Synthetic Waste Generator (SWaG) for Classification Training

Bachelor Thesis

Khalid El Haji, Hakan Ilbaş, Sergen Karpuz, Noah Posner, Victor Wernet

Faculty Of Electrical Engineering, Mathematics and Computer Science

Delft University Of Technology

TU Delft

Synthetic Waste Generator (SWaG) for Classification Training

Bachelor Thesis

by

Khalid El Haji, Hakan Ilbaş, Sergen Karpuz, Noah Posner, Victor Wernet

to fulfil the requirements for obtaining the degree of

Bachelor of Science
in Computer Science and Engineering

at the Delft University of Technology,
to be defended publicly on Wednesday 1 July, 2020 at 14:00 PM.

Project duration: April 20, 2020 – July 1, 2020
Thesis committee: Prof. dr. L. Chen, TU Delft, Supervisor
ir. O. Visser, TU Delft, Coordinator

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

As the population increases so does the waste that is generated. Manually recycling waste is expensive and slow. Computer Vision (CV) solutions aim to make this less expensive and faster [5]. Lots of data of this waste (thousands of images) is needed to train these CV solutions. This project, called Synthetic Waste Generator (SWaG) can create synthetic waste data through the use of Blender and Python. Moreover, this project makes a contribution to the current state of research by having developed an automated synthetic data generation pipeline. This synthetic data can be used to train CV solutions to enable automated recycling procedures. With the help of adjustable parameters, the synthetic data can be customized, such that different unique images of waste can be created deterministically based on a seed. Furthermore, SWaG is fully portable as it has been containerized using Docker which makes it extremely easy to obtain even faster results by running SWaG on an NVIDIA GPU enabled system as a single local container, on the cloud as a farm or incorporate it in a container-orchestration system such as Kubernetes. SWaG also crushes 3D models, to mimic real waste using soft body dynamics. The pipeline has also been suited to automatically generate COCO datasets by using masking and image segmentation techniques. SWaG can also add textures and different colors to the waste objects in the synthetically created image. Furthermore, with SWaG different conveyor belt setups at recycling plants can be simulated with the help of variable camera heights, conveyor belts, backgrounds and lighting conditions. SWaG is currently deployable and is being used and built upon by our client. After conducting empirical research experiments with SWaG, it is noted that its performance speed is linear as the amount of objects that are in a given scene increases. In fact, with between roughly 40 and 80 objects SWaG performs sub-linearly. This is an important performance criteria as images of trash on the conveyor belt often have tonnes of objects pilled up on top of one another.

Keywords: Computer Vision, Recycling, Synthetic Data Generation, Generative Adversarial Networks, GPU, MS COCO, Mask R-CNN, Blender, Kubernetes

Acknowledgement
The team would like to express our gratitude to our supervisor Dr. L. Chen for her help with our research. Dr. Chen's suggestions and constructive feedback during the planning and implementation of this research have been immensely valuable. We would also like to thank our client for his generous support and involvement in guiding this software project.

Khalid El Haji, Hakan Ilbaş, Sergen Karpuz, Noah Posner, Victor Wernet
Delft, July 2020

Contents

1

Introduction and Problem Description

Recycling is good for the environment as a lower amount of trash ends up in it. By identifying and sorting different waste materials into different categories it is possible to generate economic value by transforming the waste material into a new product. The problem with trash is that it often contains a lot of different materials, from plastic to aluminium to wood. Recycling can be done more efficiently and cheaper if there is some way to cheaply classify trash. Classifying trash will also lead to less waste on landfills. Within current waste facilities a large amount of time and money is spent on manually sorting waste, often in poor conditions. Our client is a startup focused on transforming the waste industry by using computer vision to enable waste tracking, waste classification and automated sorting of waste. However lots of data (thousands of images) are needed to train a classification algorithm since trash can be crushed in an unlimited number of ways and the conveyor belt setups that are used to move the trash around can be different from facility to facility (e.g. different lighting conditions, or conveyor belt colors).

1.1. Overview

In chapter 2 synthetic data generation will be discussed. In chapter 3 the requirements of this project will be outlined using the MoSCoW format. In chapter 4 the evaluation criteria to evaluate our eventual software project are explained. In chapter 5 the different rendering engines and programming languages that could be used to implement the software are discussed, and the final verdict for the chosen rendering engine and programming language are given. In chapter 6 the software solution that SWaG will use is described. This chapter also shows possible parameters and example JSON requests that can be used for SWaG. Furthermore some results for this example JSON requests are shown. In chapter 7 the implementation of different image scene features are explained. chapter 8 explains how the 3D models are crushed and what the idea behind it is. chapter 9 discusses the dataset generation process, specifically the masking and image segmentation functions. chapter 10 explains the dataset management approaches and how it is stored. chapter 11 elaborates on the optimizations done to improve the rendering speed. In chapter 12 the refactoring, code quality and project management are described. chapter 13 discusses performance measures and displays some results. Lastly, in chapter 14 the report is concluded and possible future research in this field is discussed.

2

Problem Analysis Of Synthetic Data Generation

This chapter first discusses synthetic data generation approaches that were found in the literature. Outlining that pure synthetic data generation will be the approach followed over the course of this research. Next the different types of synthetic data that can be generated to train a GAN for this particular context will be discussed. Finally this chapter will be rounded up by investigating comparable solutions.

2.1. Synthetic Data Generation Approaches

In this section we will give several approaches for synthetic data generation. Most existing approaches can be divided into three categories [12]. These are, "(i) Augmentation of real data, (ii) creation of synthetic data based on real data features and (iii) creation of synthetic training datasets."[12]. The augmentation of real data is to apply changes to certain aspects of the real data, which are specific to each problem. This way synthetic data is created. The creation of synthetic data based on real data features most often is done with a Generative Adversarial Network (GAN). A GAN has a generative and a discriminative part. The generative part creates output whereas the discriminative part evaluates this output. The goal of the generator is to create synthetic data indistinguishable from real data. The discriminative part has to recognize the output as being either synthetic or real and provides feedback to the generative part to improve on this. GANs do require a large set of real data to be effective. This means that GANs may not always be the preferred approach when the real data is limited. Creation of synthetic data differs from the first two categories in the sense that it requires no real data. Besides the output being synthetic, the method used to create that output is also synthetic. The end product we aim for is a system that generates synthetic data. The data in this case are images of waste. Objects in waste come in many forms, most often crushed in unique ways. To simulate this, it is less effective to have real images of waste that only depict a subset of these forms. Rather, we want to have to ability to "make" our own waste. Being able to deform objects gives the opportunity to create a data set that is broader than a set of real images. This vision leads us to the third category, creation of synthetic data. Without any use of real data, we will generate our own synthetic data. We have chosen the following approach. A graphics engine will be used that places 3D models (that approximate real data) in a scene. The images that are rendered have multiple adjustable parameters. Using the graphics engine and the parameters we will render images that represent waste. Using the engine we can simulate crushing and thus create all sorts of permutations of the objects. Combined, this gives us a resulting data set of synthetic data.

2.2. General Object and Brand Level Recognition

Trash can be classified by its type (for example cardboard or plastic) or by its product brand. The synthetic data generation pipelines ability to create both types of images will enable the client to train and deploy a accurate model.

2.2.1. General Object Recognition

By collecting and applying multiple procedural textures such as the checkerboard patterns shown in Figure 2.1, it is possible to create a data-set for a high level categorisation problem where the task is to identify which material an object is for instance.

2.2.2. Brand-level Recognition

After applying branded textures to the objects as illustrated in Figure 2.2, the classification problem for the model is now of a more similar nature to what is found in the real world when recycling. The economic benefits of a dataset containing branded objects are greater than that of general object recognition dataset for the client since data about brands enables the client to derive useful analytics from what is being recycled.

2.3. Comparable Solutions

This software product has similarities with previous research projects. Tremblay, To and Sundaralingam trained deep neural networks for *Robotic Grasping of Household Objects* [45] using synthetic data. In order to draw a full comparison with their research results, sixty-thousand generated images and sixty-thousand photo-realistic images will be required. Due to time and budgetary constraints it may not be feasible to collect and store all these images required for a comparison with other research.

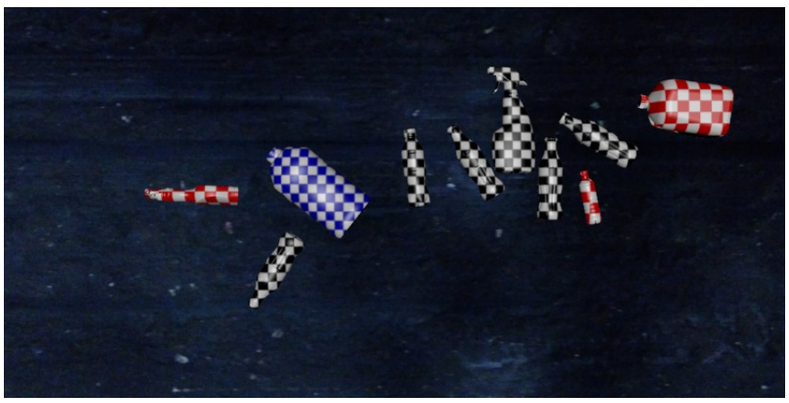


Figure 2.1: Example of a synthetically generated general recognition challenge

Figure 2.2: Example of a synthetically generated brand recognition challenge

3

Requirements

After meeting and discussing the software product with the client, technical requirements have been specified using the MoSCoW model. Some features have been labelled as Must-Haves in the MoSCoW model. Whilst other features will generate extra value to the project and were categorised as a Should-Have in the MoSCoW model. Through the course of the project the developers have been following an Agile philosophy. Therefore the requirements have been expanded throughout the project to reflect the updates that were discussed with the client at the weekly meetings. This will be explained in section 6.5.

3.1. Functional Requirements

The functional requirements state the functionality and services that will be provided by the software solution in a prioritized manner.

3.1.1. Must-haves

- The software must generate a data set from a range of materials and make the dataset usable in a real-world scenario:
  - The software must support multiple 3D models for PET, PP, HDPE, aluminium, and cardboard objects.
  - The software must support accessing labelled 3D models to be opened in rendering software.
  - For each type of material there must be at least one 3D model.
- The software must have realistic backgrounds for products to be placed on:
  - The software must ensure that in any image background a visual inspection of object size would not make the object seem disproportionate (e.g. a PET bottle the size of a car).
  - The software must place the background and objects at a variable distance of between 1-3 metres from the camera.
  - The camera must be placed in an aerial view of the materials (e.g. on top of a conveyor).
- The software must allow for changing the object material with an input parameter:
  - The software must provide a configuration file to add more materials (or in another way without coding).
  - The software must return a specific material from a selection of a number of 3D models by specifying the file name of that model.

3.1.2. Should-haves

- The software should have support for varying levels of crowding:
  - The software should be able to combine a crushed object with multiple other objects together in an image.
  - The software should have a easily changeable parameter for the number of objects in an image.
  - The software should have support for multiple material types per image.
- The software should render an image where different objects have different material types.
- The software should have the ability to alter lighting conditions for generated objects in an image:
  - The software should support the creation of multiple lighting variations (at least ten) with different conditions.
  - The software should allow for specifying lighting location, brightness, intensity, reflectivity.
- The software should change background quickly given input parameters:
  - The software should support at least 10 different representative backgrounds.
  - The software should be able to render representative backgrounds likely to be found in industry. For example, a table top, black conveyor belt, and green conveyor belt from a variety of view points.

- The software should have a parameter(s) to alter composition percentage of a material type (e.g. 60% aluminium) within a generated image.
- The software should have support for mask and bounding box output:
  - The output should be in standardized dataset format.
  - The output should produce a dataset output that can be used for training with a standard Mask-RCNN model.
- The software should deform 3D models to simulate crushing:
  - The software should be able generate any number of permutations of crushed 3D models.
  - The software should allow the use of a seed to reproduce a crushed permutation.
- The software should have the ability to change the skins of 3D models (texturing):
  - The software should have the ability to change 3D model colours.
  - The software should support least 10 variations of skin colours.
  - The software should have the ability to change or add graphics/shapes on the skins.

3.1.3. Could-haves

- The software could allow for rotations and translations variations of objects

3.1.4. Won't-haves

- The software won't have a Generative Adversarial Network (GAN) for image generation.
- The software won't have object detection.

3.2. Non-functional Requirements

Unlike the previously specified functional requirements, the non-functional requirements are constraints for the development process or the system.

- The software must be compatible with Mac OSX (10.8 or greater), Windows (7 or greater), and Linux.
- The software must be developed in Python 3.6.
- The software must render realistic 3D models of products. The rendering quality must be similar or better than Grand Theft Auto V product rendering.
- The software must be able to generate one synthetic image in at most 5 seconds when using a NVIDIA GeForce GTX 1080Ti and a recent (and decent) server grade CPU.
- The software must only contain functions with a relatively small McCabe cyclomatic complexity (within the range of 0-15).
- The software must have no functions larger than 15 lines of code.
- The software must have an API interface for generating synthetic image via a JSON request.
- The software must be developed using unit testing and static analysis tools.
- The team responsible for the development of the software must use the SCRUM methodology.
- The implementation of the software must have at least 70% meaningful line test coverage.
- The implementation of the software must be fully containerizable and deployable via Docker.
- The software must have an operation/user guide.

