1. Synthetic Dataset

Synthetic datasets for computer vision are generated virtually by computer, through the use of 3D computer graphics software such as Blender. The dataset contains images that simulate the camera perspective of a real robot and an annotation of the objects intended for the robot to recognize. Subsequently, these datasets are used to train the deep learning model. In some scenarios, data collection for the vision model can be both expensive and time-consuming. However, as the scenes are replicated in a virtual workspace, the synthetic dataset can overcome this shortcoming (Abraham, 2021). Arbitrary surfaces, lighting, environment, and object color can easily be modified automatically to generate a substantial amount of unique dataset with annotation.

1.1 Blender

Blender is an open-source 3D computer graphics software and has been used to create games, virtual reality, and more. It supports the entire 3D pipeline, ranging from 3D modeling to animation, making it well-suited for generating synthetic datasets through the creation of 3D scenes. In comparison to other 3D graphics software like Nvidia Omniverse and Unity, Blender stands out due to its open-source nature, robust community support, and abundant online resources (Chillingworth, 2023). Therefore, Blender was selected as the 3D graphics software used for generating synthetic datasets for this project.

1.2 Methodology

Figure 1 depicts the flowchart for the synthetic dataset generation process. Initially, 3D objects are either imported into Blender or created using it. By repositioning them, the workstation of the arena is replicated, and users can configure the number of images and the diversity of the dataset. Before randomizing the object, material, light, and camera properties, the program will save initial settings at the start of its execution. Then, the program will name the output directory, which is used to save annotations and images, as the project's name. Subsequently, it will check for repeated naming to prevent data overwrites. In case of repeated naming, the output folder will include an incremental number appended to the folder name, such as "folder_name (1)" and "folder_name (2)", ensuring the distinct identification and organization of the folders. In each iteration, the script will randomly change the light intensity, the object position and rotation, the material color and texture, and the camera position and orientation.

Additionally, when the object changes its rotation, it will check for potential overlap with another object. When it overlaps, it will re-randomize its orientation for a predefined amount of time. Subsequently, 2D coordinates of the objects are obtained, processed, and filtered based on the area seen by the camera perspective. Following this, the 2D coordinates are saved in COCO format, and corresponding images from the camera perspective are then stored. Finally, the configurations of the object, material, light, and camera are reset to their initial values, and the COCO format for annotation is generated.

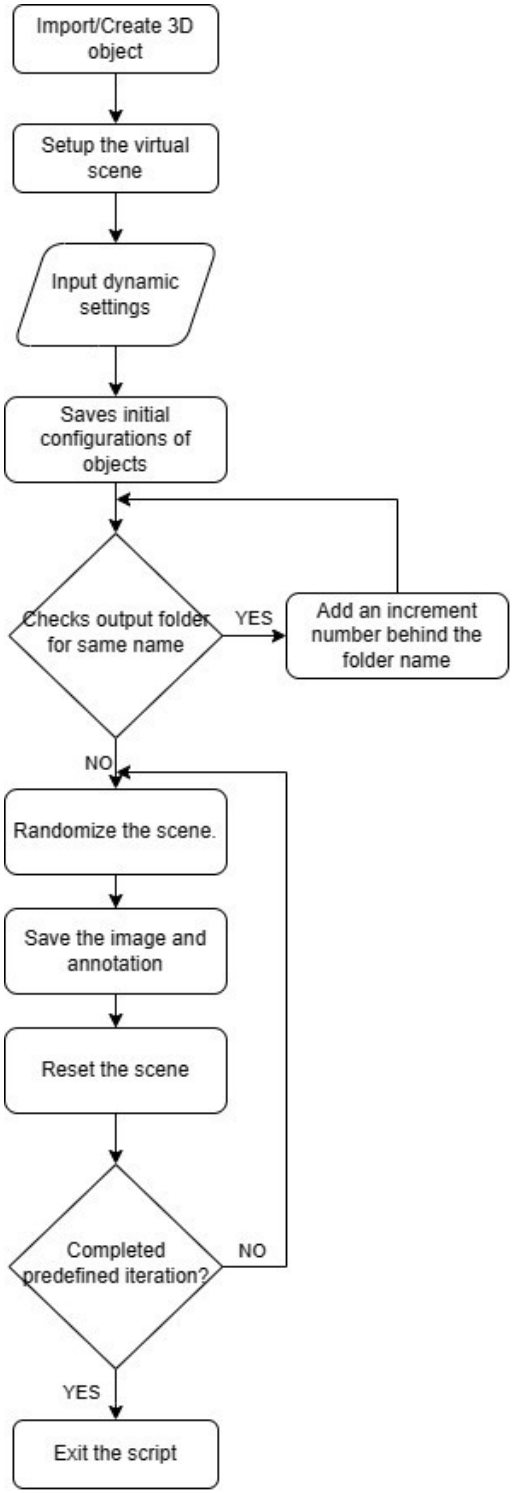

Figure 1: Flowchart of generating synthetic datasets

1.3 Scene Recreation

To create the 3D objects for the workstation scene in Blender, the objects can be modeled in Blender, or be imported as a 3D object file like STL; enabling the use of computer-aided

design software like Fusion360 or Inventor to create the object or use existing 3D objects found in online websites.

