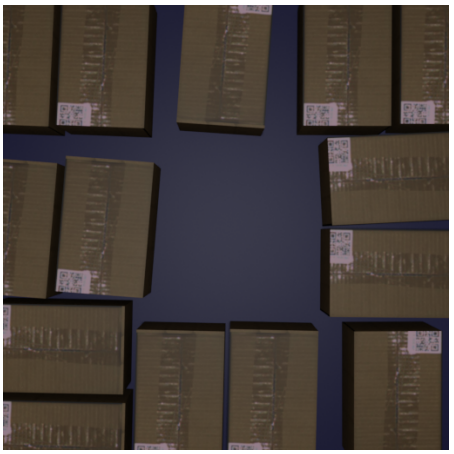
SYNTHETIC DATA GENERATION PIPELINE



Team Members

Richard Echegaray

Andrew Fyffe

Alex Carbo

Maxwell Yang

Project Sponsors @ DaoAI

Mr. Xiaochuan Chen - CTO

Dr. Sina Salsabili - ML Engineer

Project 2162

Engineering Physics 479

Engineering Physics Project Laboratory

The University of British Columbia

April 3rd, 2022

Executive Summary

DaoAI Robotics is a Vancouver-based company which aims to make robotic vision a critical component in automation by providing turn-key hardware and software solutions for bin-picking robots. Bin picking robots are often used in industrial settings such as manufacturing or e-commerce to streamline the packaging process. DaoAI has developed custom 3D cameras which produce high quality precision depth data and RGB data with a computer vision pipeline to identify objects in the image feed. DaoAI hopes to accelerate the training and deployment of robots in real scenarios by training their machine learning models on synthetic data. Synthetic data pipelines can produce images at a fraction of the time and cost to improve efficiency and model performance.

When collecting real training data, it can be quite cumbersome as one has to manually label each object and vary the objects' positions so you can have a varied set of data. This is why creating simulated training data that accurately mimics the real training data is much more convenient. The goal of this project is to provide DaoAI with an in-simulation computer vision data generation pipeline that takes in a CAD object and outputs:

- images with RGB and depth data of CAD objects in different configurations (varying orientation, relative position of objects, illumination, background, etc.)
- labels pixels according to their object class
- labels pixels according to their instance

In tandem with this goal, we must assess the efficacy of our data by:

- training a model that performs both semantic and instance segmentation on the in-sim RGB and depth images.
- assessing the model's performance on in-real data and evaluating the training efficiency of the in-real model when boosted using in-sim data.

For our pipeline, DaoAI wanted us to focus on two specific datasets: Ping Pong Dataset and Package Dataset. Our final product was able to meet all the goals listed above. With the model's performance, for the Ping Pong dataset we achieved an AP of 100, and for the Package dataset we achieved an AP of 73.9.

Table of Contents

Table of Figures

Introduction

Our project sponsors are Xiaochuan Chen and Dr. Sina Salsabili from DaoAI, a market-leading provider of 3D Vision and AI Systems for robot guidance. The team at DaoAI offers a range of high-quality cameras, as well as world-class vision software that work together to complete industrial pick-and-place tasks.

Their work with computer vision has many implications in the field of training bin-picking robots and production line processes. In most cases, training these robots to learn to pick up foreign objects is a very time-consuming and thus expensive process. In order to facilitate this process, DaoAI aims to improve their workflow of training and deploying robots in real pipelines. In light of this, our solution is to develop a simulation pipeline that will accelerate the training and deployment of robots in real pipelines.

DaoAI's end goal is for their workflow is as follows:

1) A customer would upload a set of 3D parts that need to be picked on DaoAI's web portal.
2) DaAIi pre-trains a model to segment and determine the pose of the parts in simulation
3) This model will then be refined by additional training in real, but because it has been pre-trained in-sim it will require much less training in-real, which will save them both time and money

Our solution pertains mainly to 1) and only the first part of 2), as our solution takes in a CAD Model as an input, and produces a set of synthetically generated (in-sim) data, with a specified amount of objects, and number of images.