4

Evaluation Criteria

The software shall be judged by its features, performance, image quality and how well it integrates with the clients current software infrastructure. As shown by the real example images provided by the client to the development team, the first step to training a machine learning model for waste object identification is to create images with individual objects. The next step is to combine multiple different objects in an image. Finally the last step is to create more realistic images with objects that are crushed. For more details about the evaluation criteria for the software features, which have been classified as either critical or bonus please see the MoSCoW requirements found in chapter 3.

4.1. Performance

The software's performance (speed) and image quality are the two key aspects to consider when evaluating it. If the software is too slow it will not produce enough images to be useful to train a machine learning model. The speed to generate images should be at most 5s per image on a standard NVIDIA GeForce 1080Ti. Similarly, the graphics must be realistic enough so that when the model is deployed in reality it is not confused. To enforce this the images should have at least the rendering quality of a modern video game such as Grand Theft Auto V.

Figure 4.1: Example of a soda can from GTA V gameplay. Image courtesy of Ecola [15].

4.2. Evaluation Method

Object detection is used in a variety of fields. This process relates to image classification, where the key point is to identify or label a single or multiple targets in an image with a high probability of the target being that given label. The object of course, will be surrounded by a bounding box In order to measure the quality of the data generated by our method, we would first have to use a Mask R-CNN [21]. By comparing performance when training the Mask R-CNN on the images we generate vs standard images of real trash we can determine whether our approach is promising. Such an evaluation requires a large quantity of synthetic and real images and therefore is not possible to do during the time-span of this research project. Therefore, after discussing this problem with our coach an Image Quality Rubric was developed in consultation with our client in order to evaluate the quality of the data being produced.

4.3. Image Quality Rubric

To be able to objectively determine the quality of the images being produced, the developers in collaboration with the client and coach created an Image quality rubric to grade the images. This enabled the developers to have a

key focus of what type of images would be suitable for the client and where improvements needed to be made.

	Image Quality Rubric				Score
Evaluation characteristic	Excellent (8-10)	Good (7-8)	Average (6)	Low Quality (1-5)	
Proportion	The objects are well proportioned to one another	The objects are proportionate to one another	The objects are somewhat proportionate to one another	The objects are not proportionate	
Background	The objects are placed on a realistic looking background, and there are 10+ backgrounds	The objects are placed on a decent looking background and there are enough (5) backgrounds	The objects are placed on a background, but there are not a lot of them, and they are not very realistic	The background does not fit the scene and there are not a lot of backgrounds	
Realism	The image has GTA V level rendering	The image is somewhat realistic	The image is sufficiently realistic	The image looks like a cartoon	
Textures	The objects have their associated textu	The objects have their associated textures fitted	The objects have sufficient textures	The textures are not applied to the objects correctly	
Resolution	The image quality is high: 1200 x 800	The image quality is decent: 800 x 600	The image quality is sufficient	The image quality is ins	
Results matching expectations	Image has matching charecteristics to what was specified in the user request	The image has some characteristics missing	Only some of the characteristics are matching	The resulting image does not match the parameters set	
Lighting	Lighting is similar to what is used at waste facility. For example bright lights are used with minimal shadows.	The lighting results in some shadows that are not realistic	The lighting results in many shadows that are not realistic	The lighting is too dark	
Camera positioning and aspect ratio	The camera is positioned in a location that is similar to what is used at the waste facility. The camera is directly above the conveyer belt at a reasonable height	The camera postioning is suitable. It could be further away/closer to the conveyer belt to look a bit more realistic	The camera position should be improved by moving it to a new location	The camera is way too close/far away from the objects	
Object crushing	Objects look crushed in a lifelike manner	Objects look crushed. The crushing is not very realistic i.e(Completely crumpled can that has had too much force applied to it)	Objects are crushed way too much or not at all and the result does not look realistic at all	The objects are not crushed	
Object overlap	Objects overlap in a similar way to reality	Objects overlap is mostly realistic. No objects are breaking the laws of physics i.e going through one another	Objects overlap mostly realistically. Some objects do not overlap well	The objects do not overlap in a realistic way at all	
Masking	The image masking is suitable for insta	Image masking is suitable for instance segmentation	Image masking is not really suitable for instance segmentation	The image is not suitable for instance segmentation	
				Final Score	
				Grade	

Figure 4.2: Image quality rubric

5

Survey of Technologies

Within this chapter an extensive analysis of different graphics engines and programming languages has been completed in order to pick the best tools for the project. A verdict with justification on which graphics engine and programming language are the best to use is given at the end of their respective sections.

5.1. Graphics Engines

To create the synthetic data set, we need a rendering engine such that it can create images out of 3D objects (which are models of for example aluminium cans, plastic bottles etc.). The images that are then rendered may have different backgrounds and lighting environments. There are many rendering engines that can do this task. However there are certain requirements that these rendering engines need to satisfy set by us. Firstly, the most important functionality that the rendering engine must have is an Application Programming Interface (API). This is a hard requirement since we want to automatise the synthetic data generation. By using an existing API, it is possible to fully automate not only the rendering but also the crushing algorithm. Secondly, we want the rendering engine to have support for the most common non-proprietary 3D model file types, mainly OBJ, STL, COLLADA, IGES, STEP and VRML/X3D. This is because our software can than be used with newer 3D models as well. Another thing that we have to keep in mind is the cost of a license for the rendering engines. The client has a specified budget per year to spend on licences. On the basis of these requirements we have found 3D rendering engines that satisfy them. In the following section we will shortly explain each of them and their properties. In the end there will be a comparison, where we select the best rendering engine for our uses. We will use this rendering engine then for our synthetic data generation software.

5.1.1. Blender

Blender [9] is a free and open source well-known rendering engine. It has support for the OBJ, STL, X3D 3D model file types. Blender also has a well documented Python API, which in turn can be used to automatise the synthetic data generation process. The API is also called BPY. Blender also has a huge community which can help to get us started more quickly. An attractive point of Blender is that, compared to other rendering engines, it is smaller in physical size upon close inspection. For this rendering engine comparison we will use Blender 2.79, instead of the newest version 2.8. The reason for this is that there are some huge differences in the API between both versions. Since version 2.8 is relatively new, there is less help available online, if we get stuck in the middle of implementing something. A lot of older tutorials cannot be used, since Blender changed a part of the implementation in version 2.8. Blender 2.79 has three rendering engines:

5.1.2. Blender Render Engine

This is the default rendering engine of Blender. As Blender calls it, "it is a non photo-real rendering engine" [1]. Even though Blender defines Blender Render as a non photo-real rendering engine, the renders still look realistic enough for something like a Region with Convolutional Neural Network (R-CNN), which is what our client uses to train his waste detecting AI. Also the quality of the renders generated by Blender Render is as good if not even better than GTA V graphics, which was an acceptance criteria that was set by the client. It is significantly faster in rendering scenes than its counterpart Cycles, at the cost of quality and realism, since it does not look as photo realistic. Although the quality is also not the worst and it still looks realistic enough for a R-CNN, its counterpart Cycles renders the scenes in a much more realistic way, at the cost of it being much slower. Because of this it is more likely that Blender Render will be used the most, as the increase in quality is not often needed and faster rendering times are preferred. Unfortunately it is not possible with Blender Render to accelerate rendering times with the use of a GPU. Blender Render can only render the scenes with a CPU.

5.1.3. Blender Cycles Rendering Engine

Cycles is an alternative rendering engine in Blender. It is a "ray-tracing production render engine" [2]. Its renders are much more realistic than Blender Render, because it uses ray-tracing to render the images. But for the same reason, it also takes a longer time to render the scenes. Cycles also has the advantage that it can use the GPU of the rendering machine to lower the rendering times. Unfortunately this only works for selected set of NVIDIA and AMD graphics cards. As said earlier, this is not possible with Blender Render. With the use of a graphics card, the rendering times are still not as fast as that of Blender Render, but potentially with a better graphics card, the rendering times will be as fast, if not even faster than Blender Render. It might be faster than Blender Render with a better graphics card, but unfortunately we did not have access to this. A comparison between the quality of the renders of both rendering engines can be seen below.

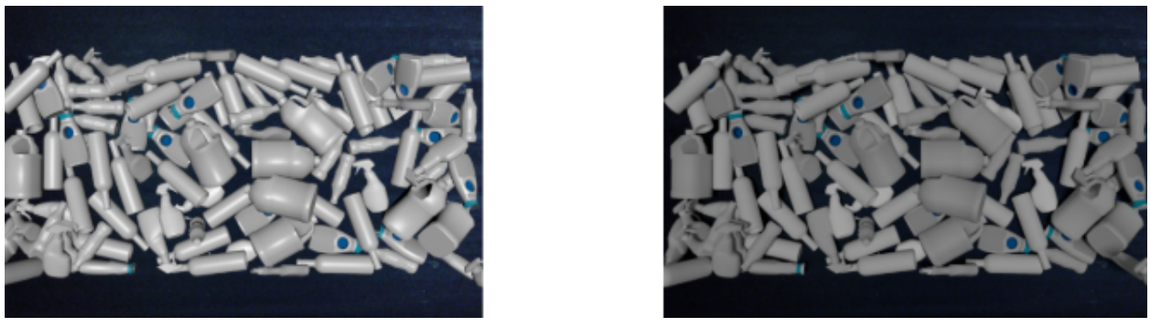


Figure 5.1: Render quality comparison between Blender Render and Cycles

The left image is a render created by Blender Render, and the right image is a render created by Cycles. Both scenes are exactly the same and contain 100 objects. Blender Render rendered the scene in **13.5 seconds** (with an Intel i7 7700HQ) with a tile size of 16x16 (tile sizes will be explained in a later section), and Cycles rendered the same scene in **88 seconds** with a CPU (Intel i7 7700HQ), with a tile size of 16x16 and in **57.71 seconds** with a GPU (laptop grade NVIDIA GeForce GTX 1050) with a tile size of 4096x4096. These numbers can be hugely improved by using better non-laptop CPU's and GPU's to render the scenes. Since Blender is free it means that we can containerize instances without the need for multiple licenses which decreases the cost of running the product dramatically.

5.1.4. Blender Game Rendering Engine

By using Blender version 2.79 the software has the capability to use Blender Game as a rendering engine. But since this rendering engine is a game engine, we did not consider using this engine.

5.1.5. Houdini

Houdini [41] is a rendering engine which was designed for artists working on 3D animations and Visual Effects (VFX) for film, TV, video games and Virtual Reality (VR). With Houdini it is also easy to explore previous iterations of models whereas this is for example more difficult with Blender. It comes with a free (Houdini Apprentice) version and a paid version. The free version comes with a bunch of restrictions and the paid version costs between a few hundred dollars to a few thousand dollars per year. The free version supports only a single file type, but it does have a Python API available. Houdini has support for a large number of file formats including STL and OBJ.

5.1.6. Unreal Engine

Unreal Engine [16] is a game engine which is very popular amongst game developers. Apart from game building, is also used for architecture design, simulations, and in automotive, to mention a few industries. Unreal Engine has support for OBJ files but not for the commonly used STL file format. Unreal Engine does however have a Python and C++ API, but both documentations are in experimental phase, meaning that it could be incomplete or contain inaccurate information. Unlike Blender, Unreal Engine is initially free, if the product its revenue is less than $3000 dollars per quarter, otherwise 5% of the product its gross revenue will be charged [30]. With Unreal Engine there are also different packages available which can be bought as a subscription to extend functionality if needed.

5.1.7. Unity

Unity [46] is a cross-platform game engine, that apart from building games, is also used in creating 3D models and simulations, as well as for 3D (real-time) rendering. Similar to Unreal Engine, Unity is also used in various industries. Unity has a Python API, that can be used to integrate the Unity Engine into a pipeline. There also is a very active community base, meaning that there is more material that we can use to learn about it. Lastly, Unity has a free version for students, and normal users are only eligible for personal use if funding for a given project has not exceeded a given amount for the last 12 months. However for a profit-driven business model a license is required.

5.1.8. Comparison and Selection of Rendering Engine

In the following matrix we compare each rendering engine using the following criterion: cost, performance, resolution, ease of use, documentation, community support and team familiarity. The criterion were determined by what needs need to be satisfied in order to realize the project. Additionally, we have assigned a weight to each criterion to represents its importance. The weights have range from 1 to 3, where the greater weight, the more important the criterion is. Furthermore, a score is given for each criterion given a rendering engine. The score have a range from -3 to 3, similarly to the weights, a greater score is better. The cost score is given on basis if its free or not. If the software is free we give 3 points. If its not and there are subscriptions, royalties, licences etc. we give a score of -3. For scores on the performance rubric we used a YouTube [1] video in which the performance of Blender and Houdini are compared. In this video a 3D model of 2 million polygons is edited. The video showed that when the same model is modified, Houdini is 3.5 times faster in showing the modification. Another video also shows that when selecting polygons of the same 2 million polygon 3D model, Houdini is just way faster when selecting a part of the 3D model, whereas there is some 2 second lag with Blender. Also when browsing through the menus in Blender with the same model, with Blender there is always some noticeable lag, where there is none with Houdini. Considering that we are going to use the rendering engine software a lot throughout the project, the performance of the rendering engine is important, hence the weight of 3. When we compare Unreal Engine 4 with Unity with the same 3D scenes we see that Unity always has about 10% more frames per second than Unreal Engine 4. Furthermore we also conducted a small test between Blender and Unity. In this test we rendered the same 3D object

[1] https://www.youtube.com/watch?v=iHxHPWdQr7k

and measured the rendering times. In this rendering there was a model of an aluminium can without a texture. Unity rendered the image within 0.005 seconds, and Blender rendered it within 0.06 seconds. So there is a factor 10 difference in rendering speed between both rendering engines.