Once the 3D objects are in Blender, textures and materials can be applied to achieve a lifelike representation of the scene. For example, incorporating textures from real-life images into the object or customizing the material with different surfaces ranging from RGB (Red Green Blue) colors to the roughness of the object. Following this, the addition of light sources can be applied to create realistic lighting and shadow effects on the objects. Then. cameras are strategically placed to simulate the robot's perspective, providing different angles for image capture to build a comprehensive dataset.

Figure 2a shows a screenshot of the scene for the Cavity workstation in Blender while Figure 2b shows an image captured by the capture for the Cavity workstation scene in Blender.

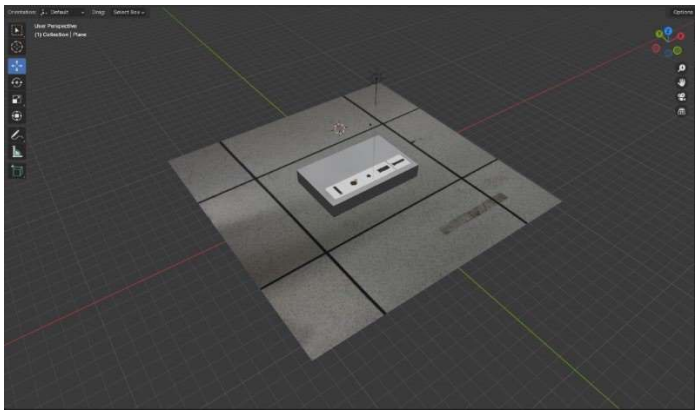

Figure 2a: The Cavity workstation recreated in Blender

Figure 2b: An image captured by the camera for the Cavity workstation scene in Blender

1.4 Bounding Box Coordinates

After setting up the scene, a Python script is employed in Blender to execute various actions, ranging from image capture to obtaining object coordinates as shown in Figure 3.

Figure 3: An annotated image of the Cavity dataset viewed through Roboflow

The Python script will first determine the object's coordinates relative to the camera's perspective and resolution before saving the annotation. Initially, the script will obtain information about the object, like its orientation, position, and scale in the 3D space. Subsequently, this information is processed to obtain the 3D coordinates of the objects. The 3D

coordinates are with reference to the world; thus, they need to be transformed into the 2D screen coordinates of the camera. Finally, the screen coordinates are normalized into the camera's pixels, where further processing is performed to obtain the bounding box of the object, as illustrated in Figure 4.

```
obj = bpy.data.objects.get(obj_name)

# Get the object's mesh data
mesh = obj.data

# Get the world matrix of the object
object_matrix_world = obj.matrix_world

# Get the 3D coordinates of each vertex in the object's mesh
vertex_coordinates = [object_matrix_world @ vertex.co for vertex in mesh.vertices]

# Convert world coordinates to 2D screen coordinates
screen_coordinates = [bpy_extras.object_utils.world_to_camera_view(scene, camera, coord) for coord in vertex_coordinates]

# Get the resolution of the render output
render_resolution_x = bpy.context.scene.render.resolution_x
render_resolution_y = bpy.context.scene.render.resolution_y

# Convert screen coordinates to pixels
pixel_coordinates = [(round(coord.x * render_resolution_x), round((1 - coord.y) * render_resolution_y)) for coord in screen_coordinates]
```

Figure 4: A code snippet that obtain the 2D bounding box coordinate of the object

However, when the script retrieves the bounding box coordinates of objects, it captures the coordinates of the objects and portions of objects that extend beyond the camera's field of view as shown in Figure 5.