Category	Core objectives
Input	● Input data must be a CAD Model ● There is a JSON config file that allows us to specify the amount pictures generated, camera placement, lighting, and what objects to be put through the pipeline as well
Data Processing	● Data to be generated from CAD Model using BlenderProc, and randomised via parameters specified in JSON config file
Model Training and Validation	● Model to be validated using the following metrics: ○ Intersection over Union ○ Average Precision
Output Data	● Data is output to a folder specified in the command line when the script is run ● All synthetically generated images are annotated as per the COCO annotation format[1]

Figure 1: Table of core objectives

The project was also originally geared towards working on three datasets: a ping pong ball dataset, box package dataset, and a tool package set. We later decided with our sponsors to narrow our scope down to just the ping pong ball and package dataset, to ensure that we could achieve higher scores on the data we generated from the package dataset.

Theory

Domain Randomization

Domain randomization is a technique used to simulate a sufficiently large amount of variations in simulated data such that real world data is viewed as simply another domain variation. In the project's context, we want our simulated data to mimic real world data as closely as possible so when we pre-train our model in-sim, the in-real training will not take long since our simulated data is strong enough. Domain randomization strengthens our simulated data by randomizing parameters such as:

- Lighting
- Objects: Types, Placement, Quantity
- Camera Placement
- Distortion: Blur, Tint, Hue

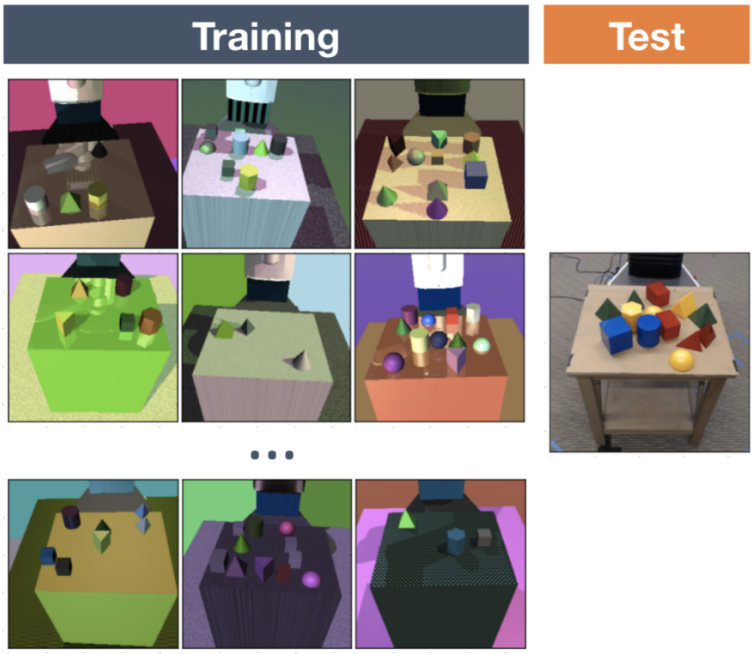


Figure 2: Sample image of training data having domain randomized parameters

Intersection Over Union

Intersection over union is a term used to describe the extent of overlap of two boxes. The greater the region of overlap, the greater the IOU. By two boxes, we are referring to one box being the predicted bounding box and the other being the true bounding box around the object. IOU is considered to be an industry standard in terms of validation metrics for object detection. The specific way to calculate intersection over union is by calculating the area of intersection of the two boxes over the area of the union of the two boxes.

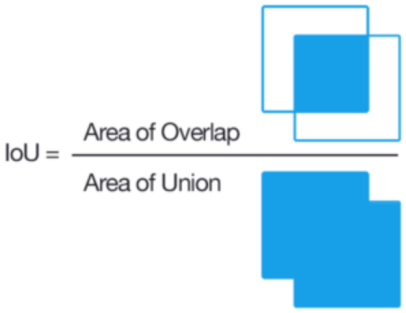


Figure 3: IOU definition and visual representation

Average Precision

Average Precision computes the average precision value over 0 to 1. Precision measures how accurate your predictions are as a percentage, so the number of true positives over total positives. Average Precision uses IOU to define a cutoff as to what is a true positive/false positive. Average Precision will be the primary metric we will use for evaluation of our synthetic data.

Solution Design

High Level Overview

Our solution can be described as a data pipeline consisting of two major components: the data processing stage and the model training and validation stage. The data processing stage handles the synthetic data generation where we generate labeled RGB-D images for the model to train on. The model training and validation stage is where we evaluate our synthetic data by training models using the synthetic data and assessing how they perform on the real data through validation metrics. Upon completion of the model training and validation stage, an iteration stage may be performed depending on the performance of our model.