Rendering Engine Software Comparison

		Blender		Houdini		Unreal Engine 4		Unity	
Criterion	**Weight**	**Score**	**Notes**	**Score**	**Notes**	**Score**	**Notes**	**Score**	**Notes**
Cost	3	3	Blender is free and open source	-3	Houdini prices range from a few hundred to a few thousand dollars per year	-3	Royalties and different subscribtion models	-3	License needed
Performance	3	-3	Very slow when modifying large 3D models	-1	Slow when modifying large 3D models	-3	Very slow when modifying large 3D models	-2	
Resolution	3	1		3	Used in VFX, games etc. Good quality	3	Used in games. Good quality	3	Used in games. Good quality
Ease of use	2	-1	Hard to use for beginners	-3	Hard to use for beginners	-3	Very hard to use for beginners	1	
Documentation	2	3	Good documentation for the API	2	Good documentation for the API	-2	Poor documentation for API	3	Good documentation for the API
Community support	1	3	Large community	1	Not so active community	2	Relatively good community	3	
Team familiarity	1	2		-3		-3		-3	
	Final Score	23		-7		-20		2	

Figure 5.2: Rendering Engines Comparison Matrix

The comparison matrix makes it very clear that Blender is the best rendering engine to meet the needs of our project. Specifically, we use Blender 2.79. The second best rendering engine, Unity, does not even come remotely close. As was explained earlier, Blender 2.79 was considered for this comparison, and not the newest version of Blender, being 2.8 at the time of writing this report. The reason for this is that it is relatively new. Some features as well as the API have changed, meaning that old documentation and tutorials are not valid anymore. New documentation and tutorials need to be created, or older ones need to be updated.

5.2. Potential Programming Language

In order to create the best possible outcome in terms of our software output, we have considered four different languages to realize our solution. The particular five languages are: C++, Java, Python, and JavaScript. These languages have been selected based on their familiarity in the team and their ease of integration with the previously discussed (see section 5.1) rendering engines. We will provide an final verdict on the chosen programming language and why it was chosen in section 5.2.5.

5.2.1. C++

The C++ language is a low-level statically-typed general purpose programming language and allows for direct memory management. C++ is frequently used for graphics intense applications (such as video games) because of its fast performance and access to low-level hardware constructs. Additionally, there are many existing libraries to support the development of graphics applications in C++. For example, some well-known ones are: OpenGL [26] and the aforementioned Unreal Engine. Furthermore, there are static analysis tools such as Cppcheck [13] that can be used for quickly analyzing code quality and there are well-established open-source testing libraries such as Boost.Test [38].

5.2.2. Java

The Java programming language is an high-level statically-typed general purse programming language with great support object-oriented programming. The Java language has many well-developed and long-existing build tools that makes continues integration trivial. Some well-known tools are Checkstyle [32] for static analysis, Gradle [20] and Maven [8] for build automation, and jUnit [25] and Mockito [43] for testing. As a result of the strong built environment and development tools surrounding Java, it becomes an attractive choice to use in a software project. However, the Java language does not have many existing well-supported graphics libraries. The most well-known graphics library for Java is JOGL [24] which is library wrapper for OpenGL. Another graphics library is Java3D which has poor cross-platform support.

5.2.3. Python

Python is an easy to write and use high-level dynamically-typed general purpose programming language. Python provides many built-in libraries and is well-known as a "batteries included" programming language. The Python language is not very fast in comparison with it's lower-level predecessors such as C and C++. However, whenever performance is key, Python allows for C extensions. With C extensions one can write routines in C or C++ and then use them within Python. Many well-known data analysis libraries that exists such as NumPy [23] use C under the hood in order to improve performance. Using C extensions can be quite cumbersome, and considering the high performance requirements of this project we can also consider to use of Cython [39] instead of native Python. Cython is "optimizing static compiler" to Python which allows easy bridging between Python and C/C++ code [3]. Furthermore, Python has reasonable tools for static analysis (e.g. PyHint [34]) and testing (e.g. PyTest [10]). Another noteable consideration is that the project client primarily uses Python in their software development.

5.2.4. JavaScript

The JavaScript (also known as ECMAScript) programming language is an high-level dynamically-typed general purpose programming language, however it's most frequently used in web development. The language has a great tooling support for static analysis (e.g. JSHint [7]) and testing (e.g. Karma [47] and Jest [17]). Some graphics engines such as Unity support the JavaScript language and hence could be useful. However, JavaScript has overall poor support for building graphics intensive applications (ref). Most web applications are written using JavaScript. Even not considering web applications, it has become more frequent to write desktop applications using JavaScript thanks to tools such as Electron [33].

5.2.5. Comparison and Selection of Programming Language

In the following matrix we compare each programming language using the following criterion: performance, ease of use, graphics libraries, testing and tooling abilities, integration support with rendering engines, community support, and team familiarity. The criterion where determined by what needs need to be satisfied in order to realize the project. Additionally, we have assigned a weight to each criterion to represents its importance. The weights have range from 1 to 3, where the greater weight the more important the criterion is. Furthermore, a score is given for each criterion given a programming language. The score have a range from -3 to 3, similarly to the weights, a greater score is better.

Programming Languages Comparison

		C++		Python		Java		JavaScript	
Criterion	**Weight**	**Score**	**Notes**	**Score**	**Notes**	**Score**	**Notes**	**Score**	**Notes**
Performance	3	3		1	Performance can be lackluster, but C extensions can help	2		1	
Ease of use	2	0	In comparison with other languages, C++ requires more attention to details	3		0		2	
Graphics libraries	3	3		2		-1	Limited in general, not well-supported	0	Often limited to the web
Testing and tooling abilities	1	1		3		3	Java has excellent tooling in comparison with the rest	0	
Integration support with rendering engines	3	3	Most major rendering engines have support for C++	2		-1		2	
Community support	1	0		3	Python is well-known for its large community	1		3	Excellent community support in general, but limited when it comes to graphics applications
Team familiarity	1	0	Team has lmited familiary with working in C++	1	Team has somewhat familiariy with working in Python	3	Team has a lot of familiariy with working in Java	0	Team has limited familiary with working in JavaScript
	Final score	28		28		7		16	

Figure 5.3: Programming Languages Comparison Matrix

The comparison matrix makes it clear that C++ and Python are the best languages to meet the needs of our project. Considering that Python has greater support for working with our chosen rendering engine (Blender, see section 5.1.8) and that Python ease of use is much better than C++, we have decided that Python best meets our needs and hence will be used to implement our project. Furthermore, Python 3.6 will be used specifically. As has been discussed earlier, the testing framework PyTest will be used for testing and the static analysis tool PyHint will be used to ensure a good level of code quality.

6

Software Solution

This chapter discusses the software solution that was developed. In section 6.1 a high level overview is given of the software architecture. The following section 6.2 gives an overview of the pipeline and how the software architecture functions in practice. Section 6.3 elaborates more on how containerization is incorporated into the pipeline. Next, section 6.4 describes the input parameters that the JSON request can contain as input. Lastly section 6.6 gives an example output that the pipeline generates depending on which input parameters were sent to SWaG. Michel Anders describes the difficulty of learning how to automate Blender: "Mastering a scripting language and getting familiar with the many possibilities that Blender offers through its Python API can be a daunting venture." [6] Blender has a GUI (Graphical User Interface) that enables designers to adjust scenes to their liking. However, SWaG will not have or make use of a GUI. In order to achieve the objective of this project the developers used Python to write Blender scripts with the Blender-Python (BPY) library. The developers implemented features such as colors, lighting, camera, and texture settings to be customizable by the user. However, designing and applying textures to objects is a manual process. Therefore, removing the designer completely from the loop is not possible for this project.

6.1. High-level Software Architecture

The high-level software architecture consists of only a back-end which can be used by sending a JSON request to it. The back-end consists of subsystems. The following figure demonstrates the back-end and its subsystems.

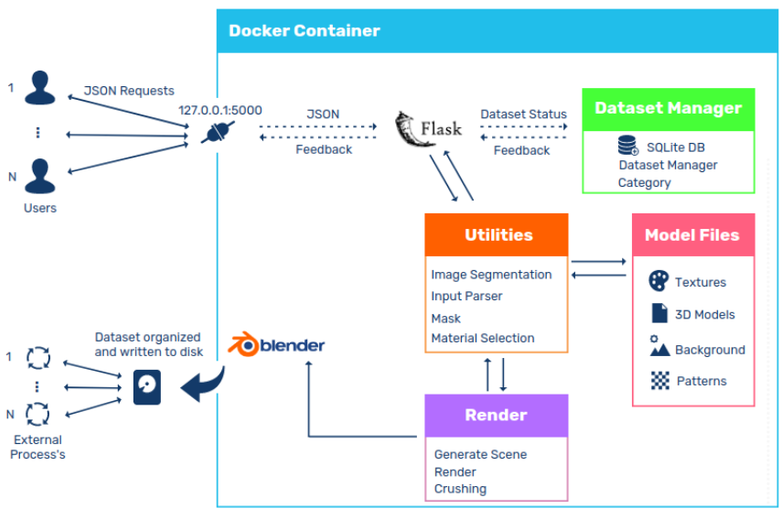


Figure 6.1: Software Architecture Diagram

Blender is responsible for rendering images and providing interfaces to modify 3D models as well as for scene creation. The utilities module is responsible for generating a list of objects that need to be rendered by accessing models from the Model Files storage and appending them to a list of objects that need to be rendered. In the render module these models are than modified/deformed/transformed into objects that can be placed into the scene. The render module takes previously created objects and place them in the desired scene (e.g. one with a certain background or one with varying crowding). The model files storage is responsible for storing models and visuals/texture respectively. The dataset management deals with dataset creation or extension. It handles multiple incoming requests and provides a concise, simple and powerful way of protecting the datasets from data loss through a lock/unlock mechanism.

6.2. High-level Overview of The Pipeline

After the user made a JSON request, the pipeline starts with the synthetic data generation process. The dataset is then generated and saved on the volume the user bound to the container. This allows other external processes that need the dataset to directly have access to it. The simplicity of this allows the pipeline to be easily incorporated like a puzzle piece into a larger already existing system infrastructure. In Figure 6.2 each consecutive step in the pipeline are discussed in a simple to grasp format without getting lost in too much detail.

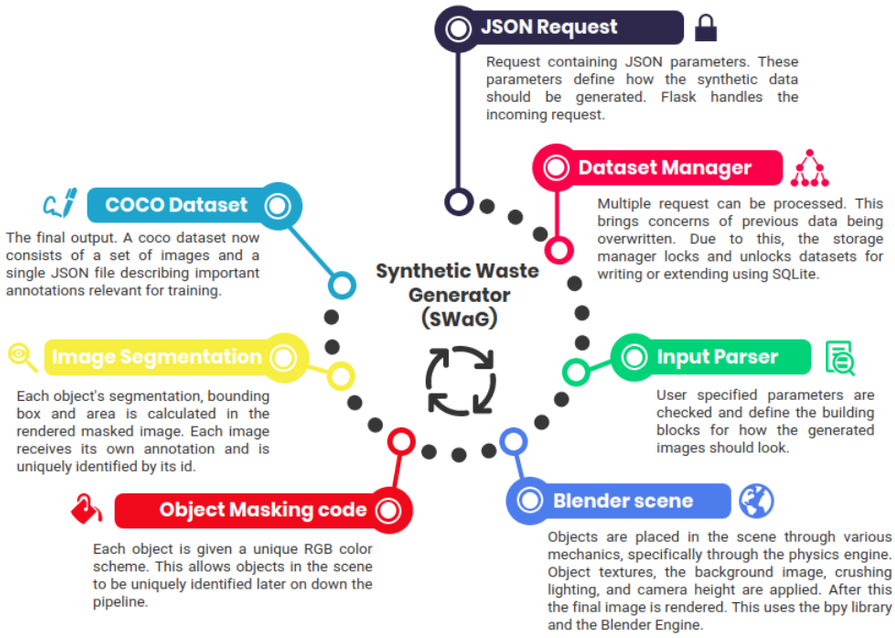


Figure 6.2: SWaG: the pipeline steps demystified.

6.3. Containerization

6.3.1. A Brief Introduction To Docker

Running any piece of code requires the correct computational environment and software. Enabling virtualisation and cross-platform portability makes Docker an incredibly powerful technology when trying to ensure consistent reproducibility. Another benefit of using Docker is that it is an open source project. Of course arguably a Virtual Machine or commonly known as a VM for short, could also have been a viable solution for the pipeline. We chose to not follow through with this approach due to the extra overhead and resources required to to facilitate such a system. Simplicity is key here, speed, and an approach that can be easily incorporated in an already existing system with a simple few instructions were the key components [11]. Below we compare the two system through an image without adding too much details to an already complicated matter. The image simply encompasses the various layers needed by both systems.