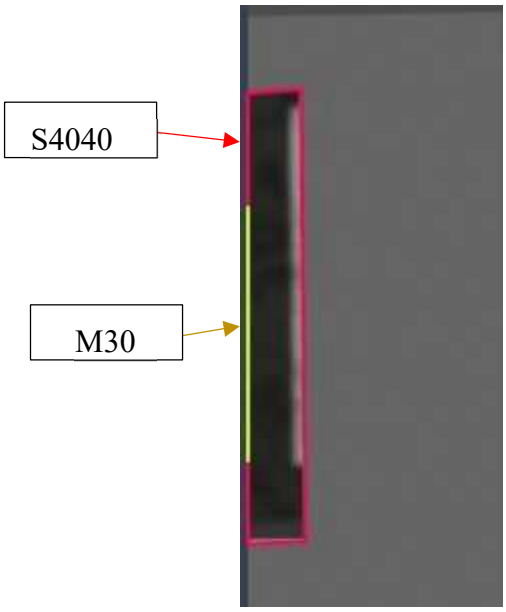

Figure 5: A magnified image of Figure 3

To address this, the coordinates needed to be limited accordingly to the pixel size of the image's boundaries. By imposing this constraint, the bounding box area can be calculated accurately, serving as a threshold for filtering out annotations that contain irrelevant objects annotations that are too small to be reliably recognized. Figure 6 shows a snippet of the code, which illustrates the process of capping the coordinates at 640 before calculating the area of the object. With the coordinates capped at 640, the area of the object outside of the camera's perspective will be set to zero, ensuring that only the portion of the object within the camera's view is retained.

```
for i, (x, y) in enumerate(zip(list_x, list_y)):
    if x > 640:
        list_x[i] = 640
    if y > 640:
        list_y[i] = 640
    if x < 0:
        list_x[i] = 0
    if y < 0:
        list_y[i] = 0

min_x, max_x, max_yx, min_yx = list_x
min_y, max_y, max_xy, min_xy = list_y

area = (max_x - min_x) * (max_y - min_y)
```

Figure 6: A snippet of the script that limits the bounding box coordinates to the camera resolution

The area of the object, when fully captured by the camera, is determined by repositioning the camera above the object at the start of the program execution to acquire its bounding box area. Following this, the minimum required area for the object is computed by multiplying the predefined area threshold by the bounding box area (as shown in Figure 7). This minimum area serves as a benchmark for comparison with the obtained bounding box areas. Any bounding box areas falling below this minimum value are filtered out, effectively eliminating annotations corresponding to objects deemed too small for reliable recognition.

```
        for i, object in enumerate(objects):
            #Move the camera
            move_camera("Camera", "above_object", object)

            #Obtain cooridnates of the object
            min_x, min_y, max_x, max_y, min_xy, min_yx, max_xy, max_yx =
annotation_coordinates(scene, camera, object)

            #Find the area then the area threshold
            area = (max_x - min_x) * (max_y - min_y)
            area_th.append(area * area_threshold)

            #Restore object initial position
            move_object(object, initial_location_[i], initial_rotation_[i])
            camera.location = initial_camera_location
            camera.rotation_euler = initial_camera_rotation
            print(f"{object} area is {area_th[i]}")
```

Figure 7: A snippet of the code to calculate the area threshold

1.5 Object Frame

In scenarios where objects possess unique shapes or only specific parts that need annotation, a frame can be generated to represent the area to be annotated. By using CAD software such as Fusion360, a frame which is a thin plane object can be created and placed at the location of where annotation is needed (as shown in Figure 8). By exporting the frame design and importing it into Blender, the frame will be relative to the object position. Thus, when both the object and frame share the same position and orientation values in Blender, they will be aligned.

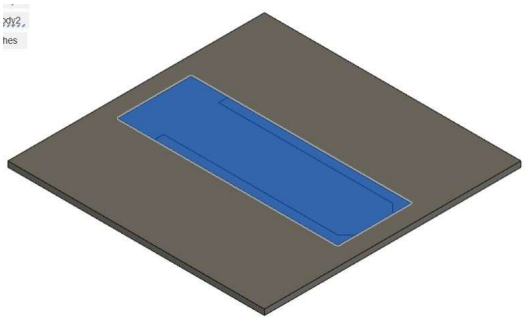

Figure 8: The frame of the M20_100 in Fusion360

When the frame is imported to Blender, its color can be set to transparent to prevent it from affecting the image collected. Furthermore, the program has a function that copies the object's position and rotation and applies these values to its respective frame, where the frame's name will have "_frame" appended to the original object name (as shown in Figure 9). This function effectively moves the frame along with the object, streamlining the annotation process (as shown in Figure 10).

```
# Check if the frame should be enabled
if enable_frame == 1:
    frame = obj + "_frame"
    obj_frame = bpy.data.objects.get(frame)

    if obj_frame is None:
        print(f"Frame not found: {frame}")
        return
else:
    obj_frame = None

if obj_ is None:
    print(f"Object not found: {obj}")
    return

# Move the object and its frame based on the provided location
if type(location) != int:
    obj_.location = location
    if enable_frame == 1:
        obj_frame.location = location
        obj_frame.location.x += offset_frame_x
        obj_frame.location.y += offset_frame_y
        obj_frame.location.z += offset_frame_z

# Rotate the object and its frame based on the provided rotation
if type(rotation) != int:
    obj_.rotation_euler = rotation

    if enable_frame == 1:
        obj_frame.rotation_euler = rotation

type_rotation = type(rotation)

bpy.context.view_layer.update()
```

