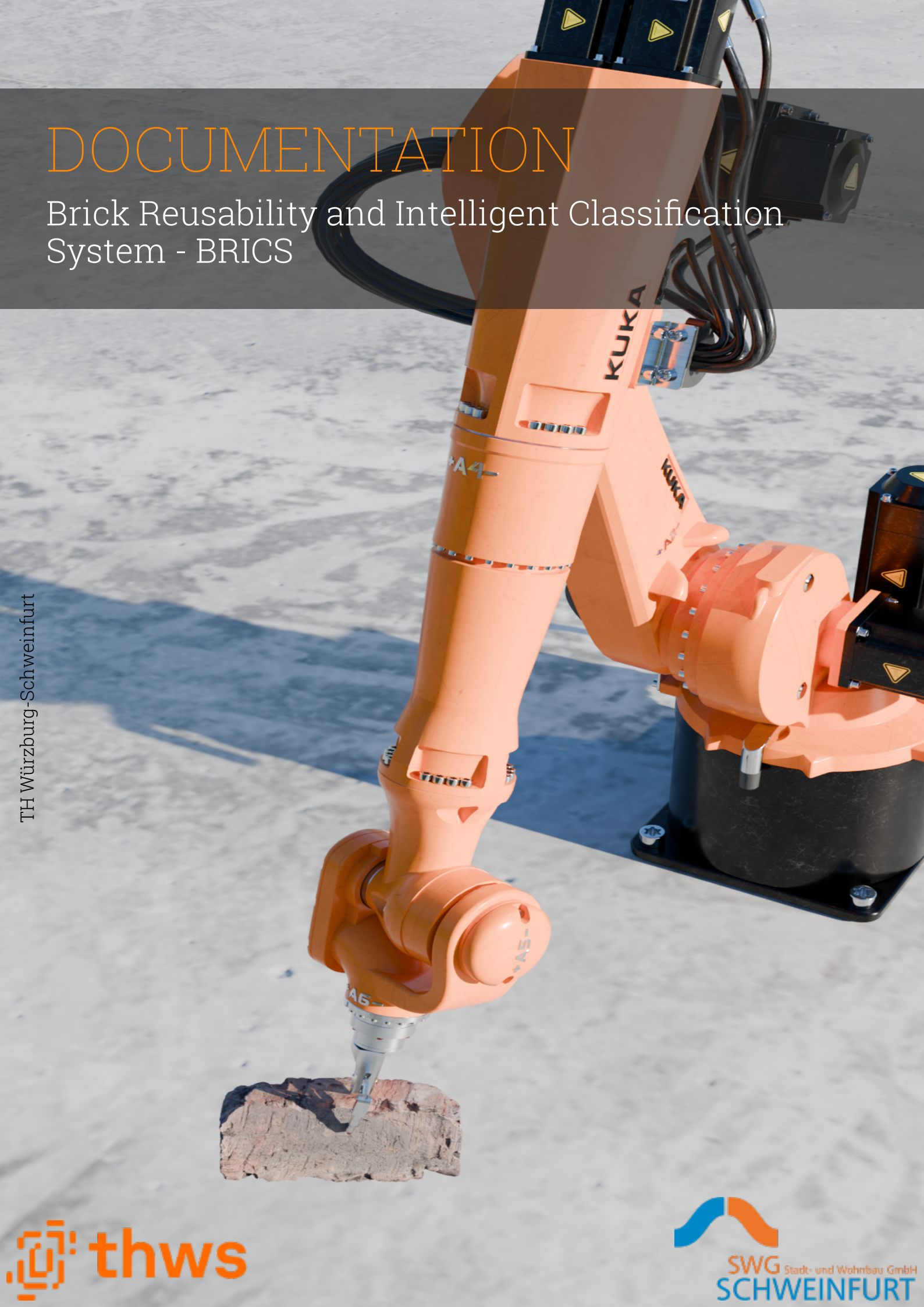
DOCUMENTATION

Brick Reusability and Intelligent Classification System - BRICS



TH Würzburg-Schweinfurt

thws

SWG Stadt- und Wohnbau GmbH
SCHWEINFURT

Brick Reusability and Intelligent Classification System - BRICS

by

Chirag Anand Reddy
Antonio Sabotinov
Anirudh Panchangam Ranaganath
Vamsi Krishna Gupta Makam

THWS Supervisors:	Tobias Kaupp and Martin Löser
SWG Schweinfurt contact:	Marco Karch, Stefan Orth and Cylvia Selavanathan Jenita
Project Duration:	15/03/24 - 15/01/25
Faculty:	Faculty of Electrical Engineering, THWS

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
SAM	Segment Anything Model
YOLO	You Only Look Once

1

Abstract

The construction industry generates a significant amount of waste, with bricks often being discarded despite their potential for reuse. This project, Brick Reusability and Intelligent Classification System (BRICS), aims to develop an automated process for assessing the reusability of bricks, thereby contributing to environmental sustainability and cost savings in construction. Our system leverages cutting-edge computer vision models like YOLO (You Only Look Once) and Segment Anything Model (SAM) for real-time object detection and precise segmentation of bricks into usable, slightly usable, and unusable categories based on their physical condition. BlenderProc, a procedural content generation tool, is utilized to create photorealistic synthetic datasets that enhance the training and testing of our computer vision models. By simulating various environmental conditions, BlenderProc aids in improving the accuracy of brick classification and detection tasks. Additionally, RoboDK is employed to simulate and optimize the robotic handling and sorting processes, minimizing human intervention and ensuring efficient operations. This integrated approach not only reduces CO2 emissions and material waste but also provides construction companies with a viable solution for recycling bricks. The project demonstrates a significant advancement in sustainable construction practices by combining machine learning, robotic automation, and 3D scanning technologies.

2

You Only Look Once

2.1. Introduction

The YOLO(You Only Look Once) model is a real-time object detection artificial neural-network that frames object detection as a single regression problem, predicting both class probabilities and bounding boxes directly from full images in one evaluation.

How did we use YOLO in our project?

- Collected a dataset.
- Manual Annotation using free annotation tools as Roboflow and CVAT.
- Trained the YOLOv8 model with our dataset.
- Tested

2.2. Main Goal

The main goal of YOLO-Detection was to achieve a bounding box with 90% confidence level for each brick on the pallet. This bounding box is important for the robot to know the exact position of the brick on the pallet for it to pick it up for the next step.

This process was performed multiple times with multiple datasets as we were facing problems with the output. Generally, the model being overtrained, which means that the model predicts very well when the bricks are arranged on the pallet with a certain distance interval to the camera, otherwise the accuracy drops significantly

2.3. Problems Faced

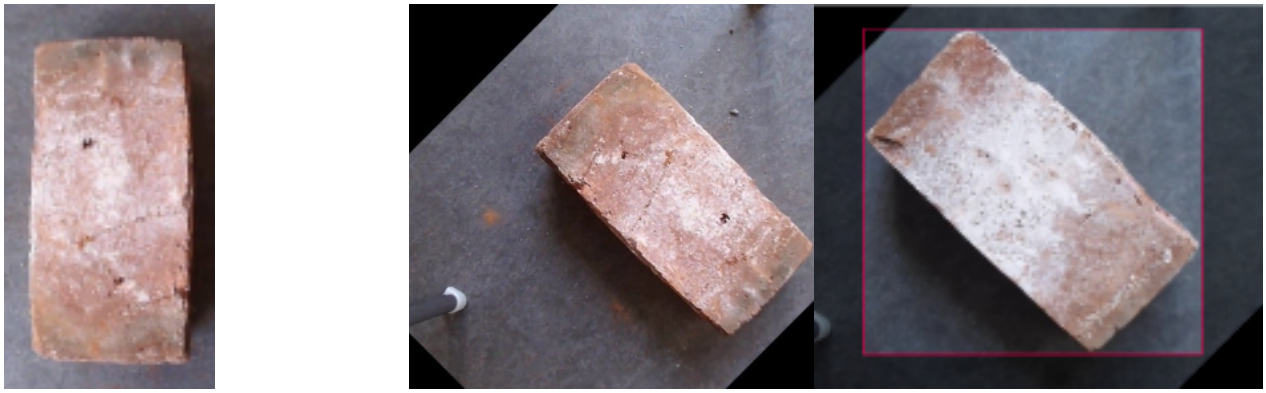
How did we tackle this problem?

- Collected a larger dataset with different lighting conditions.

(a) Input **(b)** Desired Output

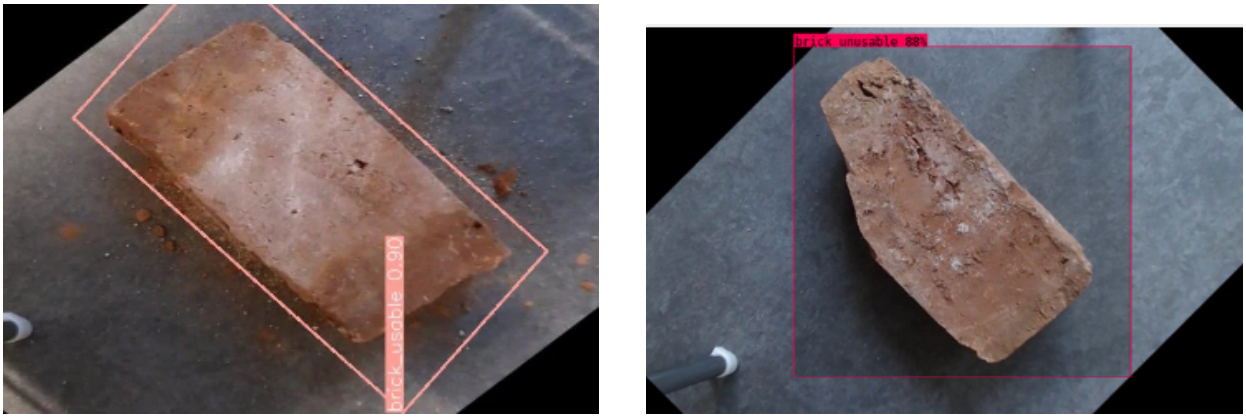
Figure 2.1: Goal of YOLO

Figure 2.2: Some of the first results from YOLOv8n - model done during inference. Dataset 320 images



(a) Dataset **(b)** Augmentation **(c)** Annotation

Figure 2.3: Comparison of Dataset, Augmentation, and Annotation



(a) Rotated Bounding Box **(b)** Classification

- Applied different augmentation steps.
- Trained models for individual bricks only.