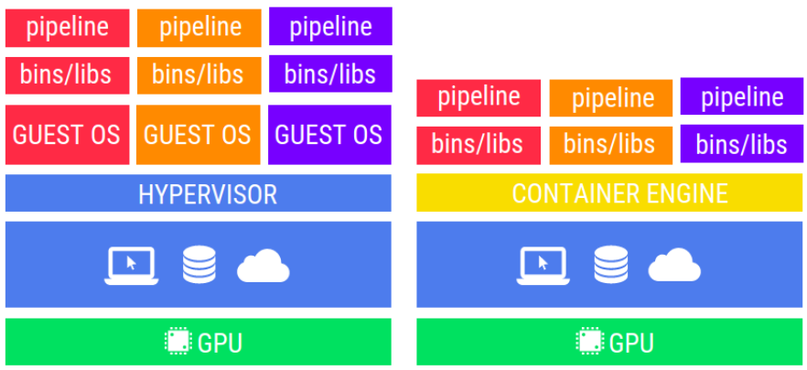


Figure 6.3: Virtual Machine versus Containerization.

6.3.2. Environment Stack

Let's briefly discuss the below depicted image in a bit more detail in a bottom up manner. A user may be running Windows, Mac OS or a Linux based OS with an NVIDIA GPU card, of course it is not necessary to utilize the pipeline with a GPU but does add the extra benefit of reducing the computation time and render time overall. The Host OS is depicted as the bottom gray layer. The layer on top contains the NVIDIA GPU driver, utilities and the CUDA toolkit. This facilitates with the GPU mapping and the correct utilization. The Host OS and the required NVIDIA software packages are the minimum requirements needed for the layers that follow. We now reach the layer where the Docker Engine resides. The Docker engine can be download from the official docker website, and is compatible with various OSs. This is now the only last required software package the user needs to install and configure. The user can now through the usage of the Dockerfile, run the required commands to build the image. The Dockerfile has been configured to pull all of the required software packages, drivers, utilities, tools and Docker images that are need by the pipeline. The user can now run the container as any other process. The visual example of the container can be seen under the containerized pipeline section in the image. This approach allows multiple containers to be run simultaneously. In other words this adds a great deal of computational power to create synthetic training data.

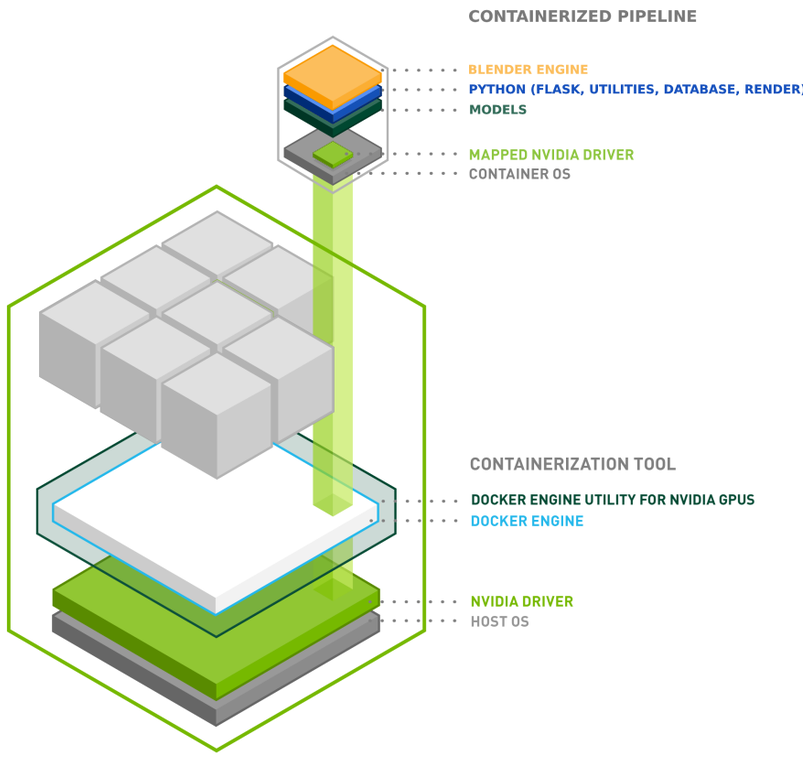


Figure 6.4: A visual representation of the current environment stack used by SWaG. Original image adjusted from the NVIDIA documentation [31].

6.4. Input Parameters

To create a synthetic image a JSON request must be sent to the server. This can be done via cURL (via the terminal) or via Postman (via a GUI). The program accepts various parameters to alter the scene. Here is a list of all parameters that can be given in the JSON request. If a parameter is given that is not in this list, it will be ignored. All parameters in the request that will not be ignored will be validated by the program. If the values of the parameters are invalid, the program will stop and return an error to the user. Furthermore there are some parameters that must be filled in, in every request.

Seed: This parameter is used to for randomly placing objects in the scene. It is used for crushing as well as for the part of the software that randomly picks a model of a certain material. A random seed will be used if a seed is not specified in the JSON request. If a seed is chosen it must be between 0 and 2147483647. Otherwise the program will stop. The random seed that will be chosen when no seed is specified is also between 0 and 2147483647. The seed value can be sent as a string ("seed": "123") or as an integer ("seed": 123). An example would be "seed": 53254352.

Crowding: This parameter specifies the amount of objects that will be placed in the scene and rendered. Sometimes it may occurred that there are more items in the scene than was specified by the crowding parameter [1]. Crowding parameter must be specified in the JSON request. If no crowding parameter is specified the program will stop and return an error saying that the parameter is not filled in, in the JSON request. The crowding value can be between 1 and 500. For values outside these bounds, the program will stop and return an error. More items in a scene will result into longer rendering times. The crowding value can be sent as a string ("crowding": "10") or as an integer ("crowding": 10). An example would be "crowding": 25.

Material types: This parameter specifies the material types of the objects that will be rendered in the scene. Depending on these material types, random models from these material_types will be chosen. To see which material_types are available please go to the models folder in the root directory of the project. More material types can be added easily by creating new folders in the models folder. These folders can also be removed if the user wants to remove certain material types. The value of material_types must be a list with strings, where the strings must be the names of the folders in the models folder. If a string is sent that is not in the models folder, the program will stop and return an error. This parameter must always be sent in every JSON request, except when a file_name is specified. Otherwise the program will stop and return an error. Depending on the crowding, material_types and material_composition parameter, a random list of models will be chosen to get rendered in the scene. For example, if the crowding parameter is set to 5, material_types is set to ['aluminium'] and material_composition is set to [1], 5 random models of material_type aluminium will be rendered in the scene. If the user wants to be more specific

[1] Sometimes it may occur that there are more items in the scene of the rendered image, than was specified in the crowding parameter. This only happens when no file_name parameter is specified (and thus material_types and material_composition parameters) are specified. This is not a bug. We will explain this with the following example. If a crowding parameter of 6 is taken, a material_types parameter that is ['aluminium', 'glass'] and a material_composition parameter that is [0.2, 0.8] means that 20% of 6 items should be aluminium, and 80% of the 6 items should be glass. 20% of 6 and 80% of 6 are 1.2 and 4.8 respectively. Since it is not possible nor realistic to have 1.2 items in a scene, the ceiling of these numbers are taken, which means that there will be 2 + 5 = 7 items in the scene. When there are no fractions the crowding parameter works as expected, but otherwise the ceiling of the fraction is taken, meaning that the objects in the rendered image exceed the crowding parameter.

about exact models to get rendered in the scene the file_name parameter must be used instead of material_types. An example would be "material_types": ['aluminium', 'glass'].

Material composition: This parameter specifies the composition of the material_types. This parameter must be a list of the same length as the material_types list. If the lengths of both lists are not the same the program will stop and return an error. The list can contain integers or strings that have integers inside them, or a combination of both. The sum of the numbers in the list must sum up to 1, otherwise the program will stop (with boundaries set at 0.99 and 1.01). This parameter must always be sent in every JSON request, except when a file_name is specified. Otherwise the program will stop and return an error. An example for "material_types: ['aluminium', 'glass'] would be ["0.5", "0.5"], or [0.7, 0.3], but not for the same material_types parameter, an incorrect material_composition parameter would be [1], [0.2, 0.2] or [0.3, 0.3, 0.4].

File name: This parameter specifies the files that will be rendered. If this parameter is specified, no material_types or material_composition parameter need to be specified (if they are specified, they will be ignored). Depending on the crowding parameter, the files specified in the file_name parameter can be placed multiple times in the scene. A single file, as well as multiple files can be specified with this parameter. Keep in mind that the actual model file names must be specified and not the folder names of the models. If a model is specified in the JSON request that is not in the models folder, the program will stop and return an error. Some examples: "file_name": "wine_bottle.fbx" or "file_name": ["wine_bottle.fbx", "coca_bottle.obj"].

Background: This parameter specifies the path of the background of the scene. If it is not specified a default background will be chosen for the background of the scene (which will be b1.JPG, being the first background in the backgrounds directory). This parameter should be the path to the background from the root folder. I.e. if in the root folder there is a "backgrounds" folder with different backgrounds, the background value would be "backgrounds/some_image.png". If an invalid background path is given, the program will stop and return an error.

Resolution: This parameter specifies the resolution of the rendered scene. If this parameter is not specified, the default picture size will be 1200x800 pixels, as specified by the client. The resolution value must be a list with two positive integers (or strings that contain integers, or a combination of both). If more than or less than 2 integers are specified in the request, the program will stop and return an error. If the values in the array are not positive the program will also stop and return an error. An example would be "resolution": [1200, 1000] or ["1000", "1000"].

Enable GPU: This parameter specifies whether or not the GPU shall be used for rendering the scene or not. This option only works if the rendering machine has an NVIDIA graphics card starting from the GeForce GTX 4xx (CUDA computing capability from 2.0 to 6.1) [4]. OpenCL is also supported for GPU rendering with AMD graphics cards, although only graphics cards with GCN architecture 2.0 and above are supported [4]. If this option is turned on and there is no (supported) graphics card, the software will render the image on CPU instead. This option only works for Cycles and will be ignored if Blender Render is used. The value can be a string or Boolean and can be with or without capital letters if a string is used. If it is not specified in the JSON request the GPU will not be used to render the scene, even if there is a supported GPU available on the rendering machine. An example would be "enable_gpu": False or "enable_gpu": "fAlSe".

Rendering engine: This parameter specifies which rendering engine to use. Currently the program supports two rendering engines: Cycles and Blender Render. The first rendering_engine is more realistic, but it takes longer to render the final image. The second rendering_engine is less realistic, but its graphics are comparable to Grand Theft Auto V (GTA V) or even better. Rendering times also take considerably less whenever this rendering engine is used. The default is set to blender_render, meaning that if no rendering engine is available in the JSON request, Blender Render is used. If the user wants to render with Cycles the value of this parameter should be "cycles", where capitalization does not matter. More rendering engines can be added later with plugins. An example would be "rendering_engine": "Blender_RENDER".

Dataset name: This parameter specifies the name of the new dataset or an existing dataset. This supports the ability for a user to create a new dataset on disk. If the dataset is not present on disk it well then be created. On the other hand if the dataset is already present, it will then append the newly created data to it. This allows the user to extend an already existing dataset in the case of missing training or validation data. The storage manager handles which folders can be written to or created. A simple but effective lock/unlock mechanism resolves any issues.

Version: This parameters provides the simple function of updating an already existing version specified by the dataset_name. A user that specifies this can expect the version to be updated to the configured value. This parameter is not obligatory. If no version parameter is configured, the default value of 1.0 is given to the newly created dataset and for an already existing dataset the version will remain the same.

Camera height: This parameter specifies the camera height. In this case it means how close the camera is to the scene. Currently three values are supported, namely the integer values '1', '2' and '3', where '1' is the closest to the scene, '3' is the furthest away from the scene and '2' is in between. The default value, if no value is specified by the user , is '1', meaning the closest to the scene. The make the system more robust some input sanitation is done. This means that this specific parameter actually can take a float value as input and sanitize it to the corresponding integer value that is required. For more information about the camera see section 7.6.

Surrounding box size: This parameter specifies the size of the surrounding box. Currently three values are supported, namely the integer values '1', '2' and '3', where '1' is the smallest box, '3' is the largest box and '2' is in between. The default value, if no value is specified by the user , is '1', meaning the smallest box. The make the system more robust some input sanitation is done. This means that this specific parameter actually can take a float value as input and sanitize it to the corresponding integer value that is required. For more information about the surrounding box see section 7.6.

Surrounding box hide: This parameter specifies whether or not to make the surrounding box visible. It supports Boolean (string) values meaning either "true", true, "false" or false. In case the value specified is "true"/true, the surrounding box is hidden from the rendered and will not show up in the rendered image. In case the value specified is "false"/false, the surrounding box is made visible and will show up in the rendered image. The default value, if this parameter is not specified by the user, is "true", meaning the surrounding box will be hidden in the rendered image. For more information about the surrounding box see section 7.6.

Light strength: This parameter specifies the intensity of the light source which illuminates the scene. Only positive integers (or positive integers inside strings) are accepted. A higher light_strength corresponds to a more brightly lit scene. Cycles and Blender Render handle light intensity differently, so different values are needed for both rendering engines to achieve the same brightness. If a negative integer is given an error is returned. The default light strength is set at 10. An example would be "light_strength": "12".