Figure 9: A snippet of the code to move the frame to the object

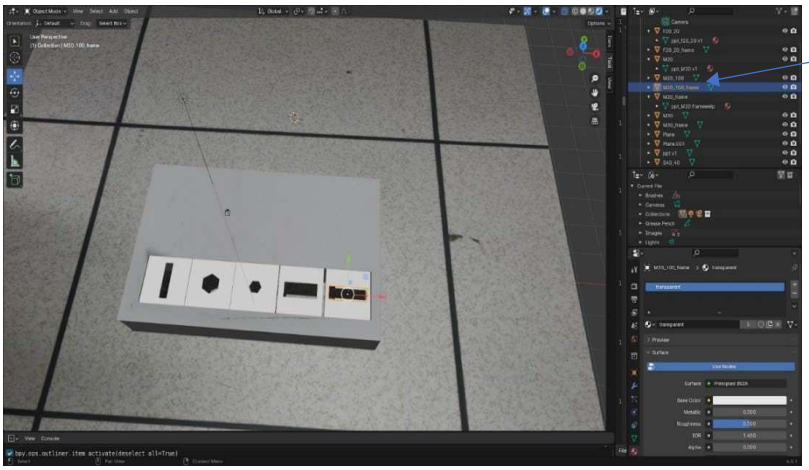

Figure 10: Selecting M20_100_frame in Blender

1.6 COCO Format

Different models, such as object detection and segmentation, often require distinct annotation formats—bounding box and segmentation, respectively. It is essential to use a format that accommodates both requirements and thus, the COCO format was chosen. The COCO format comprises three main components: categories, images, and annotations.

Categories define the name and corresponding ID of each category. Images include details such as height, width, date captured, file name, and a unique ID for each image. Annotations specify the object's details, including the area, and are linked to the respective category and image through their IDs (Manu, 2022). Additionally, the COCO format is supported in Roboflow [4], thus the synthetic data can be uploaded to visualize, edit, and merge new datasets as shown in Figure 11.

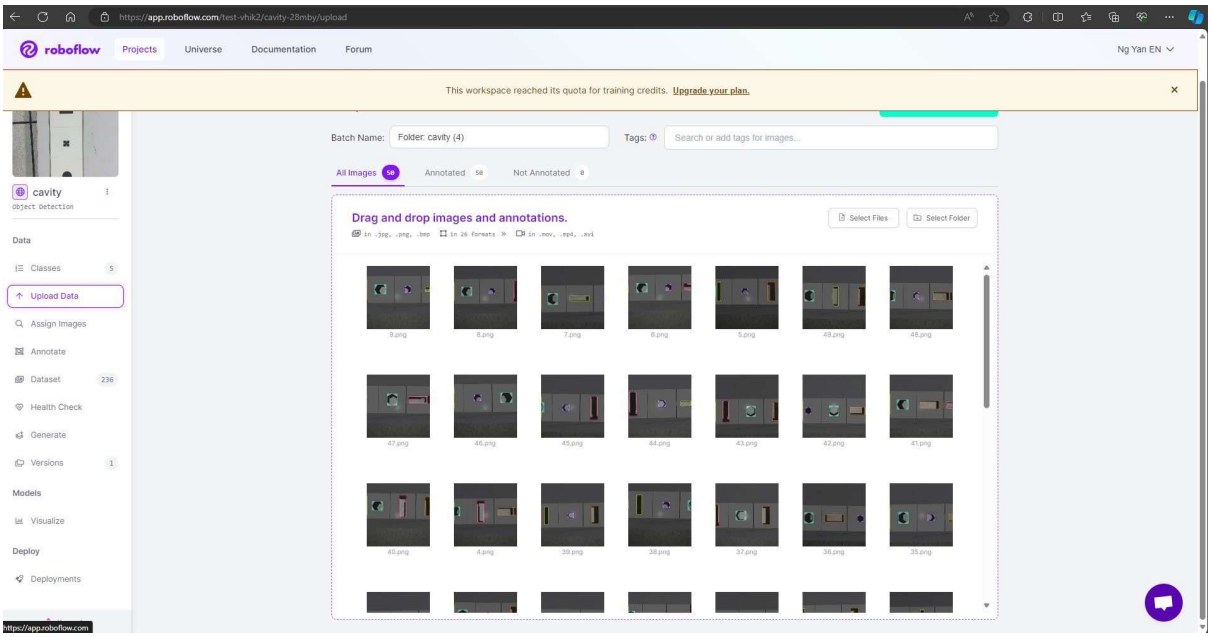

Figure 11: Uploading images into Roboflow

Figure 12a illustrates the code that saves the image to its corresponding ID, and Figure 12b shows the code that creates the categories ID. The categories ID and image ID are then used in the annotation to link the bounding box coordinate with the corresponding category and image, as depicted in Figure 12c.

```
index_ = index + 1
image_data = {
    "id": image_id_,
    "license": 1,
    "file_name": file_name_,
    "height": 640,
    "width": 640,
    "date_captured": formatted_time
}
```