2.4. Our Solution

Now that we solved the problem of overfitting, our goal was to generate:

- Rotated bounding box
- Classification for the bricks in two categories usable and not usable

Now, Why do we need rotated bounding box?

Rotated bounding box is important as in the real word, the bricks are not perfectly oriented on the pallet. Classification is important at this step as a few bricks from the first glance can be determined and separated easily.

Comparison of the results

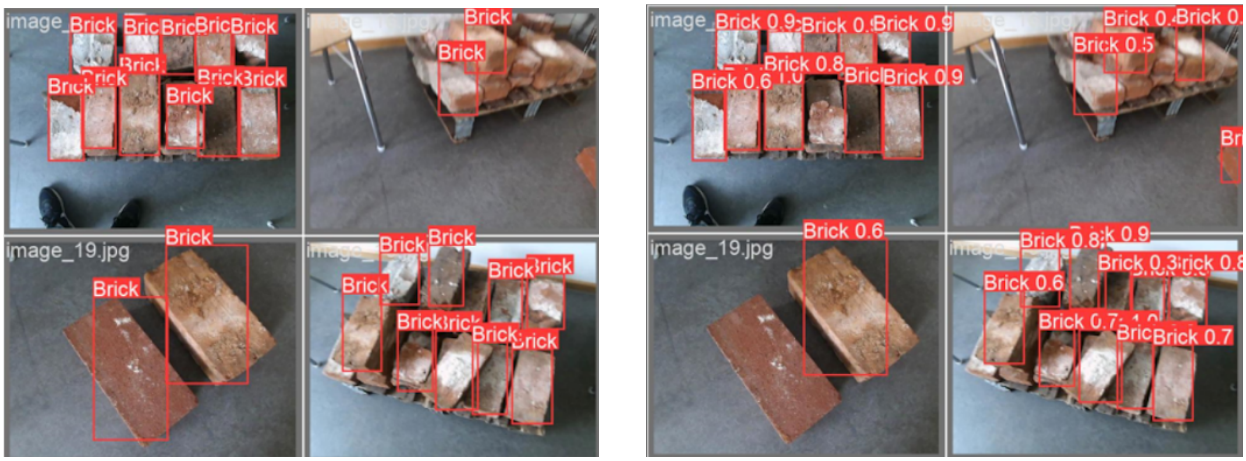
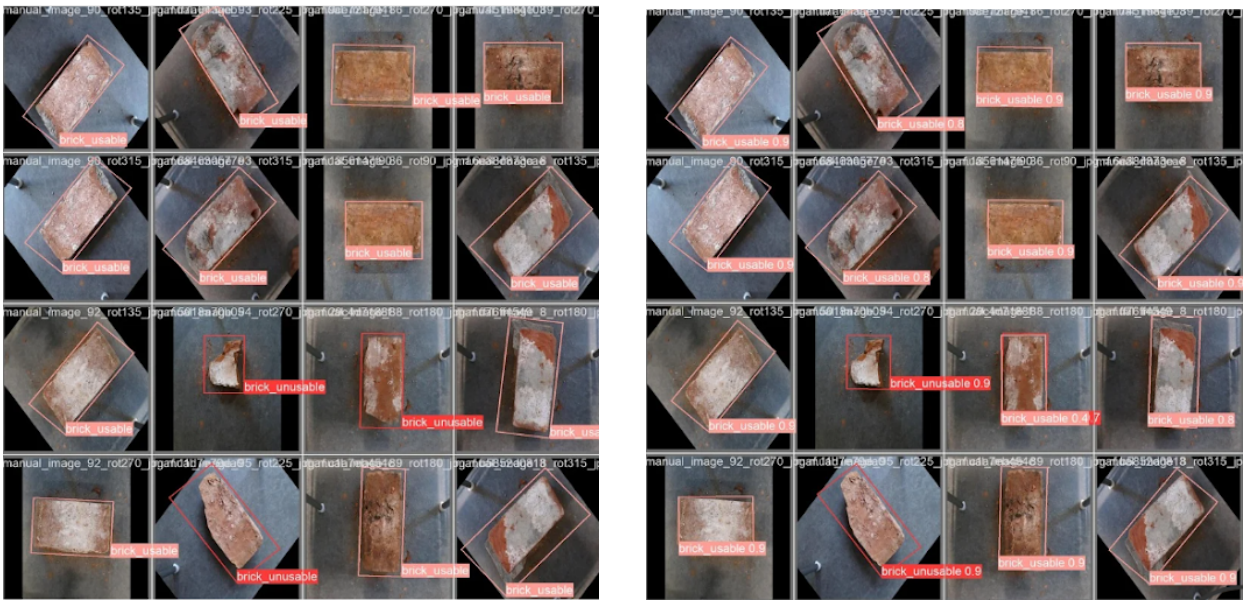


Figure 2.5: YOLOv8n (nano) model with training set of 320 – 1050 Images



(a) Training Set **(b)** Validation Set

Figure 2.6: YOLOv8n-obb (oriented bounding box) model with training data of 2000 Images

2.5. Deeper Insights

We have trained a lot of different YOLO-models such as YOLOv8n, YOLOv8m or YOLOv8x-obb.

While conducting several tests on unseen images and a variety of constellations the best and the fastest to train model was the YOLOv8n. It has less parameters and can generalize very well on our task.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Model	size (pixels)	mAP^{test} 50	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n-obb	1024	78.0	204.77	3.57	3.1	23.3
YOLOv8s-obb	1024	79.5	424.88	4.07	11.4	76.3
YOLOv8m-obb	1024	80.5	763.48	7.61	26.4	208.6
YOLOv8l-obb	1024	80.7	1278.42	11.83	44.5	433.8
YOLOv8x-obb	1024	81.36	1759.10	13.23	69.5	676.7

Figure 2.7: Different YOLO models provided by Ultralytics

While training we can observe the typical L-curve, which indicates the drop of the loss (approximation to the real data) after each epoch.

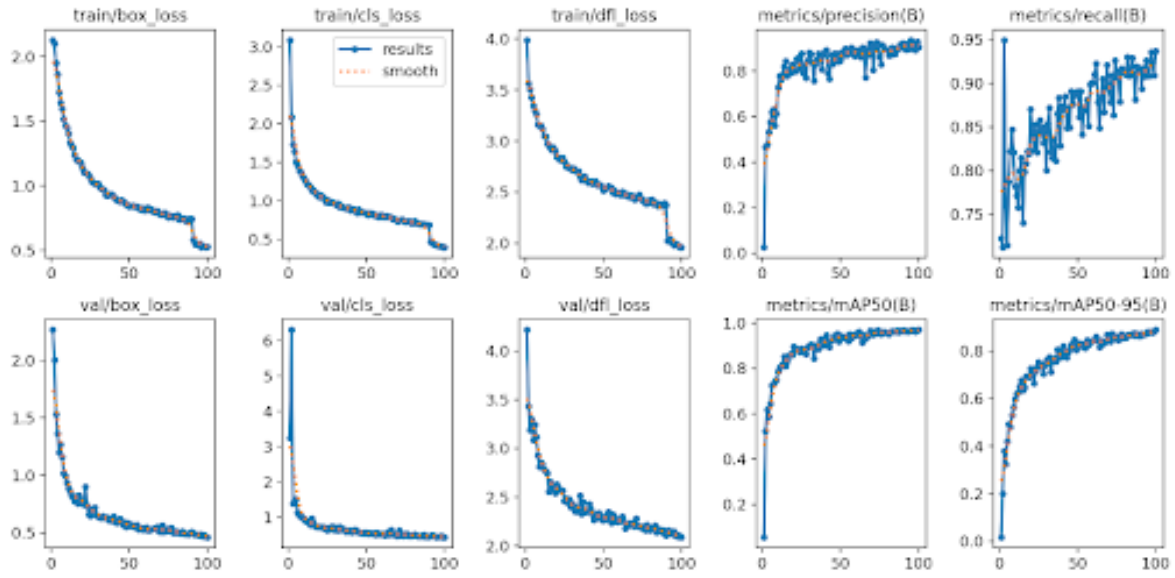


Figure 2.8: Results from YOLOv8m-obb training with 2100 images for single brick 2 class model and 100 epochs.

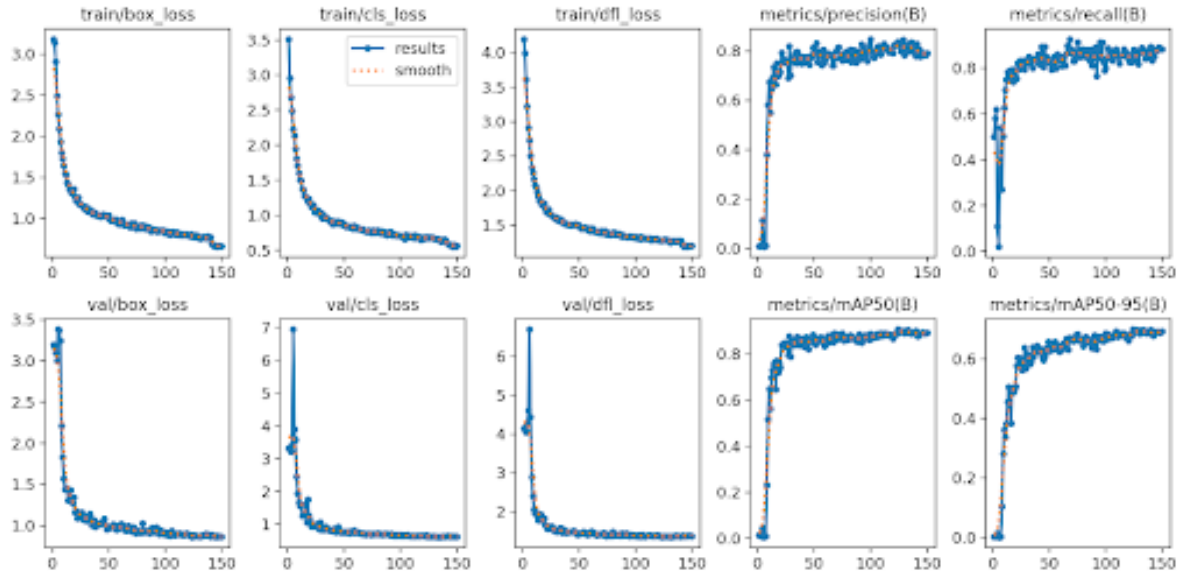


Figure 2.9: Results from YOLOv8n training with 300 images for 1 class model ant 150 epochs.