For our data processing pipeline, there were two main frameworks we looked into using: BlenderProc and Unity Perception. Unity Perception leverages Unity, a popular 3D game engine, and allows us to generate images at scale with random sampling. We were able to apply randomization in aspects such as lighting, number of objects, background textures, etc. and we could run jobs in the cloud using Unity Simulation. Since our sponsors did not have any preference to what tools/technologies we used and we did not see a critical differentiator between BlenderProc and Unity Perception besides one using Blender and the other Unity, we decided to use the better supported software with more users. The BlenderProc community being more popular would help us in case there were any issues we had in using the software.

This leads us to describing the core of our data processing pipeline: BlenderProc, a data generation pipeline leveraging Blender for photorealistic rendering. BlenderProc has Python functionality built in such that we can run a Python script to construct a 3D scene, set up camera poses and render RGB-D images. In this script, we are able to implement domain randomization for similar aspects as stated with Unity Perception. In addition, the labels for the generated RGB-D images support COCO annotations which is a JSON structure dictating how labels and metadata are saved for an image dataset. The debug mode capability also lets us view the rendered scene in real time, letting us easily know where objects are specifically being placed in our container. BlenderProc was very easy to start using with us only having to download the software and run their quickstart.py script to get

familiar with how to use the software. This was a main factor into why we ended up going with BlenderProc over Unity Perception.

Data Pipeline Implementation

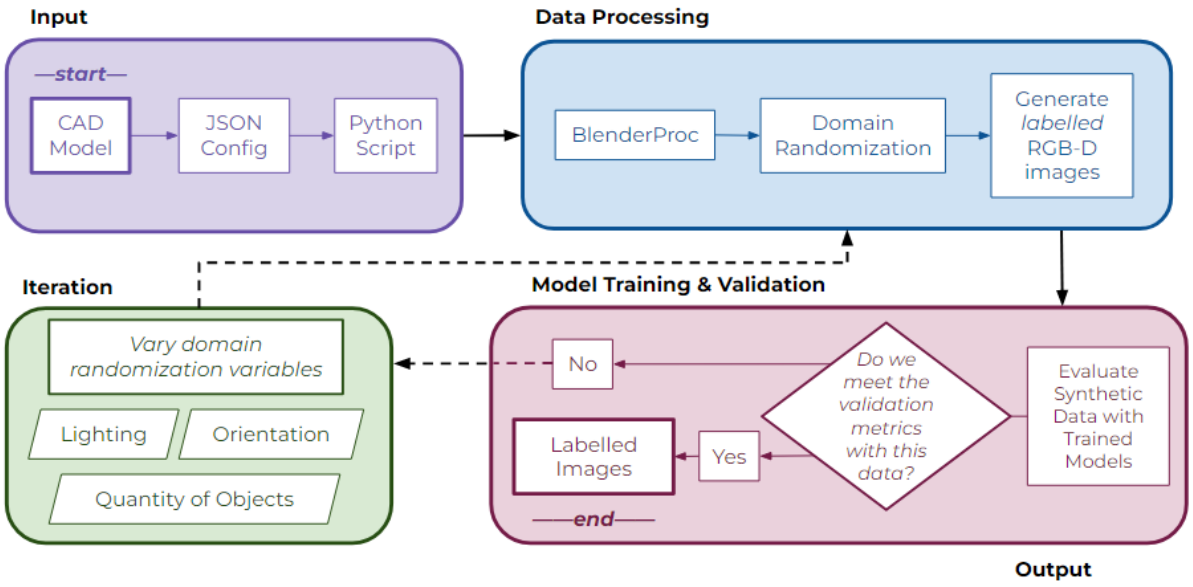


Figure 4: Overview of pipeline. The user inputs a CAD model, which is processed by BlenderProc, and physics simulation. The pipeline outputs a set of synthetically generated images, specified per the input JSON config file

The diagram above shows a comprehensive overview of the system's data pipeline. The following sections will cover the design of each component of the pipeline in more detail.

Input

The input to our data processing stage is centered around the CAD model of the object we want to generate data for. For the project context, the CAD model was either the ping pong ball or the package for the two corresponding datasets. With our input, the file path to the CAD model is referred to in our JSON configuration file. The JSON configuration file is used to customize different domain randomization variables such as lighting, object placement, number of objects, etc. and which objects we want to include in the generated

images. After the configurations have been put in place, we run a Python script to initiate the data processing pipeline.