Figure 12a: A snippet of the code that saves the image in COCO format

```
for index, objects_ in enumerate(objects):
    index += 1
    categories_data = {
                "id": index,
                "name": objects_,
                "supercategory": "none"
            }
    categories.append(categories_data)
```

Figure 12b: A snippet of the code that saves the category in COCO format

```
annotation_data = {
            "id": image_id_,
            "image_id": image_id_,
            "category_id": index_,
            "bbox": [min_x, min_y, max_x - min_x, max_y - min_y],
            "area": (max_x - min_x) * (max_y - min_y),
            "segmentation": [
                [min_x, min_y, max_x, min_y, max_x, max_y, min_x, max_y, min_x, min_y]
            ],
            "iscrowd": 0
        }
```

Figure 12c: A snippet of the code that saves the annotation data in COCO format

1.7 Output

The folders are organized based on their project name and chronological order of production for clarity and efficient handling, as shown in Figure 13. This enables easy handling of folders for the user to upload to Roboflow for visualization and editing of images and annotations before generating the dataset for model training.

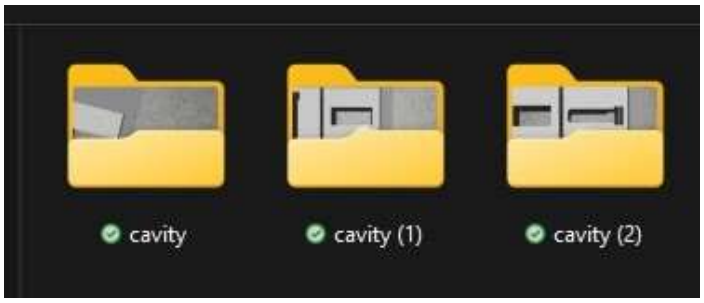

Figure 13: The output folder after generating the dataset for Cavity

The Python script initiates this process by retrieving the name of the Blender file. Subsequently, it checks for the presence of a folder with the same name in the directory where it is saving the images and annotations. If no folder with the same name exists, the script creates a new folder using the name of the Blender file. However, if a folder with the same name is found, the script appends a numerical suffix in the format '(i)' to the folder name, where 'i' represents the iteration needed to ensure a unique name, as shown in Figure 14.

```python
def check_and_rename_directory(base_path):
    # Get the current blend file path
    file_path = bpy.data.filepath

    # Extract the file name from the path
    file_name = bpy.path.display_name_from_filepath(file_path)

    target_path = os.path.join(base_path, file_name)

    if not os.path.exists(target_path):
        # Directory doesn't exist, no need to rename
        os.makedirs(target_path, exist_ok=True)
        return target_path

    # Directory already exists, let's find a new name
    index = 1
    while True:
        new_name = f"{file_name} ({index})"
        new_path = os.path.join(base_path, new_name)

        if not os.path.exists(new_path):
            os.makedirs(new_path, exist_ok=True)
            return new_path

        index += 1
```

Figure 14: The function of the Python script that creates the output folder

1.8 Dynamic approach

A large and diverse dataset, usually comprising hundreds or thousands of unique images, is crucial for training a robust deep-learning model. This ensures uniqueness, therefore enhancing the model's accuracy in various environments.

A dynamic approach is employed to ensure the efficient gathering of unique data automatically. Variations will be made in the object position, color, lighting strength, and camera position, as illustrated in Figure 15. By randomly changing these factors, the dataset will be augmented, thus preventing the model from memorizing specific instances and improving its adaptability to different scenarios.