3

BlenderProc

BlenderProc is a versatile procedural content generation tool specifically designed for creating photorealistic synthetic datasets using Blender, an open-source 3D creation suite. It enables researchers and developers to automate the production of large-scale 3D environments and scenes, which are essential for training and testing computer vision algorithms, such as those used in object detection, segmentation, and depth estimation. By leveraging BlenderProc, users can define scene properties, object placements, lighting conditions, and camera configurations programmatically, ensuring consistency and variability in their datasets. This automation not only accelerates the dataset generation process but also enhances the reproducibility and scalability of computer vision research. The following sections explain how this project uses BlenderProc to generate images of brics to train our machine learning models for detection and classification tasks.

3.1. Installation and Usage

We can find the BlenderProc tool hosted on this github page[1]. Installing it is as simple as `pip install BlenderProc` . Running this command installs blender as well as the python packages. Blenderproc only supports blender version 3.5.1.

To use this tool, one can run a python script like so : `BlenderProc run example.py` to only run the program without any blender GUI or `BlenderProc debug example.py` to run BlenderProc within a blender window. The second method is useful to debug using the GUI and to visually change any elements in the environment or scene.

3.2. The Scene

The blender scene inside blender consists of a room, two adjustable area lights and a 3D scanned palette. It also consists of a camera of which the view can be changed to

[1] `https://github.com/DLR-RM/BlenderProc`

any pose to capture a variety of angles and heights. The lights can be moved around in code as well to mimic different lighting conditions.

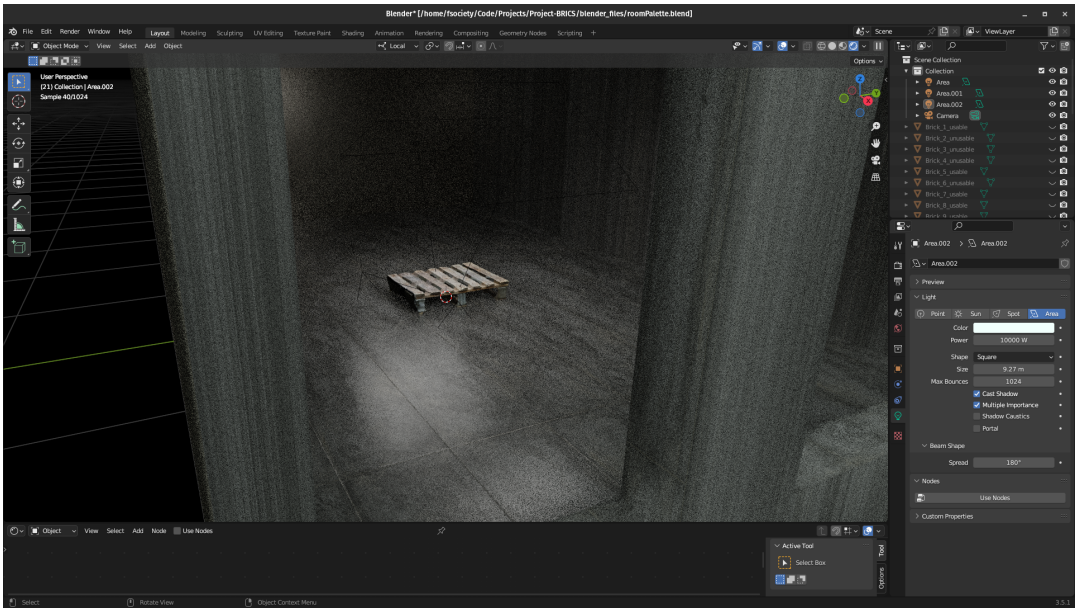


Figure 3.1: BlenderProc Environment

3.3. Polycam Tool

The 3D models that are to be used in the scene to generate images were taken from real bricks and a palette using a mobile app called Polycam[2]. The bricks were scanned by taking 100s of photos in good lighting. Using 3D reconstruction, it generates a 3D point cloud which we then import into blender, clean any outlier points/surfaces and scale the brick to actual measurements. We currently have 30 scans of different kinds of bricks which we will use to train a model to partly recognise if a brick is reusable or not.

Figure 3.2: Brick from Polycam

[2] `https://poly.cam/`

3.4. Image Generation

Now that we have the scene and all the models ready, it comes down to the script that puts together everything and generates the images. This process is mainly divided into 2 scripts, one written in bash and one in python.

3.4.1. The Bash Script

This script[3] ensures that the script that places and renders the image completes generating the whole number of desired images. This is required because sometimes blender can crash and due to this the loop of generating images is stopped. This script ensures that until the total number of images in not met, it will return a value that is different than desired, which will cause this bash script to re-run the python file again using BlenderProc.

One other advantage of this script is that it also re-runs the python file every 50 images generated because the process of rendering takes longer every time because of increasing GPU memory. This makes the average rendering time of an image to be around 40 seconds.

3.4.2. The Python Generation script

This script[4] is the main script of the whole BlenderProc segment of the project. As the name suggests, it is used to generate the bricks images using blenderproc. The flow of processes are as follows:

- It first loads the .blend scene and the 3D models of the bricks into the scene
- Next, it chooses a random pose for the 2 lights facing the pallet.
- It then chooses a random pose and height for the camera.
- Next, we choose 8 random bricks from the 30 we have and place it randomly on 8 predefined equal centers on the pallet.
- Next, we manipulate the rotational pose of the brick along the z axis so as to add variation to the dataset.
- The next step is to render the images as seen from the camera view.
- Lastly, but most importantly, the rendered images are stored in a predefined location along with the annotations of all the bricks

The annotations consists of, for each image, the names(labels)[5] of the bricks along with a binary mask for each brick.

[3] `https://github.com/anipr2002/Project-BRICS/blob/develop/src/BlenderProc/run_gen.sh`

[4] `https://github.com/anipr2002/Project-BRICS/blob/develop/src/BlenderProc/Generate.py`

[5] The naming scheme is : Brick-x-Usable / Brick-x-Unusable

Figure 3.3: Rendered bricks

3.5. Image Annotations

Now we have the capability to produce thousands of realistic images, but the new challenge is to annotate all the images. Annotating an images means to define bounding boxes in the image around the interested class of object to detect/classify along with it's label. Doing this by hand would take many days or even weeks if the dataset were to increase to thousands. But a clever trick to automate this process to be completed in a matter of mere minutes it to leverage the fact that we already know what objects are present in the images as well which class they belong to and also the exact location of the object down to the pixel.

Using the information from the above mentioned annotations, we can run a simple script[6] to define the rotated bounding box points and generate a dataset of the format required to train the Yolov8-obb model to partially detect and classify if a brick is usable or not.

[6] `https://github.com/anipr2002/Project-BRICS/blob/develop/src/BlenderProc/Annotate.py`

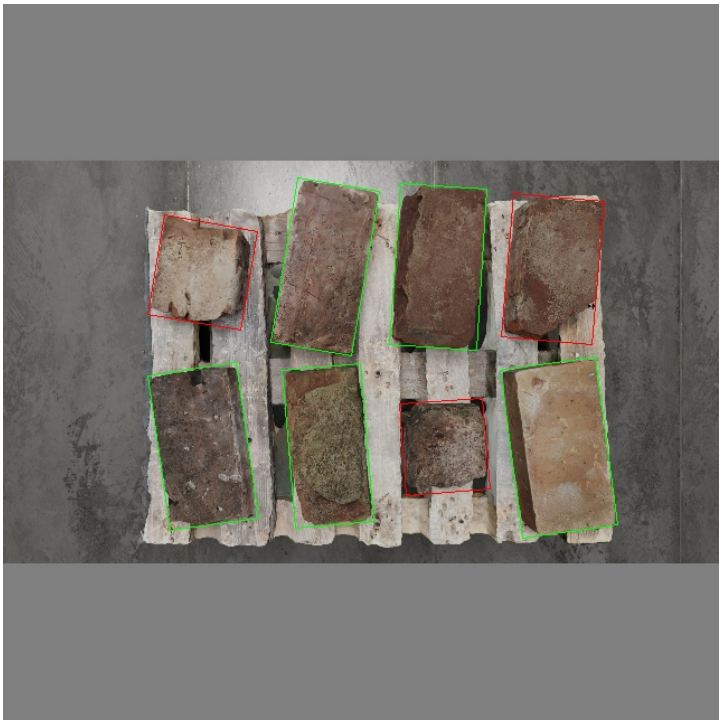


Figure 3.4: Annotated bricks

Figure 3.5: Masked bricks

4

Segment Anything Model

The Segment Anything Model (SAM) is an instance segmentation model developed by Meta Research and released in April, 2023. SAM was trained on 11 million images and 1.1 Billion segmentation masks.

Using Segment Anything, you can upload an image and:

- Generate segmentation masks for all objects SAM can identify;
- Provide points to guide SAM in generating a mask for a specific object in an image, or;
- Provide a text prompt to retrieve masks that match the prompt.

SAM has a wide range of use cases. For instance, you can use SAM:

- As a zero-shot detection model, paired with an object detection model to assign labels to specific objects;
- As an annotation assistant, a feature made available in the Roboflow Annotate platform, and;
- Standalone to extract features from an image, too. For example, you could use SAM to remove backgrounds from images.

4.1. Segment Anything Model Architecture

The SAM architecture consists of three main components:

1. **Image Encoder:** The image encoder is responsible for processing and transforming input images into a comprehensive set of features. It uses a transformer-based approach, similar to advanced NLP models, to compress images into a dense feature matrix. This matrix forms the foundational understanding from which

the model identifies various image elements.