Data Processing

BlenderProc

BlenderProc is the heart of our data generation pipeline and where the training images are created.

Blender is a popular 3D computer graphics software used in animation and modelling. It has features such as physics simulations, photorealistic lighting, and a Python interface.

BlenderProc is a framework built on top of Blender that allows for procedural generation of machine learning training images. By using the features of Blender listed above we can create high quality images that mimic real life images. BlenderProc essentially adds on to the existing features and allows for iterative image creation with randomised parameters.

The three main stages of BlenderProc are:

- **Scene Creation**
- **Physics Simulation**
- **Image and Label Creation**

Scene Creation

The first step of data generation is scene creation. In this step BlenderProc will build a scene according to the parameters that we provide it. The parameters include features such as:

- Object models

- Object placement
- Lighting conditions
- World conditions

The above parameters do not need to be exact numbers and can instead be ranges that BlenderProc will then randomly sample within to create varied images.

Physics Simulation

The next step is the physics simulation. In this step we utilise Blender's realistic physics simulation to allow the objects to rest into realistic positions. This improves the quality of our images because they mimic real situations more strongly.

After letting the objects fall BlenderProc has to meet either of two conditions to then move on to the next stage and then the next scene and image. Either all of the objects in the scene will be at rest for a controllable amount of time, or the simulation will reach a maximum breakout time where BlenderProc will extract the image from the scene regardless and move on to the next scene and image.

Figure 5: Example of a scene before and after the physics simulation.

Image and Label Creation

The final stage of the image generation is the creation of the RGB images and their labels. Since all of the objects are within Blender we know the exact position and orientation

of the objects unlike real life scenes. This allows us to create labels at a fraction of the time that it would typically take. The formats that BlenderProc can create are .hdf5 containers, COCO and BOP annotations. We have also created our own script to convert into a specific format for DaoAI.

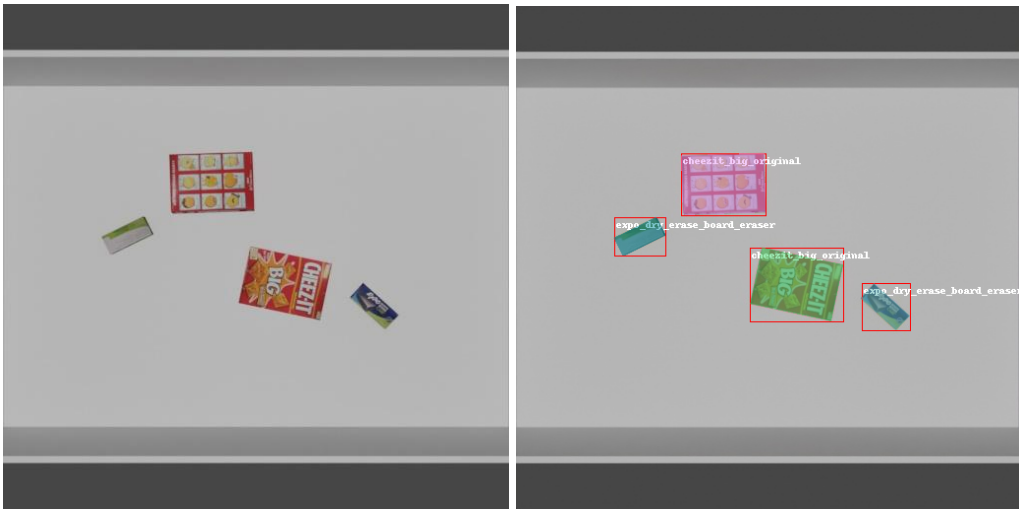


Figure 6: Example of a synthetic RGB image and the labels created with it

Model Training & Validation

The type of model we want to train is called a Mask - Region-Based Convolutional Neural Net (Mask-RCNN). There are a few layers to understanding how this type of model works. At the core, we have Convolutional Neural Networks, the base type of neural network used for image classification.

R-CNNs are used for object detection, and are designed to output bounding boxes around the objects of interest. Fast R-CNNs are an improvement on R-CNNs which improves the process of generating regions of interest by way of the Region Proposal Network (RPN). Finally, Mask-RCNNs modify the behavior of their predecessors by also outputting a “mask” - a shape attempting to fill in the object of interest - in addition to the bounding box, class, and instance labels.