Light location: This parameter specifies the location of the light source which illuminates the scene. For Blender Render it does not matter if the light location is outside of the surrounding box or not. If the light_strength is sufficient and the light is not placed too far away the scene will still be lit. But since Cycles handles light differently (and mainly because it uses actual ray-tracing), the light location can only be in the surrounding box, otherwise the scene will remain dark and it will not be illuminated. This parameter is a list that must consist of 3 integers, representing the x, y and z coordinates of the light source. Index zero represents the x coordinate, index one represents the y coordinate and the last index represents the z coordinate of the light source. If the list does not consist of 3 integers an error is returned. Furthermore the z coordinate must be positive, otherwise an error is returned. The default light location is set to [0, 0, 10], which is right on top of the middle of the scene. An example would be light_location: ["0", 3, "10"].

6.5. Additionally Added Features

At the beginning of the project the developers translated the project specifications given by the client into requirements and tasks. By following an Agile development philosophy the developers discussed the features with the client weekly. Through these meetings with the client, the developers were able to reiterate over existing requirements, discover additional requirements and implement these requirements to satisfy the client. The following features were added throughout the project.

- The software should allow multiple images to be generated from a single request.
- The container should use the GPU for rendering to speed up the render time if it is available on the host computer, and the rendering engine supports it
- The software should create deterministic scenes, e.g. with exactly the same parameters, the software should return exactly the same image for the same JSON request that is received.
- The software should have data management to manage the multiple datasets reliably.
- The software must have performance optimizations in place to speed up rendering.

6.6. Output Example

In order to create different sort of images, the user needs to be able to specify how exactly he wants the scene to look like. Below we show a JSON object to serve as an example of what could be sent to SWaG. A request primarily contains a JSON object and this is the only permitted way to communicate with the SWaG, since this best describes the instruction for the pipeline in a human readable format for representing data [22].

```
{
    "seed": 1211121213,
    "material_types": ["hdpe", "pet"],
    "material_composition": [0.5, 0.5],
    "crowding": 50,
    "rendering_engine": "blender_render",
    "enable_gpu": true,
    "texturing": {
        "random_texture_coords_type": "UV",
        "enable_random_texture": true
    },
    "light_strength": 5,
    "background": "backgrounds/b11.JPG",
    "pattern_texture": null,
    "dataset_name": "Example_For_Report"
    "version": 1.0
}
```

The request body has been formulated using the parameters in section 6.4. The output of the request could ideally look like Figure 6.5. A clear description of each parameter is given in more detail in section 6.4.

Figure 6.5: Example output of SWaG given the input parameters specified in the JSON request above

6.7. Results

Within this subsection we'll show some output examples of our system with different parameters and show how much time it took to render the scene. All of these images were generated on a base Late 2016 Apple Macbook Pro (dual-core Intel i5, without GPU). We would like to stress that the time it took for the images to be generated could have been vastly improved when using a better machine with a dedicated GPU and more powerful CPU.

Figure 6.6: Generated image with ±50 objects (80% HDPE, 20% Cardboard) using Blender Render completed within 18.43 seconds. Moderate light strength (5) with random textures.

Figure 6.7: Generated image with ±100 objects (80% HDPE, 20% Cardboard) using Blender Render completed within 34.12 seconds. Moderate light strength (5) with random textures.

Figure 6.8: Generated image with ±20 objects (60% HDPE, 20% Cardboard, 20% PET) using Blender Render completed within 5.99 seconds. Moderate light strength (5) with random textures applied on the objects.

Figure 6.9: Generated image with ±50 objects (25% HDPE, 25% Cardboard, 25% PET, 25% Aluminium) using Blender Render completed within 22.42 seconds. Moderate light strength (5) and without random textures.

Figure 6.10: Generated image with ±50 objects (50% HDPE, 50% PET) using Cycles completed within 3 minutes and 13.62 seconds. Moderate light strength (5) with random textures.

Figure 6.11: Generated image with ±50 objects (50% HDPE, 50% PET) using Blender Render completed within 22.94 seconds. Moderate light strength (5) with random textures.

7

Image Scene Features

In this chapter the developers explain the key features of the parameters that were implemented. In order to enable these user functionalities it was essential to ensure that all parts of the Blender pipeline were automated with Blender-Python code or BPY. Allowing the user to send a JSON request with their particular custom settings. After the research phase it was important to have an ongoing discussion with the client about these parameters during the weekly meetings. This enabled the developers to ensure that the features were exactly how the client wanted them. The parameters have been designed and implemented to allow the user to adjust the images being generated to their particular needs. First object colors and textures are investigated, as these features are critical to make the images look as realistic as possible. Next the physics behind the object placement and scattering is discussed. An important factor to take into consideration when generating a scene is how the objects are interacting with each other. As often it can be the case that the objects intersect with each other in unnatural ways. The developers also discuss how to allow the scene lighting to be customizable and what the benefits of adjustable light intensity and light location are for the user who wants to mimic their lighting setup that is found in the real world. Finally, adjustable camera height and object scaling is discussed along with debugging functionality that makes it easier for the user to get the exact output their heart desires.

7.1. Colors

Often objects of the same material type have a similar color. Concretely aluminium objects will normally have a chrome/silver color whilst glass such as the common wine bottle will have a an amber or dark color like green/black. In order to logically apply colors to objects and introduce randomness, the objects are first assigned multiple coloring methods by a human who can judge which colors are applicable for that particular object. This ensures that objects such as cardboard are not given an unrealistic color such as red for instance. Furthermore, the colors themselves are also adjustable with a configuration file. At run time, the objects are randomly given a color from their appropriate color pallet.

As shown below, scenes of trash on the conveyor belt at the recycling plant are often very compact with lots of objects on top of each other. The color functionality creates a small amount of variance from image to image enabling the dataset to have a greater variance. This helps develop the robustness of the model being trained to identify and classify trash.

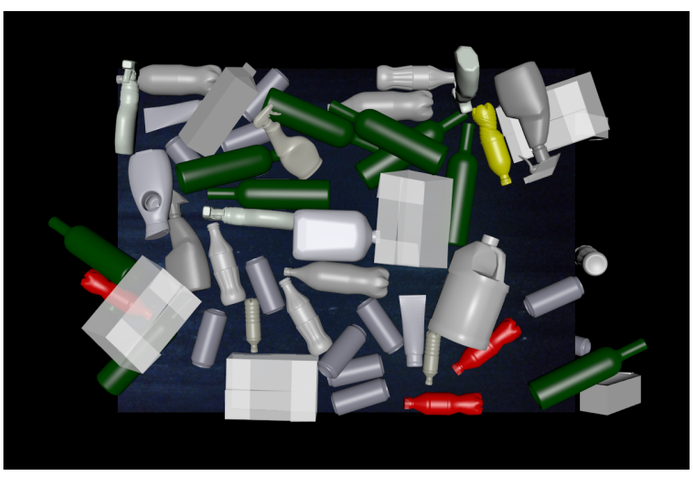


Figure 7.1: Example of generic object colors

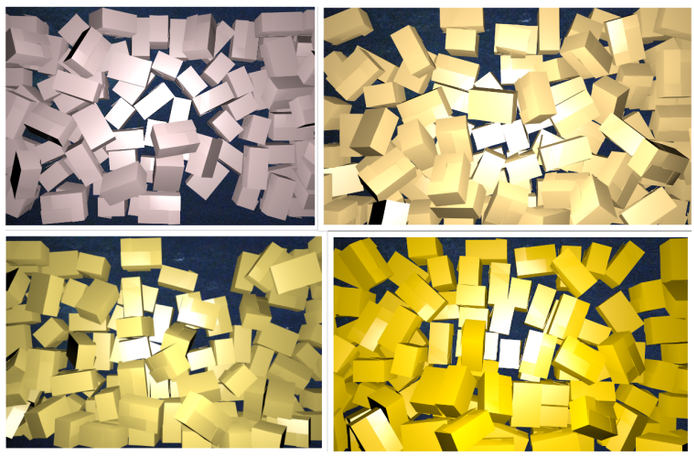


Figure 7.2: Example of cardboard colors

Figure 7.3: Yellow and Red bottle color

Figure 7.4: Blue and black bottle color change

7.2. Textures

Ye and Lewis [48] researched how textures can be effectively used to enhance the "visual realism of computer rendered images". The developers believe that procedural textures can provide more realistic images. Ideally the user can decide if they want procedural textures such as checkerboard patterns shown below. As discussed, Blender has support for many different file types. In practice it is easier to limit the number of formats as each one has its differences. For this project, it was decided to focus on the .OBJ file type. The benefit of this datatype is that it is a common format and will allow 3D model designers to easily export their work into the pipeline. By UV mapping the textures in Blender and saving it in the appropriate way, it is possible to also generate a .MTL or material file. The object can then be directly imported with its associated material or texture into the pipeline and render images of branded objects.

Figure 7.5: An example of an image with objects that have textures.

7.3. Object Scattering

Within waste facilities products on a conveyor belt are not placed in any particular way. They are essentially scattered on the conveyor belt. In order to create a realistic look that is similar to an actual conveyor belt two approaches were investigate: manual placement and placement using physical simulation.

Manual placement The manual placement of products approach is based on the notion that products can be placed next to each other and achieve a result that is sufficient for the learning purpose albeit does not create a realistic waste disposal conveyor belt look. This approach is computationally inexpensive as products can essentially be placed next to each other (on their side row-after-row). However, the downside with this approach is that it does not produce images in which products are scattered as would be expected in actual conveyor belt which might affect model training. Furthermore, placing objects perfectly on their side row-after-row might result into product collision once we start to rotate object to create a rotation in variance in the model learning. In order to remedy these issue we enter physical simulations.

Placement using physical simulation The placement approach using physical simulation can scatter objects on a conveyor belt using simple rigid bodies in a scene. The approach works as follows. Firstly, objects are placed with random orientations and random locations directly above the conveyor belt. Secondly, for each object we apply an active rigid body and enable collisions using Blender. Now at this point all products are essentially suspended above the conveyor belt plane and colliding in different places (as we placed them at random without regards for possible collisions). Within Blender physical simulations are realized as animations, hence we have to "play" the animation to achieve the desired output. This is accomplished by going through the animation frame-by-frame up to a certain point. As soon as we go through to frames (e.g. play the animation) objects fall on the conveyor belt plane and collision are naturally resolved as object cannot intersect because of the aforementioned collision settings. This approach achieve result similar to Figure 7.5 (the placement here was produced using the physical simulation approach). Despite the promising results of this approach there are two unfortunate draw backs. The first drawback is that physical simulation animation is inherently slow and cannot possible allow for parallelization. Physics simulation have to be determined frame-by-frame and there is no way to only compute the final frame without having to compute all the frames that came before, hence parallelization cannot be used to speed up the process. The second drawback is in determining how to pick the last frame. The last frame should ideally mean that all products have fallen and are now in a stable position. This can be determined using spatio-temporal comparison that compare the position of an object in one frame with the next frame to determine if the object is still moving (not in stable position yet). However, such approach is computationally expensive in comparison with heuristically picking and end-frame based on the number of objects in a scene. Because of its computational simplicity the heuristic approach has been employed to determine the final end-frame. The heuristic approach simply determines the end-frame based on the amount of objects within the scene, as in general the more objects, the more physical interactions occurs and hence the longer time it will take before all objects are stable. Despite the first drawback that was mentioned the approach using physical simulations has been implemented.

7.4. Lighting

The software our client uses to detect what kind of waste is laying on top of the conveyor belt must be able to be used on all sort of different conveyor belt setups with different lighting conditions. For the AI they use to detect this, it must be trained to also work in different lighting conditions, instead of only a single lighting conditions for all rendered images used to train the AI. Thus the software which creates synthetic images, that is used to train the AI, must be able to simulate these different conditions. The client wants the program to place lights at different locations, alter the brightness of the image, and to implement different reflections for different material types. Aluminium for example reflects more light than cardboard.

7.4.1. Light Intensity

In Blender, the intensity of a light sources can be changed. With this option it is possible for the user to simulate different brightness conditions around the conveyor belt setup by sending different values for the light_intensity parameter. Higher values can be used for more bright conveyor belt setups and lower values can be used for more dimmer conveyor belt setups. An example can be seen in the figure below, where in the left image a lower light intensity is used, and in the right one a higher light intensity is used.

Figure 7.6: Scene with a low light intensity

Figure 7.7: Scene with a high light intensity

7.4.2. Light Location

Our client also wants the light source to be placed on different locations, since in different conveyor belt setups, the light that illuminates the conveyor belt may be placed on different locations, which in turn creates a differently lit conveyor belt. In Blender, a user can change the location of each and every object with a list of length 3, that represents the x, y and z coordinates of the object. With the help of this feature, we are able to place to light source at different locations to simulate different conveyor belts setups. However lighting works different in Blender Render and in Cycles, since Cycles uses actual ray tracing (as described previously). This means that in Blender Render it does not matter whether you place the light outside the surrounding box or inside the surrounding box, the scene will be illuminated if the light strength is sufficient and is not placed too far from the actual scene. But in Cycles, these light rays are blocked by the surrounding box, meaning that it does not matter how high the light strength is, the light rays never reach the scene if it is placed outside of the surrounding box and the scene will remain dark if this is the case. Only if the light source is placed inside of the surrounding box and the light strength is sufficient, the scene will be lit up. In the figure below, the differences with lighting locations can be seen. In the left image the light source is placed in the top-left corner of the scene and in the right image the light source is placed in the bottom right corner of the scene. The light location can be altered in the request with the light_location parameter. This must be a list of length three, were index zero represents the x coordinate, index one represents the y coordinate and the last index represents the z coordinate of the light source.