2. **Prompt Encoder:** The prompt encoder is a unique aspect of SAM that sets it apart from traditional image segmentation models. It interprets various forms of input prompts, such as text, points, rough masks, or combinations thereof. This encoder translates these prompts into an embedding that guides the segmentation process, enabling the model to focus on specific areas or objects within an image as directed by the input.
3. **Mask Decoder:** The mask decoder synthesizes information from both the image and prompt encoders to produce accurate segmentation masks. It determines the precise contours and areas of each segment within the image, generating the final output

Key features of SAM's architecture include:

- **Promptable Segmentation :** SAM can generate valid segmentation masks from any given prompt, such as spatial or text clues identifying an object
- **Real-time Performance :** The architecture allows for real-time mask computation, making it suitable for interactive applications
- **Ambiguity Awareness :** SAM can handle ambiguous cases in segmentation tasks, providing multiple plausible segmentations when necessary
- **Zero-shot Capability :** The model can adapt to new image distributions and tasks without prior knowledge, demonstrating impressive zero-shot performance across various segmentation tasks

4.2. Segment Anything Model Dataset

SAM's training process involved a massive dataset called SA-1B, which contains over 1 billion masks on 11 million diverse images. This extensive training enables SAM to generalize well to new tasks and image domains without requiring custom data annotation or extensive retraining.

The model's versatility allows it to be used for various downstream tasks beyond its initial training, including edge detection, object proposal generation, instance segmentation, and even preliminary text-to-mask prediction. With appropriate prompt engineering, SAM can quickly adapt to new tasks and data distributions in a zero-shot manner.

In conclusion, the Segment Anything Model's architecture represents a significant advancement in image segmentation technology. Its innovative design, combining a powerful image encoder, flexible prompt encoder, and efficient mask decoder, enables SAM to perform complex segmentation tasks with unprecedented versatility and accuracy. This architecture sets a new benchmark in the field of computer vision

and opens up numerous possibilities for applications across various industries and research domains.

4.3. Implementation of SAM in BRICS

Firstly, let's talk about the available model checkpoints provided with SAM. It is provided with three different model checkpoints, each corresponding to a different backbone size:

1. **SAM-ViT-H**: This is the largest and most capable version. It contains about 636 million parameters. It offers the best performance but requires more computational resources
2. **SAM-ViT-L**: This is a medium-sized version consisting of about 308 million parameters. It provides a balance between performance and resource requirements.
3. **SAM-ViT-B**: This is the smallest version consisting of about 91 million parameters. It's the most lightweight option but may have slightly reduced performance compared to the larger models.

Benchmarking these checkpoints gave the following results :

1. ViT-H : 11.62 seconds
2. ViT-L : 3.34 seconds
3. ViT-B : 2.12 seconds

So we decided that the ViT-L checkpoint is good compromise between time and performance as you can see in the images below.

(a) ViT-B **(b)** ViT-H **(c)** ViT-L

Figure 4.1: Three model results side by side

The masks are saved per image as a json file. It can be loaded as a dict in python in the below format :

```
{
    "image"                 : image_info,
    "annotations"           : [annotation],
}

image_info {
    "image_id"              : int,              # Image id
    "width"                 : int,              # Image width
    "height"                : int,              # Image height
    "file_name"             : str,              # Image
      filename
}

annotation {
    "id"                    : int,              # Annotation
      id
    "segmentation"          : dict,             # Mask saved
      in COCO RLE format.
    "bbox"                  : [x, y, w, h],     # The box
      around the mask, in XYWH format
    "area"                  : int,              # The area in
      pixels of the mask
    "predicted_iou"         : float,            # The model's
      own prediction of the mask's quality
    "stability_score"       : float,            # A measure of
       the mask's quality
    "crop_box"              : [x, y, w, h],     # The crop of
      the image used to generate the mask, in XYWH format
    "point_coords"          : [[x, y]],         # The point
      coordinates input to the model to generate the mask
}
```

4.4. Using SAM for Brick Classification

Problem Statement We have to again further distinguish bricks detected as usable from YOLO into three main classes [usable, sightly_usable, unusable]

- Usable : Bricks with mortar/cement, slightly broken corners or edges.
- slightly_usable : Perfectly half bricks, bricks which are more than 75% intact.
- unusable : Bricks with major broken pieces.

Process To address the problem of distinguishing the bricks into their respective classes by analysing input images, we can break down the solution into the following steps:

1. **Image Acquisition** : Obtain high-resolution images of the used bricks.
2. **Prepressing** : Standardise the images by resizing, normalizing, and converting into SAM compatible format.
3. **Mask Generation** : After loading the ViT-L model checkpoint, used SAM to generate masks for individual bricks in the images, The segmented masks will help isolate individual features on the bricks which will be analyzed further to arrive at the result.
4. **Feature Extraction** : Analyze each mask to extract feature the indicate the condition of the brick. This involves identifying :
    - Deformaties : Cracks, chips, or irregular shapes. Use edge detection algorithms like Canny or Hough Transform to identify cracks or chips.
    - Mortar Residue : Regions with mortar still attached to the brick. Calculate metrics like the size of deformities, area of mortar, etc.
5. **Classification**: Based on the extracted features, classify the bricks as usable or unusable. This may require training a machine learning model on labelled examples of usable and unusable bricks.
6. **Output Results**: Provide a clear output indicating the status of each brick in the image—usable or unusable.

Further Ideas

Using SAM for mortar removal

During the feature extraction step mention above, we can extract the exact co-ordinates of the mortar on the brick. If 3D SAM is used, exact volume of the mortar also can be calculated. With this information we can implement a process to automatically remove the mortar from the bricks

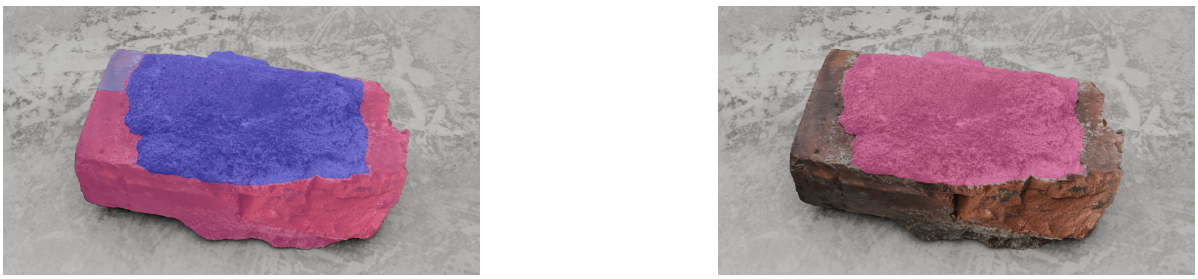


Figure 4.2: Isolating Mortar with SAM

5

RoboDK

RoboDK is a powerful and versatile software platform designed for offline programming and simulation of industrial robots. It is widely used in industries such as manufacturing, aerospace, automotive, and research to enhance robot programming efficiency and optimize robotic processes. With RoboDK, users can create, simulate, and execute complex robotic tasks without needing to write detailed code, thus reducing the time and effort required for robot programming.

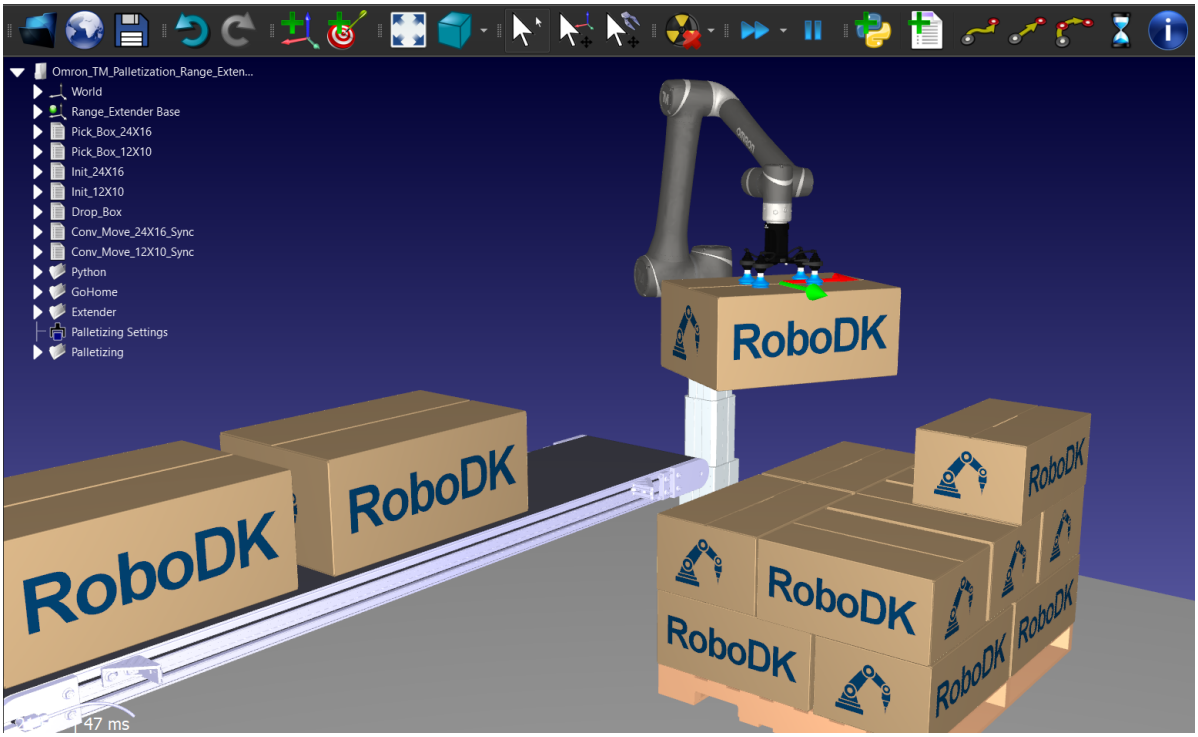


Figure 5.1: Robodk Environment

5.1. RoboDK Key Features

- Offline Programming: One of RoboDK's primary features is offline programming, which allows users to program and simulate robotic operations on a computer before deploying them on actual robots.
- CAD/CAM Integration: RoboDK seamlessly integrates with popular CAD/CAM software such as SolidWorks, AutoCAD, and Mastercam, enabling users to import complex 3D models directly into the RoboDK environment.
- Python Scripting: RoboDK offers Python scripting capabilities, providing the flexibility to customize and automate tasks through scripts. This feature allows for greater control over robotic operations and facilitates the development of complex applications. Figure