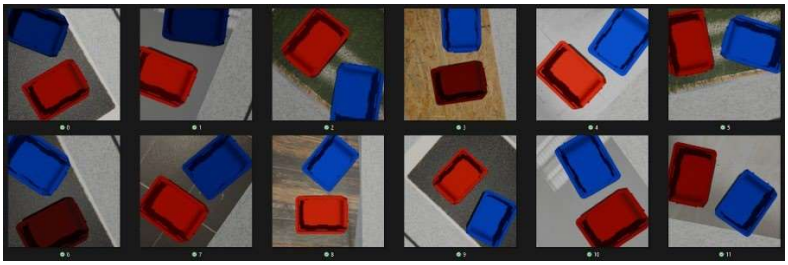

Figure 15: The output images for the Container synthetic dataset

1.8.1 Randomization of Camera Position and Rotation

A dedicated function was created to simulate the effect of the camera shaking during the robot's movement or its different viewpoints toward an object. The function first chooses a random value within a range from a predefine list. Then, it applies these values to the camera position and orientation, respectively as shown in Figure 16. Therefore, adding realism and variation to the dataset enhances the model's ability to handle diverse environmental conditions.

```
if (movement_type == "random"):
    # Generate random movement values within the specified ranges
    x = random.uniform(*camera_random_move_x)
    y = random.uniform(*camera_random_move_y)
    z = random.uniform(*camera_random_move_z)

    x_rad = radians(random.uniform(*camera_random_rotate_x))
    y_rad = radians(random.uniform(*camera_random_rotate_y))
    z_rad = radians(random.uniform(*camera_random_rotate_z))

    # Apply the random rotation to the object
    rotation = (x_rad, y_rad, z_rad)

    # Apply the random movement to the camera location
    camera.location.x += x
    camera.location.y += y
    camera.location.z += z
    camera.rotation_euler.x += x_rad
    camera.rotation_euler.y += y_rad
    camera.rotation_euler.z += z_rad
```

Figure 16: A snippet of the function to randomly move the camera

1.8.2 Randomly Swaps Object Location

While randomly moving and rotating the camera along the axis to simulate various viewpoints, another approach was also adopted to introduce diversity in the dataset; instead of moving the objects along the axis, where it will be hard to configure as the objects will likely overlap each other. The objects' positions are swapped with one another (as shown in Figure 17), thus providing an additional layer of variability and contributing to a more comprehensive training dataset for the model.

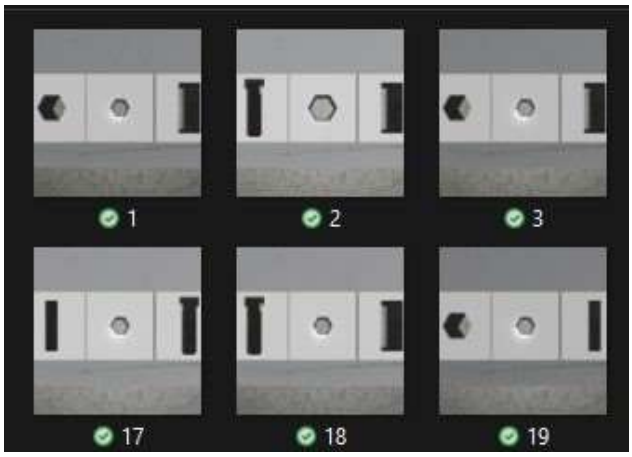

Figure 17: A screenshot of an output that randomly swap 2 object position

1.8.3 Randomly Rotates the Objects

Given the requirement for precision in both rotation and position, during scenarios like cavity placement or object picking, it becomes crucial to have a dataset that includes objects with various orientations. Hence, a function was implemented that randomly changes the object rotation, either within a predefine range or different specific values. As depicted in Figure 18, the function will first check if the list for the specific random rotation contains any value. If the list includes a value, the program will randomly choose a number from the list and apply it to the object's rotation. Otherwise, it will randomly select from a range of predefined values and apply it to the object rotation.

Furthermore, when rotating the object, it will check for overlap and re-randomize the object if it collides with another object.

```python
def rotate_object_randomly_x_axis(rotate_obj_):
    rotate_obj = bpy.data.objects[rotate_obj_]

    if len(object_specific_random_rotation) == 0:
        attempt = 0
        while True:
            # Generate a random angle in degrees
            random_angle = random.uniform(min_rotation, max_rotation)

            # Convert the angle to radians
            angle_radians = radians(random_angle)

            # Store the original rotation for later restoration
            original_rotation = rotate_obj.rotation_euler.copy()

            # Apply the random rotation to the object
            rotation = (original_rotation.x, original_rotation.y, angle_radians)
            move_object(rotate_obj_, 0, rotation)

            # Perform overlapping check
            if overlapping_object(rotate_obj_):
                # If there is a collision, restore the original rotation
                rotate_obj.rotation_euler = original_rotation
                bpy.context.view_layer.update()
                print("Collision detected. Restoring original rotation.")
                attempt += 1
                if attempt == 5:
                    break
            else:
                bpy.context.view_layer.update()
                break
    else:
        object_specific_random_rotation_ = random.choice(object_specific_random_rotation)
        # Convert the angle to radians
        angle_radians = radians(object_specific_random_rotation_)

        # Store the original rotation for later restoration
        original_rotation = rotate_obj.rotation_euler.copy()

        # Apply the random rotation to the object
        rotation = (original_rotation.x, original_rotation.y, angle_radians)
        move_object(rotate_obj_, 0, rotation)

        bpy.context.view_layer.update()
```

