# 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.

## 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).

## Methodology

Figure A1 depicts the flowchart for the synthetic dataset generation process. Initially, 3D objects are either imported into Blender or created using it. By repositioning these objects, the workstation of the arena is replicated, allowing users to configure the number of images and the diversity of the dataset. Before randomizing the object, material, light, and camera properties, the program saves the initial settings at the start of execution. Next, the program names the output directory based on the project’s name, where annotations and images are stored. To prevent data overwrites, it checks for repeated naming and appends an incremental number (e.g., ‘folder\_name (1)’ and ‘folder\_name (2)’). During each iteration, the script randomly adjusts light intensity, object position, color, material texture, and camera’s position and rotation. Additionally, it checks for potential overlap when objects change rotation, re-randomizing them if necessary. The 2D coordinates of the objects are obtained, processed, and filtered based on the camera perspective. These coordinates are then saved in COCO format, along with corresponding images. Finally, the object, material, light, and camera configurations are reset to their initial values, and the COCO annotation format is generated.

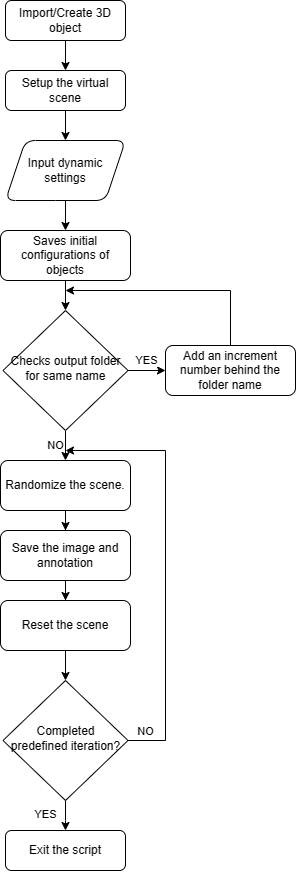


Figure a1: Flow chart of generating a synthetic dataset

## 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 a2a shows a screenshot of the scene for the Cavity workstation in Blender while Figure a2b shows an image captured by the capture for the Cavity workstation scene in Blender.

| A video game of a video game  Description automatically generated  Figure a2a: A screenshot of the scene for the Cavity workstation recreated in Blender | A close-up of a sign  Description automatically generated  Figure a3: An image captured by the camera for the Cavity workstation scene in Blender |
| --- | --- |

## 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.

Figure a4 illustrates the configuration of the camera resolution within the Python script before rendering and capturing the scene. To determine the object's coordinates relative to the camera perspective and resolution, the initial step involves obtaining the 3D coordinates of the object. Subsequently, these coordinates are transformed from world coordinates to 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 a5.

A screen shot of a computer program

Description automatically generated

Figure a4: The function in the Python script to capture the scene with the camera

A screenshot of a computer program

Description automatically generated

Figure a5: The function in the Python script to obtain the 2D bounding box coordinate of the object

However, when the Python script is used to retrieve 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 with values over the camera dimension. To address this, it is necessary to cap these coordinates to align them with the boundaries of the camera pixel range. 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 a9 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.

A screen shot of a computer code

Description automatically generated

Figure a9: The portion of the Python script to filter out the bounding box

To determine the area of the object when fully captured by the camera, the Python script (as shown in Figure a10) starts by repositioning the camera above the object to acquire its bounding box. Following this, the minimum required area for the object is computed by multiplying the predefined area threshold by the bounding box area. This minimum area serves as a benchmark for comparison with the obtained bounding box areas (as shown in Figure a11). Any bounding box areas falling below this minimum value are filtered out, effectively eliminating annotations corresponding to objects deemed too small for reliable recognition.

A computer screen shot of a code

Description automatically generated

Figure a10: A snippet of the script to move the camera above the object

A screen shot of a computer program

Description automatically generated

Figure a11: The snippet of the Python script to obtain the full area of the object

## Object Frame

In scenarios where objects possess unique shapes or only specific parts needs annotation, a frame can be generated to represent the object's bounding box. Fusion360 proves useful for creating a thin plane object, referred to as the frame, within the object file. This frame is then relocated to the desired position for the bounding box (as shown in Figure a12). By exporting the frame design and importing it into Blender, alignment with the object is achieved when both share the same position values.

A blue rectangular object with a rectangular object in the middle

Description automatically generated

Figure a12: The frame together with M20\_100 in Fusion360

To ensure the frame does not impact the collected images, the object's frame material in Blender is set to transparent. Furthermore, a Python script includes a function that copies the object's position and rotation and applies these values to a new object named with "\_frame" appended to the original object name (as shown in Figure a7). This function effectively moves the frame along with the object, streamlining the annotation process (as shown in Figure a8).

A screen shot of a computer program

Description automatically generated

Figure The Python function to move the frame together with the object

A computer screen shot of a white rectangular object

Description automatically generated

Figure a8: A screenshot of selecting the frame of M30\_100 in Blender

## 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, thus the synthetic data can be uploaded to visualize, edit, and merge new datasets as shown in Figure a12..0.

Figure a12a illustrates the code that saves the image and annotation of the bounding box and segmentation, while Figure a12b shows the code that generates the COCO format before saving it as a JSON file as shown in Figure a12c.