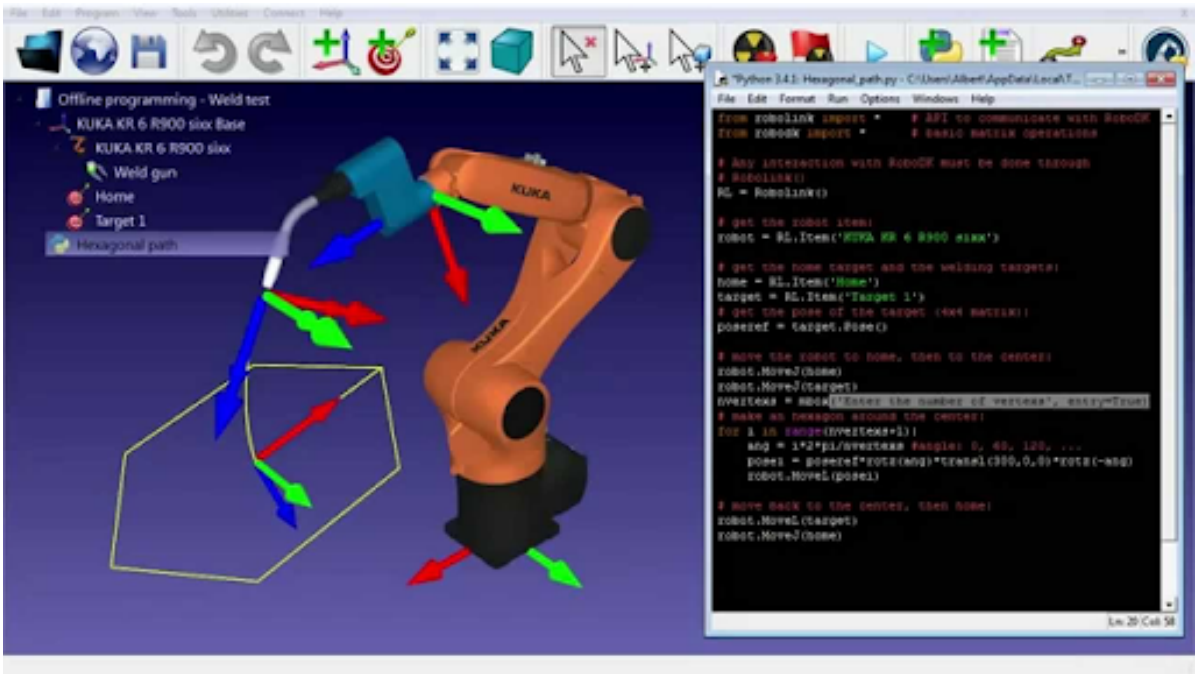


Figure 5.2: Python integration with RoboDK. Implementation.

5.2. Implementation

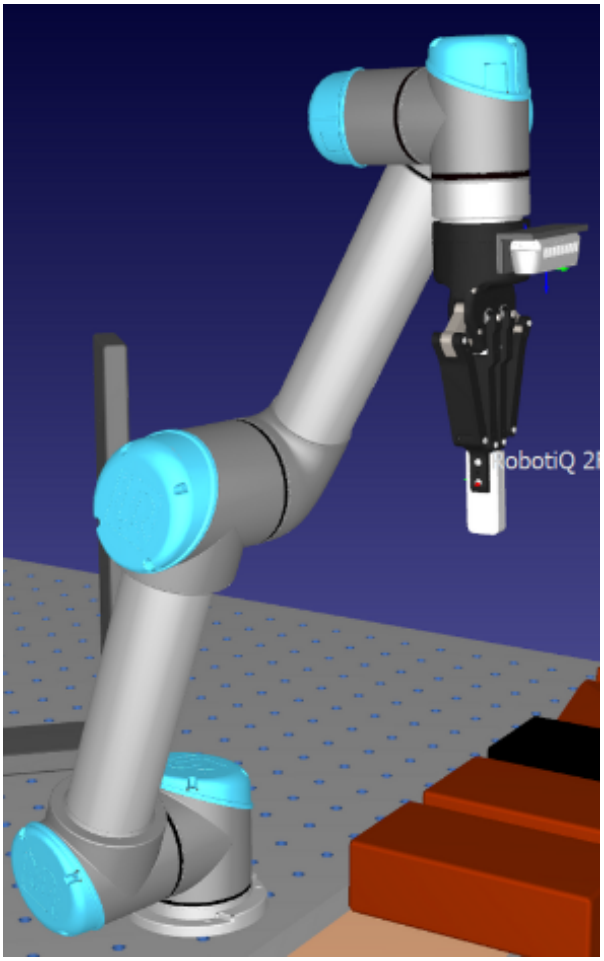
The aim is to simulate the entire process of the project within a Robodk environment. This helps with the plausibility check further helping in optimization. This also helps in checking the limitations of the process and adds collision avoidance techniques. This process is important as the minute details are important in realisation of the process.

For this entire process to be visualised, the requirements are the test bench and offline programming with python.

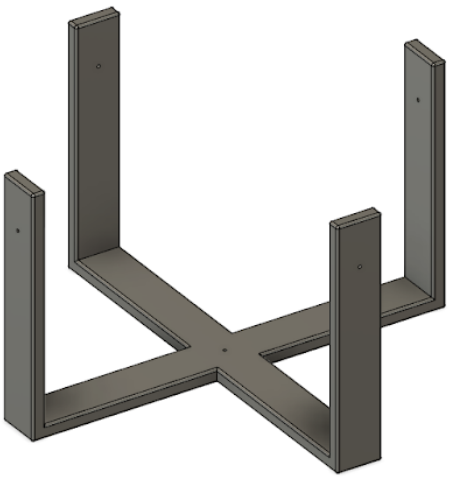
5.3. Test Bench

The testbench has three main components. The UR5 demonstrator, an Inspection stand, a pallet filled with bricks.

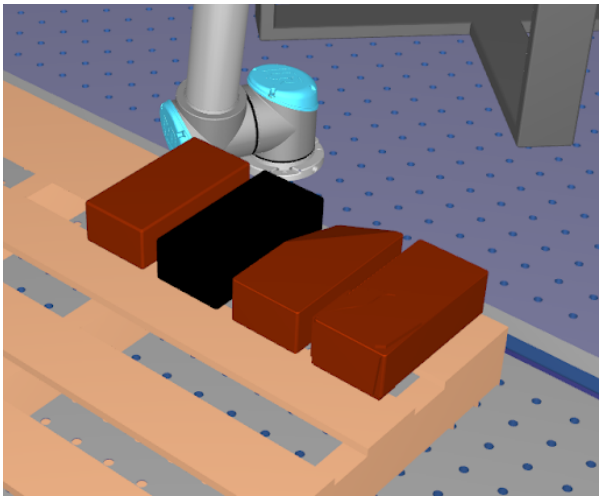
- **UR5 demonstrator :** This six axes robot arm is a medium sized robot arm with 5 kg payload capacity. The robot arm is directly found in the RoboDK library.To the robot, Realsence camera is attached for the first step of the brick analysis.
- **Inspection stand :** The inspection stand is the key component of the process.This stand is used to inspect the selected brick. The stand is fixed with five cameras for the second step of the brick analysis. It is designed in solidworks.
- **Pallet and Bricks.** For method one, a standard pallet is imported from the RoboDK library. Additionally, few bricks were designed to solidworks and imported into the environment.



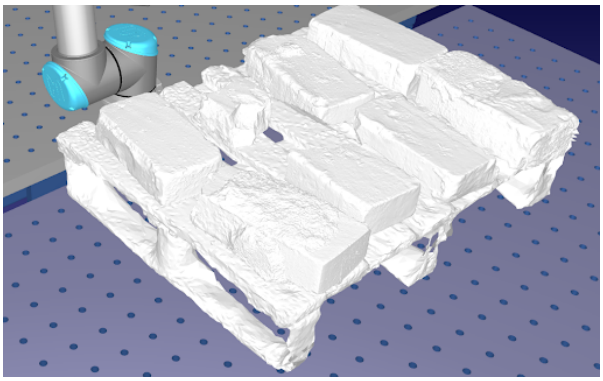
(a) UR5 Robot



(b) Inspection stand



(a) Method 1 - Pallets



(b) Method 2 - Pallets

5.4. Method

Two methods were applied to realise the process. In the first method, a program was made in RoboDK by setting target points.In the second method, the robot is programmed with python to automate the entire process.

The analysis technique remains the same for both the methods. For the first analysis step, the camera on the robot scans the pallet and uses the YOLO model to differentiate between usable and unusable bricks. The bricks which are unusable are directly thrown away

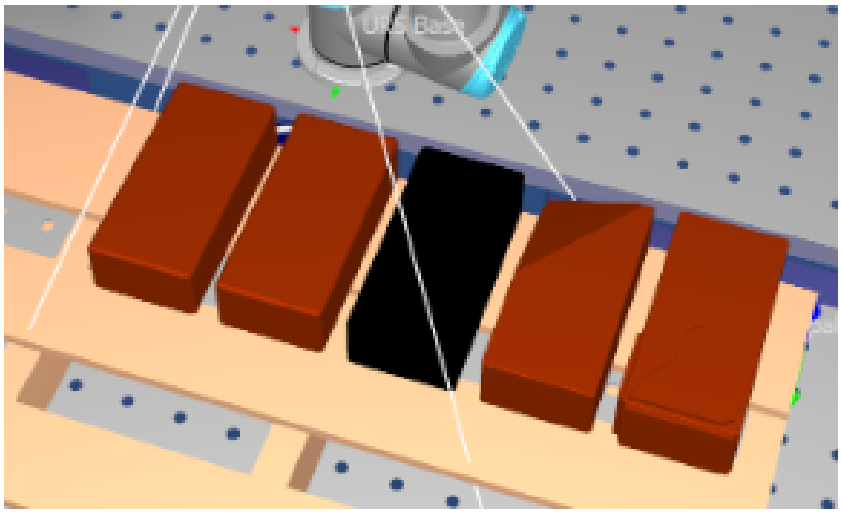


Figure 5.5: Analysis Step 1

The bricks which are usable are entered into the second analysis step. Here the robot brings the usable labelled brick to the inspection stand and perfectly holds the brick at a point equidistant from all five cameras. The cameras instantly capture pictures of the

bricks. The cameras are positioned in such a way that each camera captures one side of the brick. SAM is instantly applied to these pictures for final classification.