7.4.3. Reflections

As explained earlier, an aluminium can reflects more light than a cardboard box. Without changing any settings in Blender, this will not be shown in the rendered image, however this process has been automated. As explained before, light behaves differently for Blender Render and Cycles. Whereas with Blender Render it is very easy to change the reflectivity of objects, it requires some work with Cycles, because Cycles works in a much different way. In Blender Render reflectivity can be changed with changing some settings of the material of an object. Namely the specularity intensity and depending on the used specularity setting an additional setting, which is in our case specularity Toon smoothness, since for the specularity algorithm we use Toon, because it gave us the best result compared to the other algorithms. But with Cycles a new 'Principled BSDF' Shader needs to be created and added to the material of the object. This shader has a lot of settings, but when you change the Metallic attribute to together with the roughness attribute of the shader, different specularities can be created. This is more trial and error rather than there being an exact way to get the most realistic specularities (or reflections). Rather the specularity values where changed until a realistic result was achieved. This was done for each of the material types. For example an aluminium can reflects a lot more light than cardboard. And a plastic bottle is somewhere in the middle of those two material types. In the image below there is a clear difference between the cardboard box and the aluminium can on the left side of the image. The aluminium can is brighter and reflects more light than the cardboard

Figure 7.8: Light located at the bottom right corner of the scene.

Figure 7.9: Light located at the top left corner of the scene.

box. In this scene the light source is placed right on top of the middle of the scene, hence why the cardboard boxes in the middle of the scene reflect so much light.

Figure 7.10: The material type of a given object defines the specular reflectivity.

For both of the rendering engines and for each of the available material types in the models directory we experimented with different specularity values to get the most realistic result. We put all of these different values into a JSON configuration file. Each time a texture or color is applied to an object, a specularity is added to the object as well according to the settings in the JSON configuration file. This configuration file can be changed, and if more material types are added to the materials, the specularities of these new material types can also be easily added.

7.5. Conveyor Belt Background

Different recycling plants have different conveyor belts with differently colored conveyors. Even at the same recycling plant different conveyor belts may be used (smaller and larger ones), and even on the same conveyor belt there may be some difference with one part of the conveyor and another. One part may be very dirty and have stains, whereas another part of the same conveyor belt may be much cleaner. Therefore the client wants the option to set different backgrounds for each of the scenes. The way we do this is by applying textures to the plane on which the objects lay on top off. This texture is an image texture, and the source of this image can be changed by giving different values for the background key in the JSON request. By changing the source of the image texture we can simulate different conveyor belt setups and the software of the client will this way also be trained for other conveyor belt setups such that it can work more effectively with greater accuracy. These backgrounds must exist in the backgrounds folder in the root of the project.

7.6. Camera and Surrounding Box

7.6.1. Camera

The camera is of course an important component of the system as it actually captures the final result. To simulate the actual cameras in the waste facilities, the camera properties are copied as much as possible. That means the the orientation of the camera in the Blender scene is on top of the conveyor belt pointing downwards so that it has a top view of the belt. Next, the camera has 3 variable heights as the ones in the facilities are either at one, two or three meters above the conveyor belt. See figures 7.11, 7.12 and 7.13 for the different camera heights.

Figure 7.11: Image rendered at camera height 1.

Figure 7.12: Image rendered at camera height 2.

Figure 7.13: Image rendered at camera height 3.

7.6.2. Surrounding Box

In order to get a random placement of objects on the scene, objects are placed above the background plane and "dropped" by the means of physics. To prevent these objects from falling over the sides of the background plane, there are four additional planes surrounding the background plane. This 'box' that is created is referred to as the surrounding box. If the camera 'zooms out', meaning the height is increased, the camera will have a larger view of the background. But as long as the surrounding box its size is not adjusted, the field of placement, i.e. the part of

the scene on which objects are placed, will be limited to the size of the surrounding box. This results in images that have an "border" without any objects. See Figure 7.14. To solve this, the surrounding box size needs to be adjusted. The surrounding box should actually be as big as the camera view, so the part of the background that is visible by the camera at any height. Since the camera height is known, the dimensions of the frustum can be calculated relative to that specific height according to the following equations [14].

$$f_h = 2.0 \times cam_h \times \tan(fov \times 0.5) \tag{7.1}$$

and

$$f_w = f_h \times res_r \tag{7.2}$$

where f_h is the frustum height, cam_h is the height of the camera, fov is the camera field of view, f_w is the frustum width and the res_r is the resolution ratio, which is the resolution width divided by the resolution height. Once the dimensions of the camera view are calculated, the planes of the surrounding box can be set accordingly and ensure that the surrounding box size scales with the camera height. Also the initial object placement above the background plane is specified by a rectangle in which object are placed randomly but still limited to the borders of this rectangle. Since the dimensions of the camera view have been calculated, this 'placement box' can be scaled automatically with the camera height. This way it is ensured that objects are placed all over the camera view. Because the object placement, described in section 7.3, makes use of the surrounding box, adjusting it causes the physics simulation used for object placement to naturally result in a different orientation of the objects. See Figure 7.14 for an example of a smaller surrounding box. There is also the option to hide the surrounding planes in the rendered image. If one chooses to set this option to "false", which means not hiding the surrounding box in the rendered image, the surrounding planes are shown and actually depict the field of placement (given the surrounding box size is smaller than the camera height). See Figure 7.15.

Figure 7.14: Image rendered at camera height 3 and (hidden) surrounding box size 1.

Figure 7.15: Image rendered at camera height 3, with a visible surrounding box of size 1

8

Crushing

Within waste facilities products naturally are not in their original state. After being thrown away disposed products often undergo some level of crushing to save space. This product crushing however poses a challenge for model to be trained. If the model is only trained on products that are in a good state it will not be able to recognize products that are in a crushed state.

8.1. Soft Body Physics

Automating crushing was accomplished by applying soft body physics to an object. This is not as simple as it sounds. The soft body settings within Blender consists of many different parameters and finding the right parameters that were generalizable enough to all objects took a significant amount of time. With this said, the found values cannot always work perfectly for any material type. For example, plastic objects do not crush like cardboard objects. We recognize this limitation of the system. The settings have been carefully chosen and we provide a justification for each setting in Figure 8.1.

Soft Body settings	Explanation	Chosen setting value	Justification
Weights > Gravity	Gravity of the object	0.0 (default 1.0)	Higher gravity values can result into objects being squised by gravity or floating in the scene. Setting the object to zero effectively eliminates worrying about gravity.
Enable Goal	The “goal” is the desired end position for vertices based on this animation	FALSE (default TRUE)	When deforming there is no desired end position for the object to be in.
Edges > Pull	How much the edges are allowed to stretch	0.6 out of 0.99	The pull value results into edges not stretching too much during deformation which result into unrealistic deformations. Higher values result in objects being less easily strechted, lower values can cause objects to not strech enough (especially smaller objects).
Edges > Push	How much the soft body resists being scrunched together	0.8 out of 0.99	The high push value helps emulating the stiffness of objects.
Edges > Plastic	Permanent deformation of the object after a collision	100 out of 100	Objects should be deformed as much as possible when collision occurs. Hence, the highest value was chosen for this setting.
Edges > Bending	Bending stiffness of object	2.5 out of 10	The object stiffness provides standing stability for the object. Higher can make the object too stiff, and lower values can result into the object collapsing on itself.
Edges > Stiff Quads > Shear	This stops quad faces to collapse completely on collisions (what they would do otherwise)	0.5 out of 1.0	The object stiffness provides a reasonable amount standing stability
Edges > Collision > Edge	Checks for edges of the soft body mesh colliding	TRUE (default FALSE)	Enabling this results into more realistic object self-collisions when objects get crushed and "pushes" a part of itself onto itself.
Source for each setting explanation: https://docs.blender.org/manual/en/latest/physics/soft_body/settings.html			

Figure 8.1: Justification for each setting. Source for each setting explanation: https://docs.blender.org/manual/en/latest/physics/soft_body/settings.html

8.2. Automated Crushing

After the soft body physics settings have been applied an object is deformed by either two hitter cubes that hit the object from one side and the respective opposite side, or the object is deformed from the top. See Figures 8.2 & 8.3 for an example of how such a crushing process looks like. This process is repeated a specified number of times for each object with random variation in how much the object is rotated, how much the hitter cubes dent the object, and how fast the hitter cubes move. The random variation is all derived from a single seed that can be specified by the user.

8.2.1. Limitations of Automatic Crushing

Ultimately, crushing any model using the developed technique depends on the model. Often times, a model consists of many disconnected meshes, which might not seem like a problem initially, but it so happens that soft bodies physics strongly dependent on the mesh(es) of the object to deform the object. As a result of these disconnected meshes any soft body deformation will quickly fall apart or give undesirable results. Furthermore, objects that are not structurally sound also risk falling apart. Another limitation of automatic crushing is time. The more

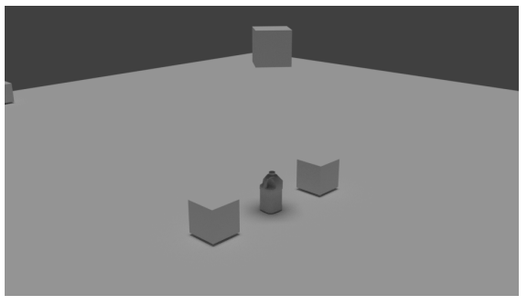


Figure 8.2: Right before a crushing occurs.

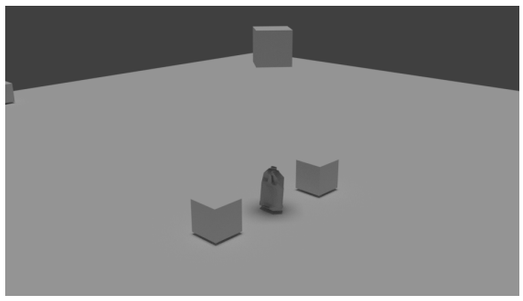


Figure 8.3: After crushing has occurred.

triangles[1] exists in an object the more time the crushing process will take, as any deformation to an object implies manipulating the triangles that make up the object. Naturally this implies the more triangles an object has, the more of them need to be manipulated and hence more time is needed to manipulate them. That is, the greater the amount of triangles in an object the more time will be needed to crush it. From these limitations we formulate three requirements that objects should have in order to be properly crushed by the system.

1. An object must be well-connected. That is, it only consists of one single mesh.
2. An object must be structurally sound.
3. An object must not contain an unrealistically high amount of triangles. (This requirement will be ultimately depended on the machine on which the crushing is executed as having lots of computing power helps.)

Within the developed crushing system the first requirement can be treated leniently. When objects consists of too many meshes they will be ignored because the crushing system will not be able to generate a proper result (it will likely fall apart before any crushing can even occur). However, the systems allows the client to specify how many disconnected meshes an object is allowed to have, in this way leniency for this requirement is provided. The leniency is necessary because certain object may depending on the object still give a coherent crushing despite not being well-connected. In order to lessen the burden on the client to realise the third and final requirement the systems allow for automatic decimation of objects that have more than a specified number faces (which in turn also reduces the number of triangles). Furthermore, the system also provide the converse feature. When objects have too few faces the crushing process can result into objects that have gone through the crushing process but have not been sufficiently crushed as there are simply not enough triangles within the mesh(es) of the object to create deformation. Hence, the converse feature allows for automatic division of objects that have less than a specified number of faces.

8.2.2. Shrink wrapping

In an attempt to eliminate the first and second requirement (and arguably even the third requirement) shrink wrapping can be applied to an object. Shrink wrapping is the process of applying a larger object to shrink to a smaller object, so that a fitting wrapping using the larger object is placed around the smaller object. The larger object that is wrapped around the smaller object can be guaranteed to be a single mesh, and a certain level of structural soundness can be enforced. While this may seem as a plausible solution to the eliminate the first and second requirement is comes with some serious drawbacks. The first concern arises when dealing with objects that contain "holes" (for example, a bleach bottle with a handle), after shrink wrapping these "holes" are no longer apparent. Even objects without "holes" lose some precision in their geometry. A second concern arise with the textures. The shrink wrapped object has no texture mapping information and the textures that were available to the original object are lost. There does not exist a way for the texture mapping of the original object to be applied to the shrink wrapping as textures are highly dependent on the original object meshes, but those are not available in the shrink wrapped object. If that information was available then, either the object was not shrink wrapped or a different process was applied. As a result of these major drawbacks, shrink wrapping looks promising at first but actually introduces new and harder problems to solve. Hence, shrink wrapping was not implemented within the system.