To judge the quality of our synthetically generated data, we train one of these models, and test its performance in terms of Average Precision (AP) on a Test Dataset provided by DaoAI. For this task, we fine tune a pre-trained model, so as to not waste time training from scratch, which would take a long time.

Iteration

After training our model using the synthetic data, if the validation metrics (Average Precision) do not meet our expected criteria, we iterate on our data. Utilising domain randomization, we alter the parameters in our configuration file such as lighting, number of objects, object placement, etc. so we get more varied data. As an example, for our package dataset, we iterated on the method we used to place the packages numerous times to get better synthetic data. After altering these parameters, we run the Python script once again through our data processing pipeline and once again train and validate our model on the newly generated synthetic data. We continue this process of iteration until we meet our desired criteria for validation metrics.

Colab Notebook

We have provided a colab notebook: *DatasetUtils.ipynb*, which includes the code and workflow used in viewing and manipulating the datasets, translating DaoAI labels to standard Detectron2 dataset dicts, training and evaluating the Mask-RCNN.

The notebook provides functionality useful in working with the generated synthetic data, such as viewing labels for a given image, viewing the labels for a random sample of images from an arbitrary dataset, comparing predictions versus ground truth for a given model and image, as well as examples of how to use each.

Our pipeline generates COCO labels for our data. Detectron2 and other libraries contain many quality-of-life improving functions that support this format, and detectron2 can automatically convert its standard dataset dict format to COCO, so we wrote a function to convert the DaoAI format labels to detectron2 dataset dicts. The function *get_package_dataset* takes the base path of the package dataset and returns a function

which returns the dataset dicts when called, allowing it to be used in the custom dataset registration workflow.

Results

Ping Pong Dataset

Introduction

The first dataset that we worked with for DaoAI was referred to as the Ping Pong dataset. This dataset consisted of images of an orange ping pong ball in an industrial background with various random unlabelled objects. This dataset is typically used by DaoAI for calibration of their computer vision cameras. DaoAI provided us with 36 real images that we labelled by hand as our “real” dataset.

The images are fairly simple and were mainly used as a proof of concept for our data pipeline before we moved on to more difficult problems. Our goal was to first improve performance when training our model with real and synthetic data and then testing on real data. Next we would train with purely synthetic data and test with real images.

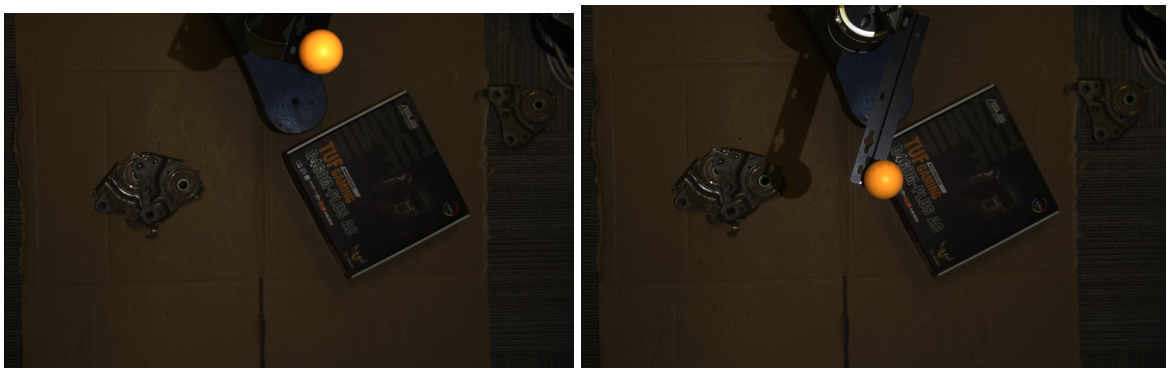


Figure 7: Example Images of the Ping Pong Dataset

Synthetic Images

Seen below are the results of our synthetic data generation pipeline with the Ping Pong dataset.

Figure 8: Example Images of the Synthetic Ping Pong Dataset

Performance Metrics

Below is a table of the results of our synthetic data. There are two different types of labels used in this testing. The first is “bbox” which is a bounding box around pingpongs defined by two points. The second is “segm” which is a segmentation border that labels individual pixels of the image as either a ping pong ball or not a ping pong ball.