Figure 18: A function in the Python script to randomly rotate the object

Figure 19 shows the function to check for overlapping objects. It first obtains the vertice and the polygon of the object to create a BVH [5] (Bounding Volume Hierarchy) tree. Subsequently, it also gets the BVH tree of the other objects and uses the overlap function from math utils.bvhtree to detect overlapping of the objects. If overlapping is detected it will return True else, it will return False.

```python
def overlapping_object(target_obj_):
    # Get the geometry in world coordinates for the target object
    target_obj = bpy.data.objects[target_obj_]
    mat1 = target_obj.matrix_world
    vert1 = [mat1 @ v.co for v in target_obj.data.vertices]
    poly1 = [p.vertices for p in target_obj.data.polygons]

    # Create the BVH tree for the target object
    bvh1 = BVHTree.FromPolygons(vert1, poly1)

    for obj in objects:
        if obj != target_obj_:
            print(str(target_obj) + "-" + str(obj))
            temp_obj = bpy.data.objects[obj]
            mat2 = temp_obj.matrix_world

            # Get the geometry in world coordinates for the temporary object
            vert2 = [mat2 @ v.co for v in temp_obj.data.vertices]
            poly2 = [p.vertices for p in temp_obj.data.polygons]

            # Create the BVH tree for the temporary object
            bvh2 = BVHTree.FromPolygons(vert2, poly2)

            # Check for overlap
            if bvh1.overlap(bvh2):
                return True  # Return True if overlap is detected

    return False  # Return False if no overlap is detected
```

Figure 19: A function in the Python script to check for overlapping objects

1.8.4 Randomly Change the Lighting

The lighting conditions when the robots are operating outside the lab can be inconsistent and can be influenced by external factors such as sunlight or clouds covering the sun. Hence, a function was created to simulate it in a virtual environment where it will randomly change the lighting properties - intensity, diffuse, and specular; to different intensities. The program will first obtain a random number within a list and change it to its corresponding parameter, as shown in Figure 20.

```python
def random_light(lighting_list):
    for i, lighting in enumerate(lighting_list):
        light = bpy.data.objects.get(lighting)

        random_intensity_ = random.uniform(*random_intensity[i])
        # Set the random intensity to the light
        light.data.energy = random_intensity_

        random_diffuse_ = random.uniform(*random_diffuse[i])
        # Set the random intensity to the light
        light.data.diffuse_factor = random_diffuse_

        random_specular_ = random.uniform(*random_specular[i])
         # Set the random intensity to the light
        light.data.specular_factor = random_specular_
```

Figure 20: A function in the Python script to randomly adjust the lighting configuration.

1.8.5 Randomization of Material

When working with arbitrary surfaces and various lighting situations, having diverse datasets with different object colors and image textures can enhance the model's robustness. To emulate this in Blender, a function was created to access the Node tree[1] of the material before randomly changing it to red, green, and blue values within a range of values, as shown in Figure 21.

```
def random_material(material_list, material_list_type, image_list_):
    if (image_list_[0] == "random"):
        filter_list = image_list_[1:]

    for i, (material, type_) in enumerate(zip(material_list, material_list_type)):
        material_ = bpy.data.materials.get(material)

        if (type_ != 'image'):
            red_value = random.uniform(*red_range[i])
            green_value = random.uniform(*green_range[i])
            blue_value = random.uniform(*blue_range[i])

            if material_ is not None and material_.node_tree is not None:
                material_.use_nodes = True
                # Iterate through all nodes in the material's node tree
                tree = material_.node_tree.nodes[type_]

                tree.inputs[0].default_value = (red_value, green_value, blue_value, 1.0)

                # Set the new color
                material_.diffuse_color = (red_value, green_value, blue_value, 1.0)
```

Figure 21: The snippet to randomly change selected material's configuration

In addition to altering color, the material's texture can also be randomly modified by selecting an image currently active in the Blender session. The program begins by loading images from a designated folder into Blender. Subsequently, it retrieves a list of active pictures in the current Blender session and randomly picks an image from this list. If the selected image is on the filter list, it reiterates until it finds an unfiltered image. Following this, the function accesses the material's Node tree [1] and changes its image, as depicted in Figure 22

```
    elif (type_ == 'image'):
        # Get the material by name
        active_material = bpy.data.materials.get(material)

        # Enable use_nodes for the material
        active_material.use_nodes = True


        if (image_list[0] == "random"):
            # Get a list of all images in the current Blender session
            image_list_ = bpy.data.images
            image_list_len = len(image_list_) - 1

            while(1):
                random_image_index = random.randint(0, image_list_len)
                random_image = image_list_[random_image_index].name

                if(random_image != "Render Result" and not random_image in filter_list):
                    break

            for node in active_material.node_tree.nodes:
                if node.type == 'TEX_IMAGE':
                    image_texture_node = node
                    break
            image_texture_node.image = image_list_[random_image_index]
```

Figure 22: A snippet of the function to randomly select image for the material

1.9 Results

Two datasets were created: one using real-life images and the other using synthetic images. Each dataset contains 500 annotated images. YOLOv8-n was used to train models on these datasets individually and on a combined dataset. These models were then used to predict the same batch of images, and the results were recorded. Chart 1 shows the outcomes of the three models. The model trained on the combined dataset was significantly more accurate, outperforming the models trained on the real-life and synthetic datasets by 29% and 13.8%, respectively.