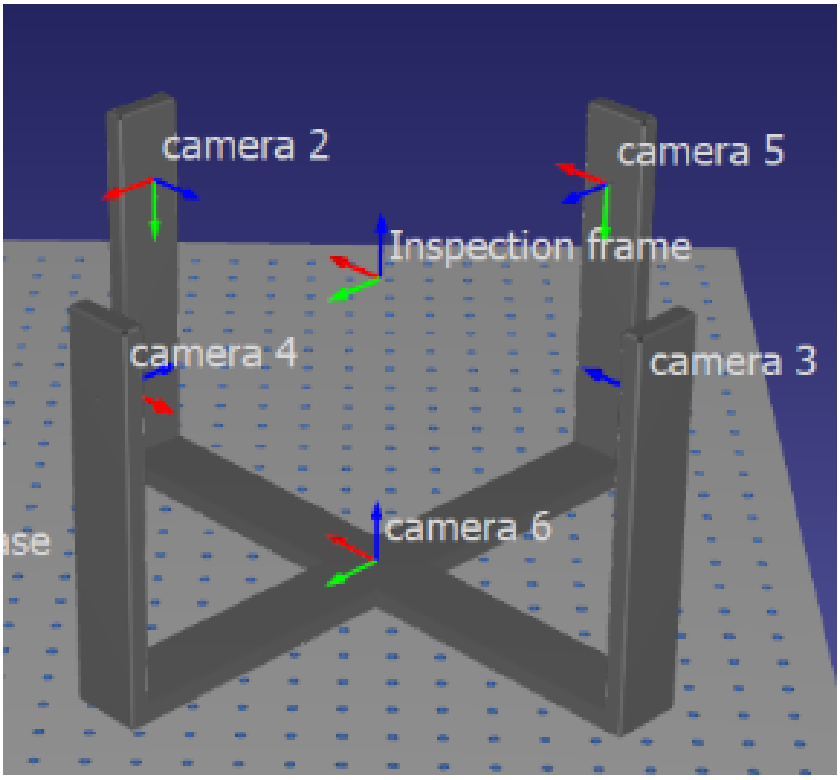


Figure 5.6: Analysis Step 2

From these steps, all the data of all the six sides of the brick is obtained and the brick is completely classified.

Figure 5.7: Final result from analysis step two

5.5. Python Script

The script[1] described in this document is designed to control a robotic manipulator (specifically, a UR5 robot arm) using the RoboDK API. The aim is to manage and process virtual brick imported dynamically from BlenderProc within the RoboDK environment.

Setup

First, the necessary libraries are imported: `os` for filesystem operations, `time` for delays, `json` for parsing JSON data, and `robolink` and `robomath` for RoboDK API functions. The RoboDK API is then initialized:

```
RDK = robolink.Robolink()
```

Retrieve RoboDK Items

Required items are retrieved from RoboDK. These include the two frames (`pallet_frame` and `usable_frame`), the robot itself (`UR5`), and its base frame (`UR5_Base`). The script then checks if all items are successfully found:

- `pallet_frame`
- `usable_frame`
- `UR5`
- `UR5_Base`

Import Bricks and Set Their Poses

The script iterates over each brick defined in the `brick_data`. The data is automatically generated from BlenderProc, in which 8 random bricks with random orientation are created. For each brick, a pose (a 4x4 transformation matrix) is constructed based on the position and rotation data, and the pose is set for the brick. The pose is adjusted with a rotation of 90 degrees around the X-axis to account for export discrepancies from Blender.

- `AddFile`
- `setPose`

Define Robot Joint Positions

Joint positions for the robot's home and scan positions are defined:

- `home_robot_joint_pos`
- `scan_robot_joint_pos`

[1] `https://github.com/anipr2002/Project-BRICS/blob/main/src/Robodk/main.py`

Process Each Brick

Identify Brick and Calculate Initial Pose

The script iterates through each brick in the `brick_data` (JSON data) and determines if the brick is usable or unusable based on its name. Various poses are extracted and calculated:

- `brick_pose_abs`
- `pallet_pose_abs`
- `usable_pallet_pose_abs`
- `ur5_pose_abs`

Process for Usable Bricks

1. Calculate Target Poses: Define a target pose just above the brick with a slight offset and transform this target pose from the pallet frame to the UR5 frame, adjusting for necessary rotations.
2. Add and Move to Target: Add a new target in the UR5 frame and set its pose. Move the robot linearly using `MoveL` to the target pose.
3. Grip and Attach the Brick: Close the gripper using predefined gripper movements and attach the closest brick to the gripper tool.
4. Inspection: Move the robot to a predefined scan position to inspect the brick. After inspection, move back to the home position.
5. Return and Detach: Return to the initial target position above the brick. Execute a predefined detachment program to release the brick. Delete the temporary target and move the robot to the home position.

Process for Unusable Bricks

1. Calculate Target Poses: Similar to usable bricks, define a target pose just above the brick and transform it to UR5 frame.
2. Add and Move to Target: Add a new target in the UR5 frame and set its pose. Move the robot linearly using `MoveL` to the target position.
3. Grip and Attach the Brick: Close the gripper and attach the closest brick to the gripper tool. Delete the temporary target, then move the robot to the home position.
4. Move to Second Pallet: Move the robot to a predefined joint position for placing bricks on the second pallet. Add a new target for the second pallet and calculate its pose. Move the robot linearly using `MoveL` to the second pallet target position.
5. Detach: Execute the detachment program to release the brick onto the second pallet. Move the robot back to the home position and delete the temporary target for the second pallet.

6

Hand-Eye Calibration

Hand-eye calibration is crucial in robotics for determining the spatial relationship between a robot's manipulator (hand) and a camera (eye). This calibration ensures the robot can accurately interact with its environment, making it fundamental for tasks like pick-and-place operations, assembly, and inspection.

6.1. Concepts and Setup

In the system, the camera (Intel RealSense D435i) is mounted on the UR5e robot in an eye-in-hand configuration. A 9x6 OpenCV checkerboard pattern is used for calibration, while ArUco markers placed on the table corner validate the results.

The calibration involves finding a transformation matrix X, mapping the robot's coordinate frame to the camera's frame. This matrix satisfies the equation:

$$AX = XB$$

Here:

- A: Motion of the robot's end-effector relative to its base (from kinematics).
- B: Camera's motion relative to the checkerboard (from image processing).
- X: The hand-eye transformation matrix.

6.2. Procedure and Challenges

Calibration Steps:

1. Capture images of the checkerboard at various robot poses.
2. Use OpenCV to detect checkerboard corners and calculate the pose relative to the camera.

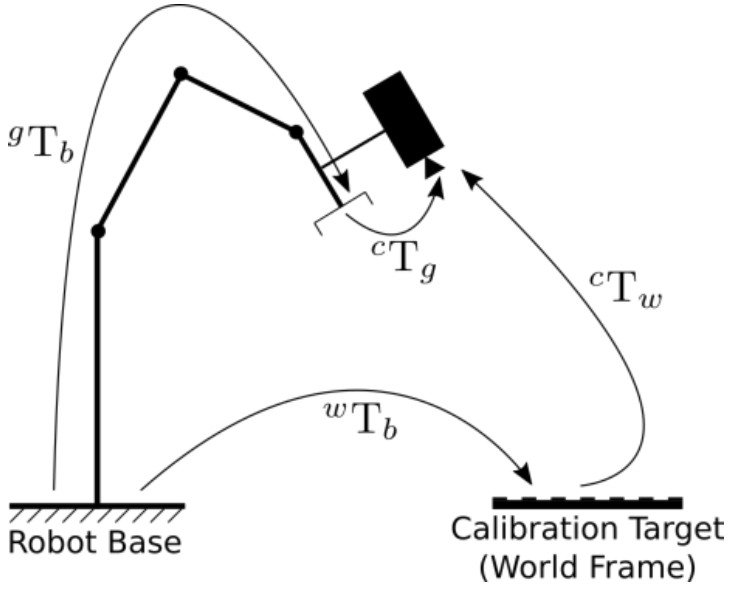


Figure 6.1: Hand-Eye Calibration Setup with RealSense D435i and UR5e Robot.

3. Record the end-effector poses from the robot's inbuilt encoders.
4. Solve $AX = XB$ using standard mathematical methods from OpenCV.
5. Validate using ArUco markers to compare expected and observed positions.

Challenges:

- Extracting accurate poses from images and encoder data.
- Minimizing noise from sensors and calibration errors.

6.3. Conclusion

The integration of the RealSense D435i camera, UR5e robot, OpenCV checkerboard, and ArUco markers forms a robust framework for hand-eye calibration. While noise and precision pose challenges, this method provides accurate and reliable results for robotic applications.

7

ROS Workspace

The ROS workspace for the BRICS project was developed to integrate various robotic components efficiently. The system utilizes the Universal Robots ROS2 driver, MoveIt for motion planning, Librealsense for RealSense camera management, and the Ultralytics library for vision-based object detection with yolo. This workspace enables the execution of the project demonstration pipeline by integrating all these software components seamlessly. The code for the package can be found here, https://github.com/anipr2002/ros_brics.git

7.1. Universal Robots ROS2 Driver and MoveIt

The project employs the Universal Robots ROS2 driver, which provides essential functionalities for controlling the UR5e robotic arm. This driver enables real-time communication with the robot and allows seamless execution of motion commands. Additionally, *Moveit*, which is included in the package, facilitates motion planning and path execution. MoveIt was used for:

- Generating and executing trajectories for brick grasping and placement.
- Collision avoidance during the movement of the robotic arm.
- Optimizing path planning for efficient handling operations.