(bbox, segm)	Train Simulated Data, Infer Simulated Data	Train Simulated and Real Data, Infer Real Data	Train Simulated Data, Infer Real Data
AP	(89.6, 97)	(100, 100)	(90, 90)
AP50	(100, 100)	(100, 100)	(100, 100)
AP75	(100, 100)	(100, 100)	(100, 100)

Figure 9: Bounding Box and Segmentation Metric Scores

Package Dataset

Intro

The second and last dataset that we worked with for DaoAI was referred to as the package dataset. These images and objects were much more complicated than the previous dataset and required many iterations of data generation before we obtained reasonable results.

The images consist of cardboard boxes in an industrial setting. Our goal was to label only the boxes on top that were “pickable” by a robotic arm. Similarly, DaoAI provided us with real labelled images to use as a testing set.

First Iteration

Figure 10: Example image from the first iteration of our synthetic package images

An example image from our first iteration of synthetic package data is seen above. We used very simple object placement and textures to test basic logic.

The main challenges this iteration had were:

- Poor textures
    - This was a problem that we had foreseen and was shown since the model struggled predicting the boxes because of the lack of tape

- Unrealistic placement
    - This was a problem that we did not expect, and was primarily because our object placement was not structured.

Second Iteration

Figure 11: Example image from the second iteration of our synthetic package images

Above is an example image from the second iteration. We addressed the texture problem by adding QR codes and a strip of grey that acted as tape. We also added box placement logic that created much more realistic placement. Surprisingly we found that disabling the physics simulation and allowing the boxes to float also created better images.

The box placement logic works as follows:

- Split the scene into a 3x3x2 grid.
- At each point, randomly place either 0,1, or 2 boxes
    - Include restriction such as: 2 boxes cannot be on top of 0
- Randomly translate boxes translation and rotation

The problems that we found with this iteration were:

- Tape was still not accurate.
    - We still would often predict on each half of the box around the tape

- Labelling tiny pieces of boxes
    - We were labelling every piece of a box that showed in the image, but that was not the way the real images were labelled which decreased performance

Final Iteration

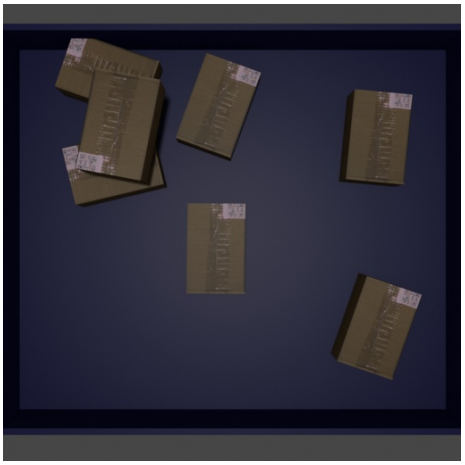


Figure 12: Example image from the last iteration of our synthetic package images

To address the previous iterations challenges we:

- Copied an image of a box to use as a texture for the boxes
    - Blender has a lighting simulation but without adding tiny folds to the tape to reflect light realistically the light would just appear as a circle
- Added much more variation in the images.
    - Some images were simple like seen above, and some were as complex as the second iteration
- Only labeled boxes in the top layer.

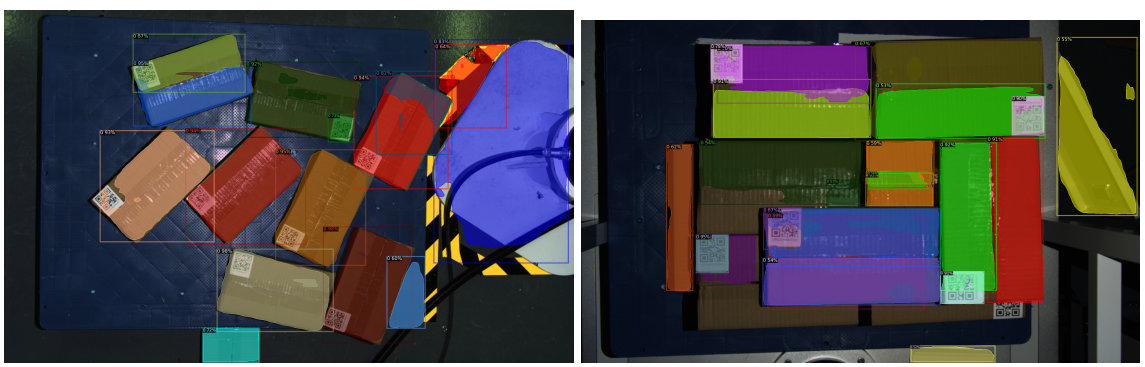


Figure 13: Example predictions on real images of DaoAI model trained with only synthetic images

As shown in the image above performance improved but was still mixed. The model performed very well with images where box placement was simple. In complex images this iteration still would often predict halves of boxes and also struggled to know which boxes were “pickable” as in the last iteration. See recommendations for further discussion.