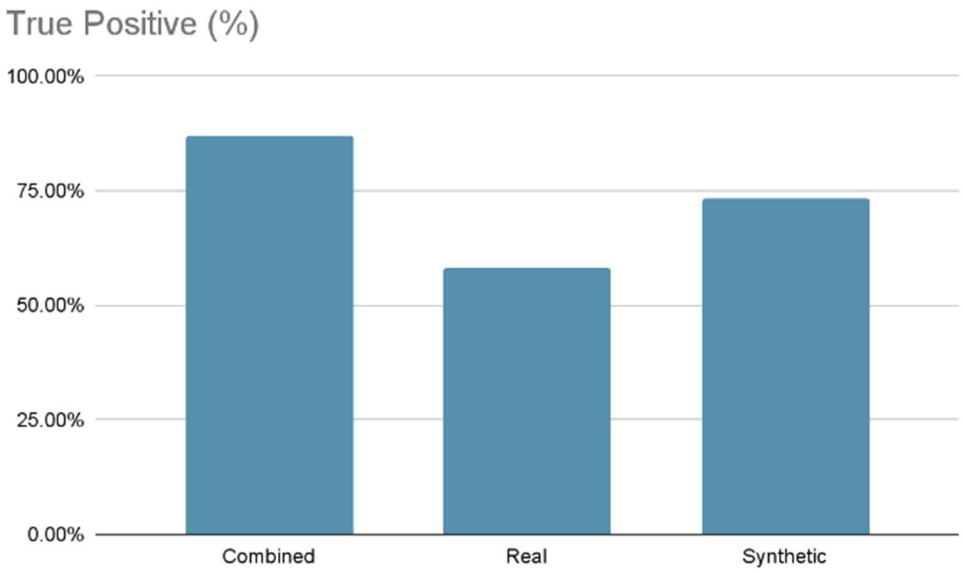

Chart 1: Results of Combined, real, synthetic dataset accuracy+

Overall, synthetic datasets can easily diversify data and efficiently generate large amounts of annotated images. However, they cannot fully replicate real-life images and their effectiveness depends heavily on the accuracy of the simulated environment. By combining the strengths and addressing the limitations of both synthetic and real-life datasets, a better model can be created.

Reference

Abraham, R., 2021. Synthetic Data for Computer Vision: Have you given it a thought? [online]. Available at: https://medium.com/geekculture/synthetic-data-for-computer-visionhave-you-given-it-a-thought-b775dfb3b428 (Accessed: 6 December 2023).

Chillingworth, A., 2023. The Pros & Cons of Creating 3D Content With Blender Software [online]. Available at: https://www.epidemicsound.com/blog/blendersoftware/?_us=adwords&_usx=11303056710_&utm_source=google&utm_medium=paidsearch&utm_campaign=11303056710&utm_term=&gclid=CjwKCAiA75itBhA6EiwAkho9exSCDo-i9HVYUQoSt2MUoHUsKHsNxomuhirhVpZZR0vwsxz5MvE7GhoCZdgQAvD_BwE (Accessed: 6 December 2023).

Manu, 2022. COCO format , what and how. [online]. Available at: https://medium.com/@manuktiwary/coco-format-what-and-how-5c7d22cf5301 (Accessed: 8 December 2023).

Appendix

Node tree [1] – It contains the parameter of the material, which can be changed to configure the properties of it.

Epoch [2] - The number of times a deep learning model iterates over the entire training dataset during the training process.

mAP50-95 [3] - The mean Average Precision computed at different intersection-over-union (IoU) thresholds ranging from 0.5 to 0.95, typically used to indicate the accuracy of the deep learning model.

Roboflow [4] – A website designed for uploading images or datasets to create or edit annotation. It has an augmentation function that can apply adjustments to the dataset by altering the brightness, saturation and more. Additionally, users can export their annotated dataset in various formats like YOLOv8, YOLOv5, COCO and more.

BVH [5] - The BVH algorithm is a recursive algorithm that parses the space of every object and assigns them to a particular node in the binary tree. The algorithm recursively analyzes each node until each node contains two objects most likely to collide.