7.2. Librealsense for RealSense Camera

The *Librealsense library* was integrated into the ROS workspace to manage data acquisition from the Intel RealSense camera. This library provides:

- Depth and RGB image streaming for detecting and localizing bricks.
- Camera calibration and intrinsic parameter handling.
- ROS2-compatible drivers for interfacing with the perception module.

7.3. Ultralytics Library for Object Detection and Pose Estimation

For real-time brick detection and pose estimation, the Ultralytics YOLOv11 library was utilized. This library played a crucial role in determining the brick's center point and orientation, which was essential for robotic grasping. The main functionalities included:

- Detecting bricks on the conveyor belt using deep learning-based object detection.
- Estimating the center point and pose of the detected bricks.
- Providing real-time data to the motion planning pipeline to facilitate robotic pickup.

7.4. Overview of Running Nodes

This chapter provides a comprehensive explanation of the nodes running in our robotic system. Each node plays a crucial role in ensuring that the robot operates efficiently and accurately. The following sections detail the functionality and importance of each node.

UR Controller

The UR Controller is initiated via a dedicated launch file. It is responsible for launching the robot's controller along with RViz, a visualization tool. RViz allows users to monitor the robot's status in real time, offering a graphical interface to observe the robot's movements and sensor feedback. This integration is vital for debugging and ensures that any issues in the controller's operation can be quickly identified and resolved.

Moveit Launcher

The Moveit Launcher is designed to initialize Moveit, the motion planning framework for robotic arms. By launching Moveit, the system gains the ability to plan and execute complex trajectories, ensuring that the robot moves smoothly and safely from one pose to another. This node is crucial for tasks that require high precision and dynamic response in motion planning.

Static Transform Publisher Node

This node is tasked with publishing the static transforms of the robot. Static transforms define the fixed relationships between different coordinate frames in the robot, such as the TCP (Tool Center Point) relative to the robot's base. By maintaining an accurate transformation tree, the system ensures that the robot's perceived position and orientation are consistent with its actual physical configuration. This is particularly important for tasks that involve precise manipulation and interaction with the environment.

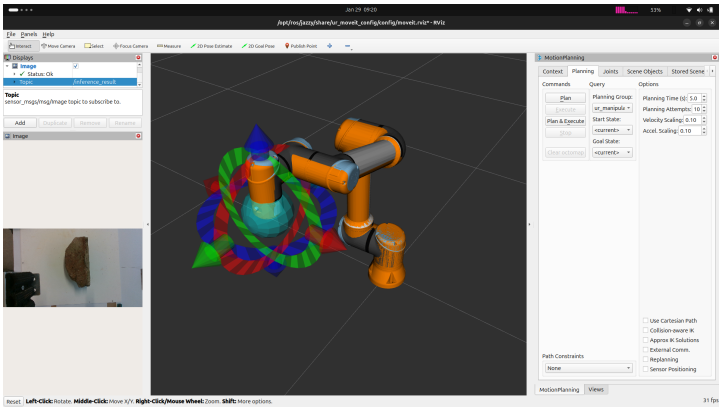


Figure 7.1: Moveit Rviz Dashboard

Realsense Node

The Realsense Node is responsible for initializing the Realsense camera and publishing the image topics. The camera provides critical visual feedback, which is essential for both navigation and object detection. The published image topics are used by other nodes in the system for further processing, ensuring that real-time visual data is seamlessly integrated into the robot's decision-making processes.

Yolo Node

The Yolo Node subscribes to the image topic provided by the Realsense Node. Its primary function is to perform real-time object detection using the YOLOv8 model. Detected objects are then published on the `Yolov8Inference` topic. This node is integral for enabling the robot to understand and interact with its environment by recognizing various objects, which can then be used for tasks such as obstacle avoidance, object manipulation, or scene understanding.

Move to Pose Action Server

The Move to Pose Action Server is a dedicated node that handles commands for moving the robot to a specified pose. It translates high-level pose instructions into precise motor commands, ensuring that the robot reaches the target position with the desired orientation. This node is crucial for tasks that require precise positioning, such as assembly operations or detailed inspection tasks.

In summary, the coordinated operation of these nodes underpins the entire robotic system, allowing it to perform complex tasks through a combination of motion planning, sensor data processing, and real-time decision-making.

8

BRICS Gripper

8.1. Robotiq 2F-140 Gripper

The Robotiq 2F-140 gripper is a highly versatile and adaptive 2-finger robotic gripper designed for collaborative and industrial applications. Its flexibility and ease of integration make it a popular choice for automation projects.

Key Properties

- The gripper has an adjustable gripping range of 0 to 140 mm.
- The gripper can support payloads of up to 2.5 kg.
- The grip force is adjustable and ranges from 10 N to 125 N.
- The fingertips of the gripper are replaceable and customizable.

Limitations with Robotiq 2F-140

- The original Robotiq gripper fingers had a narrow gripping area, which made it difficult to securely grasp larger objects such as bricks.
- Slippage occurred frequently, especially when picking up bricks with mortar or uneven surfaces. This was primarily due to the smooth surface of the gripper fingers.
- The force range of the Robotiq gripper was insufficient to secure heavy or irregularly shaped bricks during manipulation.

Figure 8.1: Robotiq 2F-140 Gripper

8.2. BRICS Gripper Design

In the BRICS project, the original Robotiq 2F-140 gripper was found to have limitations when handling certain materials, specifically bricks with mortar.

Custom Gripper Design

To address these challenges, a custom gripper design was developed and implemented. The modifications included:

- A wider finger design created using extrusions to increase the overall grasping area and provide additional width. This adjustment allowed the gripper to better accommodate the bricks.
- The grasping surface was 3D printed using thermoplastic polyurethane (TPU). This design, inspired by a soft robotics concept, helps in grasping irregular shapes. TPU was chosen for its flexibility and durability.

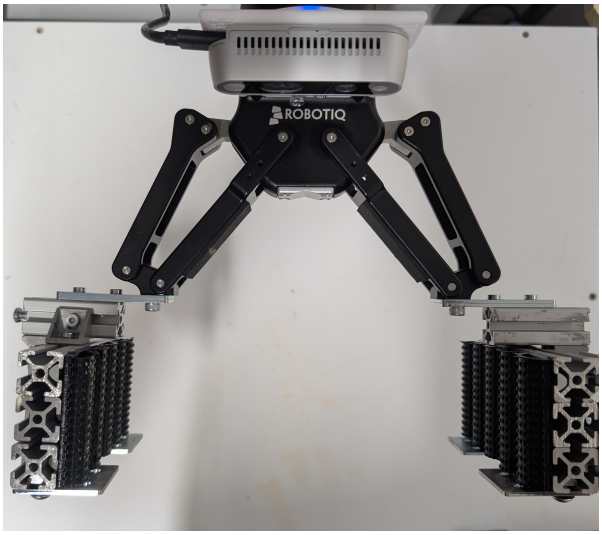


Figure 8.2: BRICS Custom Gripper

8.3. Results and Remaining Challenges

The custom gripper effectively solved the slippage problem, enabling more reliable handling of bricks, even those coated with mortar. However, the limitation of the grip force of the Robotiq 2F-140 gripper remained a challenge. Despite improvements in gripping area and surface adaptability, grip force was insufficient to manipulate heavier bricks. The only solution to this problem is to use a different gripper with a higher grip force.

9

Project Pipeline

A project pipeline defines a structured sequence of steps required to achieve a specific goal, typically in an automated process. It ensures efficiency and consistency in complex operations by integrating various hardware and software components.

The project pipeline for our robotic system integrates a RealSense camera, YOLOv11 model, and a streamlined brick inspection process to efficiently classify and handle bricks as usable or unusable.

The colors in the flowchart indicate whether a particular process step is an automated or manual process. The violet and turquoise colors indicate an automated step, while yellow represents a manual process step. By using this color-coded approach, operators can quickly distinguish between tasks requiring human intervention and those executed autonomously, improving overall efficiency and reducing errors.

9.0.1. Changes in the Inspection Stand

In the previous iteration of the pipeline, a five-camera inspection stand was used. The robot brought each brick to the stand, where images of all sides (top, bottom and sides) were captured for classification. However, this set-up was unnecessarily complex. It was determined that sufficient data for an accurate classification could be obtained from only two images:

- The top image captured by the RealSense camera mounted on the robot.
- The bottom image captured by a single camera on the inspection stand.

This change has significantly reduced the complexity of the setup, requiring only a single-camera inspection stand while maintaining classification accuracy and efficiency.

Figure 9.1: New Inspection stand

9.1. Demonstration Pipeline

The demonstration pipeline is as follows:

1. A RealSense camera is mounted on the robot's tool head, and hand-eye calibration ensures accurate spatial alignment.
2. A brick is manually placed on the test bench to begin the process.
3. The RealSense camera detects the brick and passes the data to the YOLOv11 model for classification.
4. The YOLOv11 model classifies the brick as either usable or unusable based on the top image.
5. If the brick is classified as unusable, the robot does not pick it up, and the process ends for that brick.
6. If the brick is classified as usable, the robot picks up the brick.
7. The robot moves the brick to the single-camera inspection stand, where the bottom image of the brick is captured.
8. The top and bottom images are processed by the YOLOv11 model to confirm the brick's usability.
9. If the brick is confirmed as usable, the robot places it aside in the usable bricks collection area.
10. If the brick is classified as unusable during the second check, it is discarded appropriately.