Performance Metrics

As discussed previously, results were very mixed. Below are the results for training with synthetic data and testing with real images.

(bbox, segm)	Train Simulated Data, Infer Real Data
AP	(73.9, 73.1)
AP50	(89.2, 85.5)
AP75	(84.3, 83.0)

Figure 14: Performance metric results for training with synthetic data, testing with real data

In this case, when improving performance it is not useful to observe metrics such as average precision because they do not give insight as to how our data is failing which is why we did not create a full table of metrics. Instead, it is much more useful to observe predictions such as the images seen above.

Conclusions

Our solution meets all of the core objectives that we set with our sponsors. We are able to take in input CAD models for two different datasets and generate an arbitrary

amount of synthetic data that could be used to initially train bin-picking robots, thus increasing the speed and efficiency of the training process, as well as saving money.

The next stage of development should be focused on two things: one being expanding this pipeline to be more robust, i.e. working with a greater set of input cad models, and the other being user testing and feedback. As this project had an extremely short lifecycle, more time was allocated towards implementing functionality and system design. However, hands-on testing is still required to determine the effectiveness of this pipeline in our sponsors' workflow. User testing will also give insight into new features that could be added to future iterations of the pipeline.

The performance of the application is specified in the appendix, but requires improvement if larger datasets (in the order of millions of images) are desired. The majority of processing time in our application is spent in the rendering, and the physics simulation. The rendering works with the GPU, and the physics simulation works with the CPU. Recommendations for processing time are discussed in the next section. Python also operates with much more overhead than other languages, so our design decision to go with a rendering software such as Blender that works with Python could have potentially cost us some performance, in exchange for easy integration between BlenderProc and the Python pipeline. Thus, implementing this pipeline in another rendering software such as Unity (which runs with C# natively), or even implementing the pipeline in a different language altogether could yield performance improvements.

Recommendations

Pipeline

Containerizing the pipeline to allow for computation to perform on the cloud would greatly increase performance and usability. This is likely a later step that will come once interactivity is streamlined but it is the ultimate goal of the project.

Additionally, standardizing more features of the scripts to allow for interaction to only occur through the JSON configuration file would greatly increase usability. As of now, the configuration file has very limited control over the data that the scripts create and the data generation scripts are quite specific. Generalizing the scripts and increasing functionality would greatly increase usability and be a big step towards containerization.

The last recommendation for the data pipeline is a data management storage tool. Being able to keep track of what parameters were used for which data and also being able to mix and match batches of data for specific uses would also be another big step towards creating a fully automated pipeline.

Package Dataset

The package dataset currently has a few problems that need to be solved.

The first is the textures of the boxes. We have improved the textures of the boxes to a great degree but they still have room to grow. We only use a single texture and having multiple different box textures would improve model robustness. Additionally more domain randomization and possibly more data would help this problem considerably.

The next problem that needs to be solved is the poor labeling of boxes. The solution to this problem is not clear at the moment. It may be best to only consider labels above a certain area to solve this issue.

Appendix

Image Generation Benchmarking - RTX 3070, i7-9700K

Complexity / Number of Runs	1 Ping Pong Ball	1 Ping Pong Ball 1 Expo Marker 1 Cheez-it Box	1 Ping Pong Ball 10 Expo Markers 10 Cheez-it Boxes
1 image	0 min 10 **6 images per min**	0 min 11 **5.5 images per min**	0 min 16 **3.8 images per min**
10 images	0 min 28 **21.4 images per min**	0 min 33 **18.2 images per min**	1 min 17 **7.8 images per min**
100 images	3 min 33 **28.2 images per min**	4 min 20 **23 images per min**	12 min 18 **8.1 images per min**

Figure 15: Image generation benchmarking