A computer screen shot of a program code

Description automatically generated

Figure a12a: The portion of the Python script that saves the image and annotation data

A screen shot of a computer screen

Description automatically generated

Figure a12b: A function of the Python script that saves the annotation in COCO format

A computer screen with text and images

Description automatically generated

Figure a12c: The portion of the Python script that saves the COCO format as a JSON file

## Output

To facilitate the visualization and editing of images and annotations before generating the dataset for model training in Roboflow, it is crucial to organize them in folders for clarity and efficient handling as shown in Figure a13. 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 a14.

A screenshot of a computer

Description automatically generated

Figure a13: Folder saved after

A computer screen shot of a code

Description automatically generated

Figure a14: The function of the Python script create the folder

## Dynamic approach

In deep learning models, a large and diverse dataset, often comprising hundreds or thousands of unique images, is crucial for training a robust model. To ensure uniqueness and enhance model accuracy in various environments, a dynamic approach is employed. This involves introducing variations in object position, color, lighting strength, and camera position, as illustrated in Figure a15. By randomly changing these factors, the dataset is augmented, preventing the model from memorizing specific instances and improving its adaptability to different scenarios.

A collage of a screenshot of a video game

Description automatically generated

Figure a15: The output images for the synthetic dataset

### 2.7.1 Randomization of Camera Position and Rotation

To simulate the effect of the camera shaking during the robot's movement or its different viewpoints toward an object, a dedicated function was created. As illustrated in Figure a16, this function introduces random rotations within specified ranges for the x, y, and z-axis, stimulating the dynamic nature of real-world scenarios. Additionally, the function incorporates random translations along the x, y, and z axes, creating variations in the camera's position. This simulation adds realism to the dataset, thus enhancing the model's ability to handle diverse environmental conditions.

A screen shot of a computer program

Description automatically generated

Figure a16: A snippet of the function to move the camera

### 2.7.2 Randomly Swaps Object Location

While randomly moving and rotating the camera along the axis to simulate various viewpoints, another approach was also adopted for introducing diversity in the dataset. Instead of physically moving the objects, the object's positions are swapped with one another (as shown in Figure a17). This provides an additional layer of variability, contributing to a more comprehensive training dataset for the model.

A screen shot of a computer program

Description automatically generated

Figure a17: A function in the Python script to randomly swap 2 object position

### 2.7.3 Randomly Rotates the Objects

Given the requirement for precision in both rotation and position, particularly evident during scenarios like cavity placement or object picking in competitions, it becomes crucial to have a dataset that includes objects captured from various angles. To achieve this, a dedicated function has been implemented. This function enables the random change in the object rotations, either within a specified range or different specific values (as shown in Figure a18). To prevent overlapping of the object, another function was created to check for overlapping objects. The function first obtains the vertice and the polygon of the object to create a BVH (Bounding Volume Hierarchy) tree. Subsequently, it also obtained the BVH tree of the other objects and used the overlap function from mathutils.bvhtree to detect overlapping (as shown in Figure a19). If the object is detected, the Python script will rotate the object again. By incorporating such variability, the dataset becomes more comprehensive, ensuring the model's robustness in handling diverse and real-world scenarios.

A screenshot of a computer program

Description automatically generated

Figure a18: A function in the Python script to randomly rotate the object

A computer screen shot of a black screen

Description automatically generated

Figure a19: A function in the Python script to check for overlapping objects

### 2.7.4 Randomly Change the Lighting

The lighting conditions during the competition are inconsistent and can be influenced by external factors such as sunlight or cloud covering the sun. Therefore, it becomes crucial to create a dataset that encompasses various lighting scenarios. This can be achieved by incorporating a function in the Python script to randomly change the lighting intensity, diffuse, and specular parameters within a range of predefined values (as shown in Figure a20). This diversity is essential for simulating different lighting conditions and ensuring the robustness of the system against real-world variations in illumination.

A screen shot of a computer code

Description automatically generated

Figure a20:  A function in the Python script to randomly adjust the lighting configuration.

### 2.7.5 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. Thus, a function was created in the Python script that randomly adjusts the red, green, and blue components of the material within the predefined ranges (as shown in Figure a21).

A screen shot of a computer code

Description automatically generated

Figure a21: The function in Python 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. The function then retrieves a list of active images in the current Blender session and randomly picks an image from this list. If the selected image is on the filter list, the random image selection process is repeated until it finds an unfiltered image. Following this, the function accesses the material and changes its image through the node tree, as depicted in Figure a22.

A screen shot of a computer code

Description automatically generated

Figure a22: A snippet of the function to randomly select image for the material

### 3. 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-vision-have-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/blender-software/](https://www.epidemicsound.com/blog/blender-software/?_us=adwords&_usx=11303056710_&utm_source=google&utm_medium=paidsearch&utm_campaign=11303056710&utm_term=&gclid=CjwKCAiA75itBhA6EiwAkho9exSCDo-i9HVYUQoSt2MUoHUsKHsNxomuhirhVpZZR0vwsxz5MvE7GhoCZdgQAvD_BwE)(Accessed: 6 December 2023).