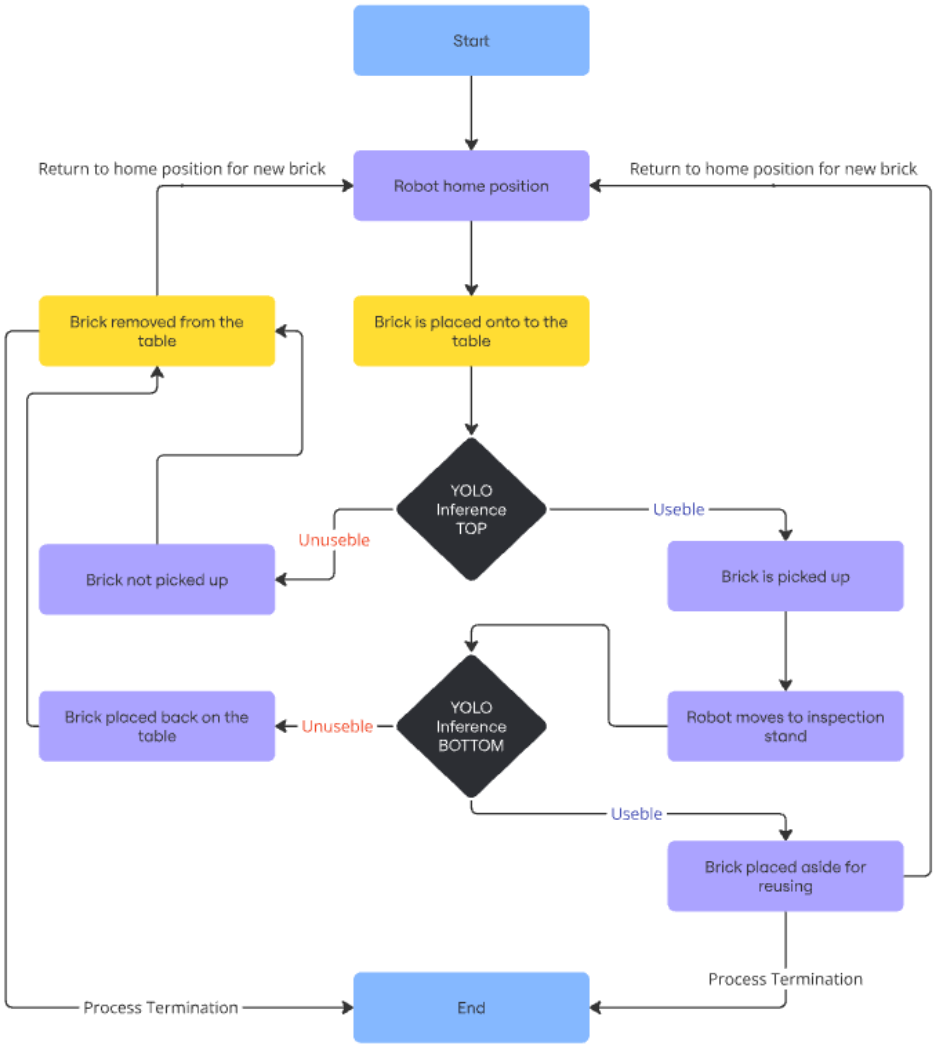


Figure 9.2: Pipeline Flowchar

9.2. Goal Pipeline

The main project pipeline improves upon the demonstration pipeline and operates as follows:

1. All bricks are thrown into the feeder system, which vibrates to separate weak or breakable bricks. Only intact bricks enter the conveyor belt.
2. The conveyor belt transports bricks to the robot station.
3. The mounted RealSense camera detects the brick and passes data to the YOLOv11 model for initial classification.
4. The YOLOv11 model classifies the brick as either usable or unusable.
5. If the brick is classified as unusable, it remains on the conveyor belt and rolls to the end, where it falls into a bin.
6. If the brick is classified as usable, the robot picks it up.

7. The robot moves the brick to the inspection stand, where the bottom image of the brick is captured by a single camera.
8. The top and bottom images are processed by the YOLOv11 model to confirm the brick's usability.
9. If confirmed as usable, the robot places the brick on the pallet designated for usable bricks.
10. If classified as unusable during the second check, the brick is discarded appropriately.

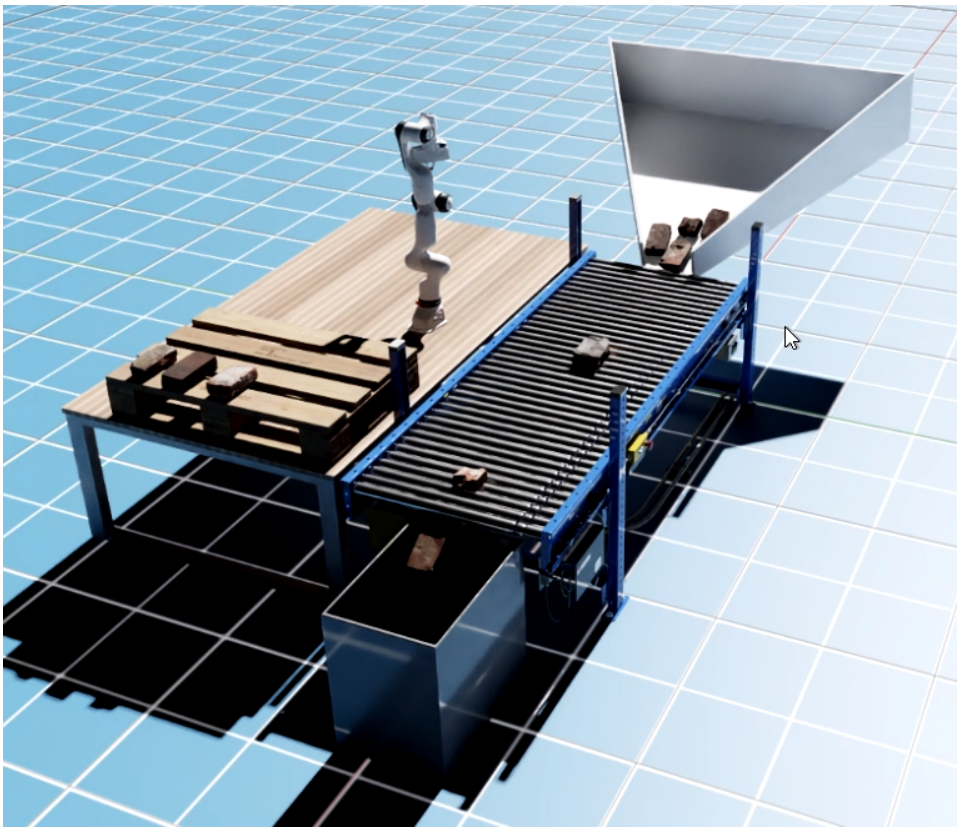


Figure 9.3: Goal pipeline(generated using Omniverse)

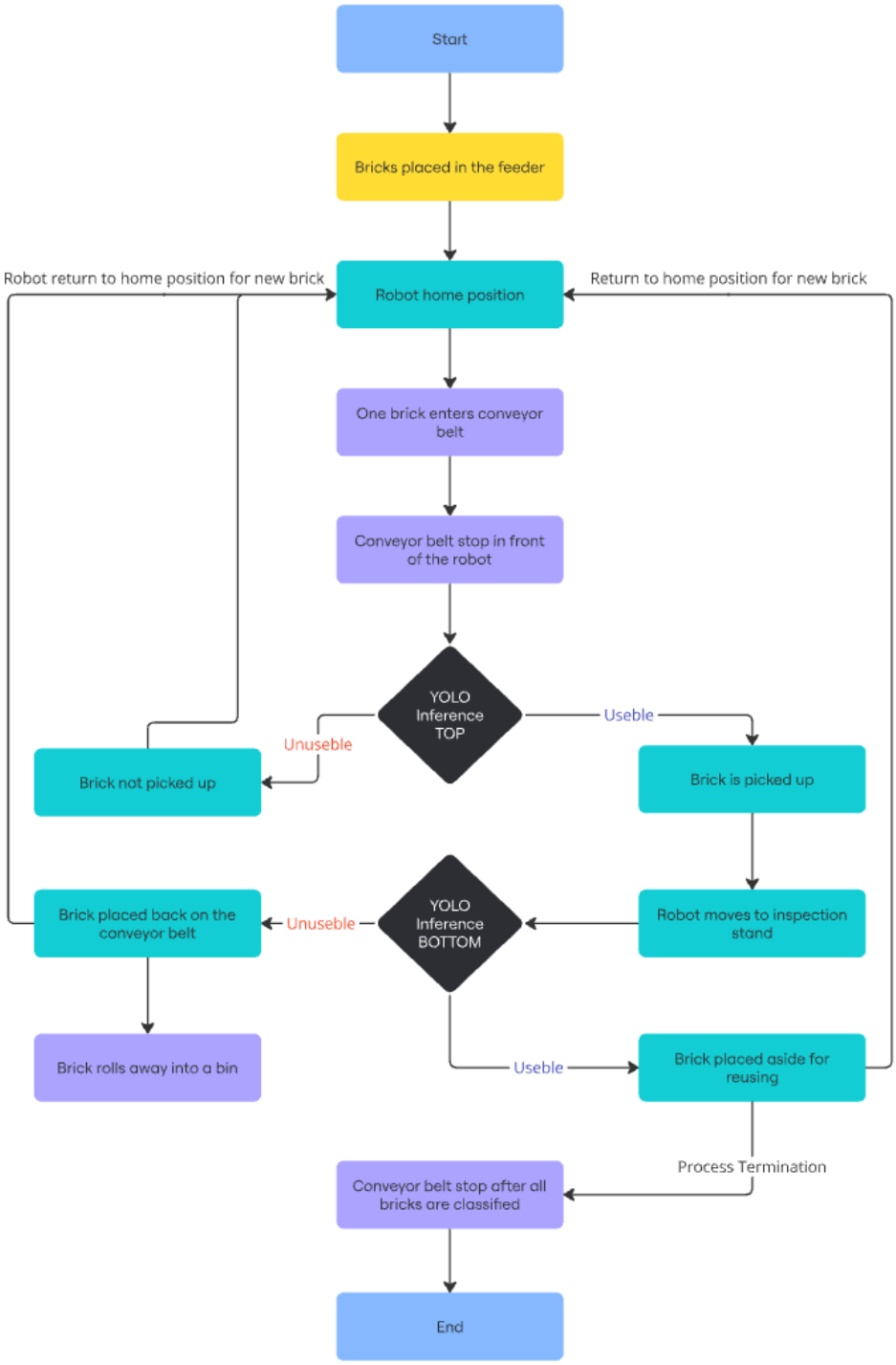


Figure 9.4: Flowchart Goal Pipeline