8.3. Pre-crushing

As a result of the computationally expensive process of crushing objects, it does not make sense to start crushing objects when handling a request. Instead, a much more efficient process is to create a number of crushed objects before any requests is made. Hence, products are crushed beforehand to save time and crushed models are essentially treated as any other model within the system. The client can specify what seed should be used during the pre-crushing, how many crushed variations should be created for each object, the decimation and division threshold values, and the well-connectedness leniency threshold.

[1] All 3D models consist of triangles.

9

COCO Dataset

9.1. A Brief Introduction to COCO

"The Microsoft Common Objects in Context (MS COCO) dataset contains 91 common object categories with 82 of them having more than 5,000 labeled instances, In total the dataset has 2,500,000 labeled instances in 328,000 images" [28]. COCO has fewer categories, but more instances per category. This can aid in learning detailed object models capable of precise 2D localization. In other words COCO is a "large-scale object detection, segmentation, captioning dataset" [28] that provides a concise way on how to structure your datasets. In our pipeline we have used it to organize the datasets for object detection. Each dataset contains n images and a single JSON file containing all of the required image annotations for classification. Of course to even provide such a functionality without adding extra manual overhead, it was necessary to incorporate an automatic way to know what an object in the image is and also how to extract the correct annotations for that given object that can then later be placed in the correct location in the dataset. All 3 above mentioned problems were solved in our pipeline. 3D models are created by the user and given their object name, objects in the scene are masked, and the correct bounding box, segmentation and other annotations are calculated.

9.2. Masking

A COCO dataset contains the pixel location of each object in its annotations and since those boundary pixels are not known yet, they need to be acquired somehow. This is where masking comes into play. A mask is the area (2D) of an object as it is visible in the rendered image. Using the COCO API the boundary pixels can be obtained from a mask of an object. So at this point, first the masks of the objects in the scene must be generated so that afterwards the COCO API can generate the annotations. Since Blender is used as the graphics engine, its 'Compositor' functionality can be used to achieve this. The Compositor is used for compositing, which is combining multiple images or effects into one final image. This particularly comes in handy in this case since the masks of each individual object need to be merged in one final image for the COCO API. Blender's Compositor uses a 'Node tree' as an architecture. Now it is a matter of connecting the correct nodes and setting the correct node properties.

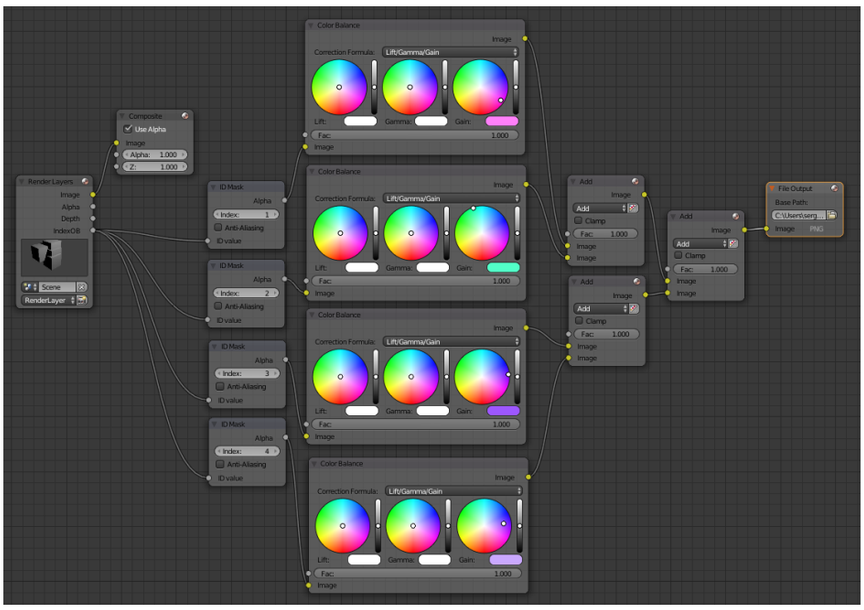


Figure 9.1: Blender Compositor Node tree

Figure 9.1 shows an example Compositor node tree for four objects. The third column of nodes (from the left) are the 'ID Mask' nodes which are unique nodes per object in the scene and actually also nodes that generate the mask of that object. Afterwards these nodes are each linked to unique 'ColorBalance' nodes (fourth column), which contain a color that is assigned to that specific mask. Finally all these colored masks need to "merged" into one final image. This is done through linking the 'ColorBalance' nodes with several 'MixRGB' nodes, which in turn are again merged together by other 'MixRGB' nodes. This process keeps on going until all masks are merged into

one image, which then is saved through the 'FileOutput' node. In practice this tree and its nodes scale with the number of objects in the scene. An example output is the masks image in figure 9.3 generated from the scene in figure 9.2.

Figure 9.2: Original rendered image with objects on a conveyor belt.

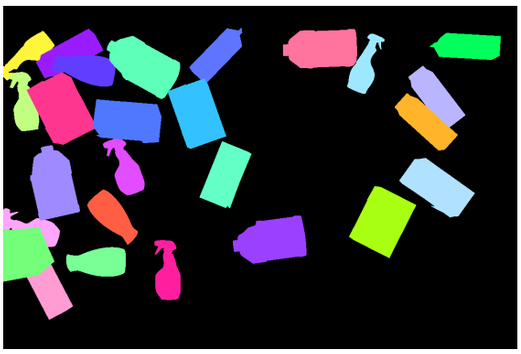


Figure 9.3: Colored object masks. A mask of each object is displayed with a unique color.

9.3. Image Segmentation

Moving to the next part in the pipeline, it is important to realize that the masking algorithm gives us 2 important pieces of data. A dictionary containing unique RGB colors as the key, the values containing the id of the given object and the image on disk. It is now possible to move forward with the image segmentation algorithm. The image segmentation algorithm also keeps account of the object name, the object id, and the image information. This will prove to be useful, due to initially before calling the masking algorithm we pass it a dictionary containing the names of the objects as the key, and the id as the value. The information is complete, and we are now able to correctly create the annotations for each object in an image. The last step is now to create the COCO JSON file that follows the object detection format. Below this JSON file can be seen, but simplified. It merely shows the basic components of such a COCO dataset and what such a dataset should contain, this will be discussed shortly. The rendered image can be seen in Figure 19, and the COCO JSON annotations applied to the same image in Figure 20. This image was generated using Jupyter Notebook [36] as a verification method.

Figure 9.4: Original rendered image with objects on a conveyor belt. No bounding boxes or segmentation's are visible.

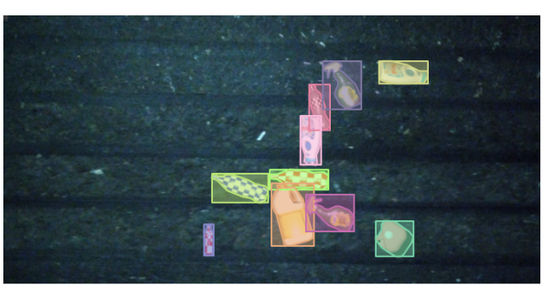


Figure 9.5: Object detection image with objects on a conveyor belt. Here the bounding boxes and segmentation's are visible.

9.4. The COCO Data Format

```
{
    "annotations": [{
        "category_id": 1,
        "bbox": [311.5, ..., 98.0],
        "id": 1,
        "segmentation": [[339.0, ..., 173.5]],
        "area": 2401.25,
        "iscrowd": 0,
        "image_id": 1}, ...
    ],
    "categories": [{
        "id": 1,
        "name": "coca_bottle",
        "supercategory": "glass"}, ...
    ],
    "info": { ... },
    "licenses": [{ ... }],
    "images": [{
        "height": 400,
        "width": 800,
        "id": 1,
        ...,
        "file_name": "image1.png"}, ...
    ]
}
```

To enable the annotations for Figure 34, it is important to have the COCO JSON file at hand. This allows through the use of the COCO API [29] to show the generated annotations for the image, though to be clear this is not incorporated in the pipeline. This is a mere visual representation of how the annotations generated by the pipeline for the given image looks. The annotations key is a list of dictionaries containing the annotations, in particular the segmentations of the objects in an image. Each annotations is referenced by an image through the image_id. The categories key is a list of dictionaries and contains what categories the objects in the image belong to are and what the object is. The info key contains information pertaining to the COCO JSON file and licences for the given COCO JSON. The final key, in particular the images key is a list of dictionaries containing information of all the images and where the image can be found on disk.

10

Dataset Management

As requests to the pipeline build up, the issue of previous datasets being overwritten or data loss arises immediately. This is a critical issue and needed to be resolved in the best elegant manner without adding extra overhead to the pipeline. One of these overheads and a point to stress on, is the issue with data duplication. If a dataset contains one single image and a single JSON file, the disk usage is small, but as more data is generated the disk space usage becomes problematic. Finding an efficient and robust way of handling disk space and data loss is the core functionality of the Dataset Management.

10.1. Solution Approach

The initial idea was to incorporate a NoSQL database into the pipeline. This was due to the fact that NoSQL databases (aka "not only SQL") are non tabular, and store data differently than relational tables [19]. The main types and the most relevant for the pipeline are document, arrays and key-value types. This is the exact structure used in the COCO JSON. The only main problem with this approach was that the JSON files would have to be saved twice. Once on disk (COCO JSON) and once in the database itself, which in turn is also saved on disk, but in a format that the database uses. The research focus was then placed on finding a way to re-write specific parts of the NoSQL database driver in order to incorporate a more efficient way of data storage. In other words, using the same COCO JSON as a database file and using the database's built-in concurrency handling functionality. This has proven to not be very trivial and would require extra time not available in the current time frame of this project. The idea than shifted to using a very simple approach. Mainly, using the built-in sqlite3 database in Python. This new idea would prove to be trivial and very robust. The functionality would rely on a lock/unlock mechanism. Datasets can now be locked and unlocked based on a first-come-first-serve approach.

10.2. Multiple Request Handling

It is required that the system can handle multiple requests that are either trying to create new COCO datasets or simply extending existing ones. In order to do this, a simple mechanism has been devised using Python's built-in sqlite3 database. The database would contain a single table called ***datasets***. The schema of the table contains 3 fields. These are: dataset_name, locked, and image_id. The dataset_name is the name of the COCO dataset, the locked field is true for a dataset that is currently being processed and false for the opposite, and the image_id is the highest image id number in the current dataset. The latter is used to start rendering an image from $max(id) + n$, where n is initially 1 and is incrementally increased by 1. As requests come in, it is the responsibility of the Dataset Management class to verify through the help of the database if a certain action is allowed. If a COCO dataset exists and it is unlocked, the request is accepted and the COCO dataset can then be extended. The dataset is locked immediately. The opposite state of this would deny any incoming request wanting access to the dataset. This simple approach now allows running pipeline processes to finish working on a particular dataset and protect against datasets getting overwritten.

10.3. Auto-generate Categories

This functionality allows the user to auto-generate a complete categories list of the models folder. As previously discussed in paragraph 8.4, the COCO dataset contains a categories key. This categories key is normally defined by the user through manual means. Due to the models folder increasing in size through the addition of new models, it is very difficult to manually update the categories for a given COCO dataset, or in particular what category a specific object belongs to. This manual process has been automated in the pipeline. Users are now able to get the categories generated automatically and appended to a newly created dataset without any extra hassle. This in turn is another elegant approach for automating as much as possible of the synthetic data generation process.

11

Rendering Speed Optimizations

Scalability and rapid data delivery are key product requirements for the synthetic data generator. Therefore the developers have tried to design and implement the pipeline in the most efficient way possible. In this chapter we will analyse some of the data such as speed ups that have been achieved by conducting optimization research experiments.

11.1. Tile Sizes

When Blender renders a scene, it divides the to be rendered image into several squares and it then renders each of these squares to get to the final image. Varying the sizes of these squares can have a positive impact on the render time of the scenes [35]. To optimize rendering times we tried out several tile sizes for our CPU and GPU. These different tile sizes were tried out on an image that has a resolution of 1200 by 800 pixels, since our client is primarily going to use this resolution to train his R-CNN with. The reference we found this information from advised to keep the tile sizes in power of 2's. Tile sizes were varied until a tile size of 4096x4096 because there is no significant difference anymore in render time starting from a tile size of 1024x1024. This tile size experiment was conducted on a laptop with an Intel i7 7700HQ, with a mobile NVIDIA GeForce GTX 1050 graphics card.

