It's crucial to specify the image encoding, as the default format used by RGB cameras is rgb8, while bgr8 is the canonical OpenCV format.

Once the Image data is subscribed from the respective camera nodes, these encoded images can be used for prediction with the trained YOLOv8 model.

4.2 Results

4.2.1 Object Detection without Occlusion

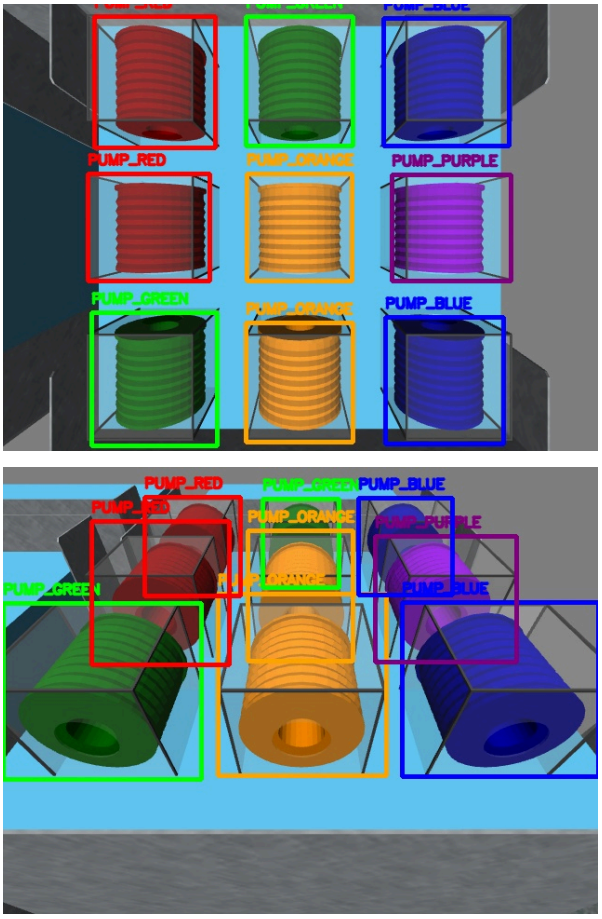

Fig. 21

4.2.2 Object Detection with Occlusion -

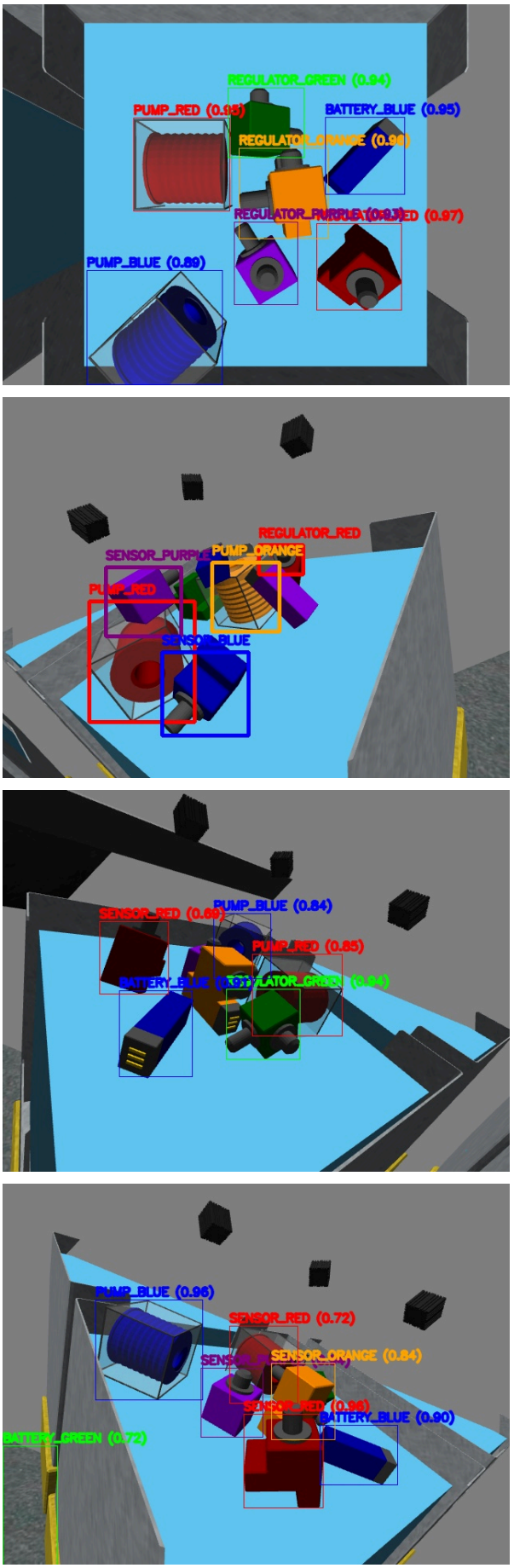

Fig. 22

Conclusion:

In conclusion, this project represents a notable achievement in the domain of object detection and manipulation within complex environments. Through the utilization of Blender 3D rendering software, a substantial synthetic training dataset of over 60,000 images was generated, incorporating objects sourced from the ARIAC environment. The YOLOv8 object detection model was trained on this dataset, demonstrating its effectiveness in detecting objects amidst varying degrees of occlusion.

Comprehensive testing and analysis provided insights into the performance metrics of the YOLOv8 model, particularly emphasizing the impact of occlusion on detection accuracy. This understanding is crucial for informing future optimizations and improving real-world performance.

Furthermore, the deployment of the object detection model within the ARIAC environment marks a practical application of the research, facilitating enhanced automation and efficiency in industrial settings. Additionally, the implementation of Normal Vector Analysis to identify optimal grasping points in cluttered environments showcases a commitment to addressing real-world challenges with innovative solutions.

Looking forward, the focus will be directed toward further integration and refinement. Future efforts will concentrate on deploying the solution within the ARIAC environment and seamlessly integrating it with the ROS2 MoveIt framework. By doing so, the aim is to enhance the capabilities of robotic systems, enabling them to navigate and manipulate objects with precision and adaptability in dynamic environments. This project not only highlights technical prowess but also underscores dedication to advancing the field of robotics and automation.

References

[1] Ultralytics YOLOv8 Docs - Object Detection.

[2] YOLOv8 Github Repository

[3] Synthetic data generation for YOLO with Blender and Python

[4] Detection Model of Occluded Object Based on YOLO Using Hard-Example Mining and Augmentation Policy Optimization

[5] Blender Manual

[6] Blender Camera Follow Path for Dynamic-Shots

[7] Make Camera Follow Path in Blender

[8] ARIAC documentation

[9] Processing Data from an RGBD Camera

[10] ARIAC Github Repository

[11] Train Yolov8 object detection on a custom dataset

[12] Yolov8 custom dataset training Github Repository