Tile size	Blender Render	Cycles on CPU	Cycles on GPU
1x1	00:06.12	00:50.39	10:40.00
2x2	00:06.16	00:51.40	10:20.00
4x4	00:06.33	00:52.82	10:17.00
8x8	00:06.21	00:52.07	10:16.00
16x16	00:04.11	00:50.12	5:30.00
32x32	00:03.52	00:46.89	02:07.88
64x64	00:03.45	00:46.07	00:51.15
128x128	00:03.34	00:46.71	00:36.61
256x256	00:03.26	00:50.10	00:32.54
512x512	00:03.44	01:15.2	00:31.80
1024x1024	00:04.63	2:50.00	00:32.04
2048x2048	00:04.95	3:28.00	00:31.92
4096x4096	00:04.98	3:25.00	00:31.98

Tile size	Blender Render	Cycles on CPU	Cycles on GPU
3x2	00:06.13	00:48.65	10:23.00
6x4	00:06.20	00:49.75	10:12.00
12x8	00:05.22	00:51.07	8:15.00
15x10	00:04.61	00:49.00	6:44.00
24x16	00:03.82	00:47.78	4:17.00
30x20	00:03.65	00:45.30	3:24.00
48x32	00:03.72	00:44.48	1:34.32
60x40	00:03.37	00:43.54	1:09.92
75x50	00:03.34	00:45.83	00:54.00
120x80	00:03.28	00:44.17	00:40.15
150x100	00:03.27	00:43.90	00:37.85
240x160	00:03.32	00:45.84	00:33.28
300x200	00:03.37	00:48.87	00:32.98
600x400	00:03.36	00:59.35	00:31.98
1200x800	00:04.86	3:25.00	00:32.07

Figure 11.1: Rendering times for each of the rendering engines with different tile sizes. The time format is minutes:seconds.milliseconds

The table with the results can be seen in the upper figure. The time format is minutes:seconds.milliseconds. The red cells indicate runs that were not fully run until the scene got rendered. These runs were stopped after 2 minutes and the remaining time to render the scenes, as given by Blender itself was added to these 2 minutes. These runs were stopped after 2 minutes because otherwise they will take a lot of time to render, without giving any more useful information since for the same scene shorter rendering times were already achieved during experimenting. As one can see from the table, the optimal tile size for Blender Render is around 32x32 to 512x512. For Cycles on CPU it is around 32x32 to 256x256. For Cycles on GPU we see that the rendering times keep decreasing when the tile size is increased, until a certain point of around 512x512. After that the performance is more or less the same and there is no significant change. Instead of choosing tile sizes in the power of 2's, and instead of only having square tiles, we were wondering if there would be any performance increase if we take tile sizes which can divide the entire to be rendered scene in even tiles, which are not squares. This way you wont have tiles on the edges of the scene which only render a smaller piece of the total render, instead of its actual tile size. To explain this a little bit more, lets say that we have a scene of 1200x800 pixels with a tile size of 128x128. 1200 divided by 128 is not an even number, and the same holds for 800 divided by 128. But if we take a tile size of 300x200, we can divide the entire scene into 16 tiles, which will all have an equal amount of pixels to render. The results can be seen in the right table in the figure above. As one can see by comparing both tables there is not really a performance increase because of this. The best time in the left table for Blender Render is only 0.02 seconds slower than the best time in the right table, which is negligible. For Cycles on CPU however there is a 2.17 seconds performance increase. The best time in the left table is 46.07 seconds, whereas this is 43.90 in the right table. But percentagewise it is only a 4.7% increase in render speed, which is also negligible. And the same holds for Cycles on GPU, where best times only differ within 0.18 seconds, which is also negligible. So to conclude there is not really a performance increase

when the to be rendered scene is divided into squares which all have the same amount of pixels to be rendered. To conclude, the tile sizes of the software will be set to 64x64 for Blender Render and Cycles on CPU (since there is not really a huge different in time between a tile size of 32x32 to 512x512 for Blender Render and 32x32 to 256x256 for Cycles on CPU). And for Cycles on GPU a tile size of 1024x1024 will be used, since the performance differences after a tile size of 256x256 are negligible.

11.2. Object Imports

After having implemented all the must-haves of our MoSCoW document, we noticed that getting a result after sending a JSON request took too long. We analyzed the cause of this and noticed that importing the objects into the scene took the longest time. For some reason importing models into a scene takes a long time with Blender. The chosen rendering engine does not matter. We also noticed that Blender re-imported the same models into a scene, that were already in the scene if they occurred more than once in the JSON request. It was not smart enough to know, to not re-import the same model more than once if it is already present in the scene. And since we already know that importing models into a scene takes a relatively long time, we knew that this process could definitely be improved. So we started on changing the code to instead of re-importing the same object more than once, to duplicate it instead to see if there was a performance increase. We tested this idea out with 100 items of the material types "glass" and "hdpe". Initially, before implementing this idea, our software would have executed the import function a 100 times, even though if the same object is already in the scene. The request eventually had a list of 10 different objects, where the occurrences of each different object in the scene differed from 5 times, to 30 times, meaning that each object got imported at least 5 times and at most 30 times. The time it took to execute the import function 100 times 38.14 took seconds. After implementing the optimization, the import function was only called 10 times, and the total time of putting all objects into the scene took 2.87 seconds. This was a huge improvement for us of 1330%. Although there was a huge improvement, we were aware of the limitations of this idea. This idea only provided a speed-up if objects occurred multiple times in a scene. If you have a scene with each object only appearing once (even though this chance is small), there is essentially no speed-up. Therefore we also came up with another idea to improve the import time of the models in another way. This is described in the next section.

11.3. Scene Pre-loading Optimization

In order to save time during handling requests we create a single scene Blender file that contains all objects and simply load that file for each request. The creation of this file is straightforward, all the existing models are simply put into this one single file. The advantage of putting all the model in file is that no time needs to be spend on importing the model into Blender as such process often spends notable amount of time on converting the object. The scene pre-loading resulted in a 20% decrease in the total time to generate an image.

11.4. GPU Rendering

On default the scenes in Cycles are rendered only with the CPU of the rendering machine. But there is an option to also use the GPU to render the scenes. This lowers the amount of time it takes to render the scenes. As described earlier, if the user wants to use his GPU to render the scenes, the user must specify this in the JSON request. Unfortunately there are some restrictions with this optimization. This option only works if the rendering machine has an NVIDIA graphics card starting from the GeForce GTX 4xx (CUDA computing capability from 2.0 to 6.1) [4]. OpenCL is also supported for GPU rendering with AMD graphics cards, although only graphics cards with GCN architecture 2.0 and above are supported [4]. Also it does only works for Cycles and not for Blender Render. And this optimization only works to render the final image, and thus it does not lower the physics rendering speed. Nonetheless, with an optimal tile size there is an approximately 15 seconds time improvement compared to CPU rendering with an optimal tile size as well (with the same scenes).

12

Refactoring, Code Quality, and Project Management

12.1. Continuous Integration/Continuous Development

Implementing a feature is one thing, but what has to be kept in mind is that the code of the implementation must also be readable and understandable by others who are not as familiar with the code. The code also needs to be tested to make sure it is working correctly. The implementation may work with a certain set of parameters, but not with others. Also the code may change over time, and with these tests it is possible to see if the implementation still works correctly. In this software project we use a linter called Pylama [27], which is code audit tool that includes a large number of other tools and linters [44] to determine the quality of our code. Pylama is integrated into our Continuous Integration/Continuous Development (CI/CD), which means that Pylama analyzes all the code every time a new commit is pushed to the repository. Whenever the code quality could be improved (e.g. whenever a line is too long, or a function is too complex), the CI/CD pipeline fails and the developer gets notified about this with an email. In the pipeline the developer can then look at the Pylama section and see why the pipeline fails and where his code quality should be improved. The software also uses Sonarcloud [42], with which we are able to get instant feedback on the current code quality. Sonarcloud specifically allows us to get smart insights in the security risks and potential code smells exists using techniques that go beyond the usual static code analysis. The aforementioned tools allow us to keep a high-level of code quality while getting "smart" feedback suggestions via Sonarcloud as to what could be improved. The reason for using Sonarcloud additionally to Pylama is that this was a request from our client, since our client also uses Sonarcloud for their own code which they develop in house. To test our code we PyTest, which is a no-boilerplate alternative to Python's standard unittest module [37]. The tool is easy to use and has a lot of features including coverage reports, with which a developer can see exactly which parts of his code are tested and which ones are not, how much of the code is tested etc.

12.2. SIG Code Review and Refactoring

Furthermore, we are obliged to hand in our Python code to the Software Improvement Group (SIG) that gave use specific feedback about our code base and why we should improve. At several moments through our project we have refactored code before the hand in for SIG. After we got our feedback back, we saw that our code quality was not too bad, but there were some notable improvements to make. Our score across the different categories can be seen in the figure below.

Figure 12.1: Overall and partial score for each of the code quality categories we received from SIG

As can be seen, our overall score was 3.2 stars out of 5. We did well on volume (a little bit too good, more on this later), and duplication. At the time our code base was not so big, and we had 2 lines which occurred twice in the whole repository at the time, so cutting of 0.5 starts for that was a little bit too much in our opinion, but we eventually fixed this by moving around some lines of code, so we only need to have it once. We also did well on model

coupling. But where our code could be improved was with unit size, unit complexity and unit interfacing. We could also drastically improve our component balance, but weird enough we did not get any feedback about this, even though it is 0.5 stars. The description for it is also a bit vague, so we do not know what the problem actually is. However there was enough feedback on unit size, unit complexity and unit interfacing. In our repository we previously had a lot of long functions exceeding 15 lines of code (LOC), exceeding a McCabe Cyclomatic Complexity of 5, and we also had plenty function using three or more parameters. Although we think that 15 LOC, 5 CC and two parameters per function is a little too strict, we still refactored all of our code to comply with these rules.

Lowering the LOC per function goes hand in hand with lowering the CC and parameters per function. Our idea was to basically split up our larger functions into multiple smaller functions. By lowering the amount of parameters per function to two, there are less possible operations to do in a function. This also lowers the LOC and CC per function. For functions which exceed a 15 LOC and 5 CC, but which have two or less parameters, we used our idea of splitting the functions. We still wanted everything to be as coherent as possible, so while refactoring the code to create multiple smaller functions instead of one large functions, we tried to keep the naming appropriate, such that everybody would understand what was happening in the code. If there were multiple if statements per function which increased our CC to go over the threshold of 5, we split those if statements between multiple functions. We tried to apply these rules (not more than 15 LOC, 5 CC and 2 parameters per function) for all functions in our repository. We are very happy with the result, because in our opinion our code is now much easier to read and understand. According to PyTest all functions in our repository have a McCabe CC of 5 or less. Doing this also allowed us to write tests for a larger part of the code. Whereas our previous test code coverage was 56% according to SIG. However according to PyTest our coverage was 82%. This is because PyTest does not include modules that do not have test modules in to the coverage.

Even though we tried to test as much as possible, there were still big parts of the code that we could not test. Since we are creating rendered images for example, we cannot test whether we for example actually placed a model into the scene, or whether the background of the scene, is the right background. There are ways to test this, but that would involve image recognition, AI etc., which is a whole different project. Also there are no ways to test the Blender API functions with PyTest, since there is no way to simultaneously run PyTest (or any other testing tool) together with the Blender API. The Blender API expects a script to run, and there are no ways to see if the implementation of this script is okay. Unfortunately the only way of testing is manual testing, in which the developer implements a piece of code, and manually verifies if the implemented feature is working or not. We made sure to do this for all functions that we could not test automatically, to make sure that our code is working correctly. To these kind of functions that do not have tests we added a small comment for future developers, such that they know why a certain piece of code is not tested. In the code base there are two modules which entirely consist of function that use the Blender API, and one other module that uses the Blender API in approximately 80% of its functions.

After we doubled if not even tripled the amount of code in our repository our code coverage according to PyTest is 80%. In our opinion this is more than good, since most of the code that is not covered by tests also uses the Blender API (there is a module of which 80% of its functions use the Blender API), which means we cannot test it. If we exclude this, we get close to 93% coverage, which the team is very proud of. However, we made some exclusions for the rules that were defined by SIG. In python each function in a class has an additional parameter (self). However, this self parameter is rarely filled in by developers and not used. Therefore we do not count this parameters to the max amount of parameters per function. So for functions in classes we allow for 3 parameters instead of 2 parameters, since the first parameter of a function in a class is always self, and having a maximum of only 1 additional parameter per function would be unrealistic. Another exclusion we made is for the __init__ functions of classes. We made the exclusion that they could have more than 2 parameters, since having only two parameters per object is in our opinion also not realistic. Often more than two parameters are needed for a class, and restricting this to only two makes everything much harder to implement. Nonetheless, it also makes the code harder to understand in our opinion.

For the second SIG code review we made sure that the functions in our code did not exceed 15 LOC, 5 CC and 2 parameters per functions (with one exclusion for classes, which is named above). Even though we think these numbers are a little bit to strict, since in our opinion 20-25 LOC, 10 CC and 4 parameters per function is also okay, we still think that our code quality improved drastically and that our code is now easier to understand for anyone who is not familiar with it. Eventually after all the refactoring we did for the second review of SIG, all of our functions had a McCabe CC of 5 or less, all functions had 15 LOC or less, and only 3 functions had more than 2 parameters (excluding __init__ functions and functions in classes with had a maximum of 3 parameters, because of self), because in our opinion it would only complicate the code if we split up these functions as well to only have 2 parameters, since they were already relatively short. Unfortunately during the second SIG code review we forgot to remove an add-on for Blender from the code we gave SIG to review. Since we did not create this add-on, we also did not refactor it. But since we handed it in for the SIG code review, this file also got analyzed by SIG. And unfortunately the code quality of this file was not very good. It was in fact so bad, it lowered our score by a lot. However in the graphs in the figure below you can still see that we lowered our unit size, unit complexity and unit interfacing if you ignore the red parts in the bars. All red parts in these graphs are caused by the add-on we forgot to remove. A part of the orange bars is also caused by this add-on. If you ignore the red parts of the bars you can actually see that our score should be much higher than it is currently. Even though our end score is a 3.6, and is improved when compared to our previous score, the score would be much higher if we did not forget to remove this add-on.

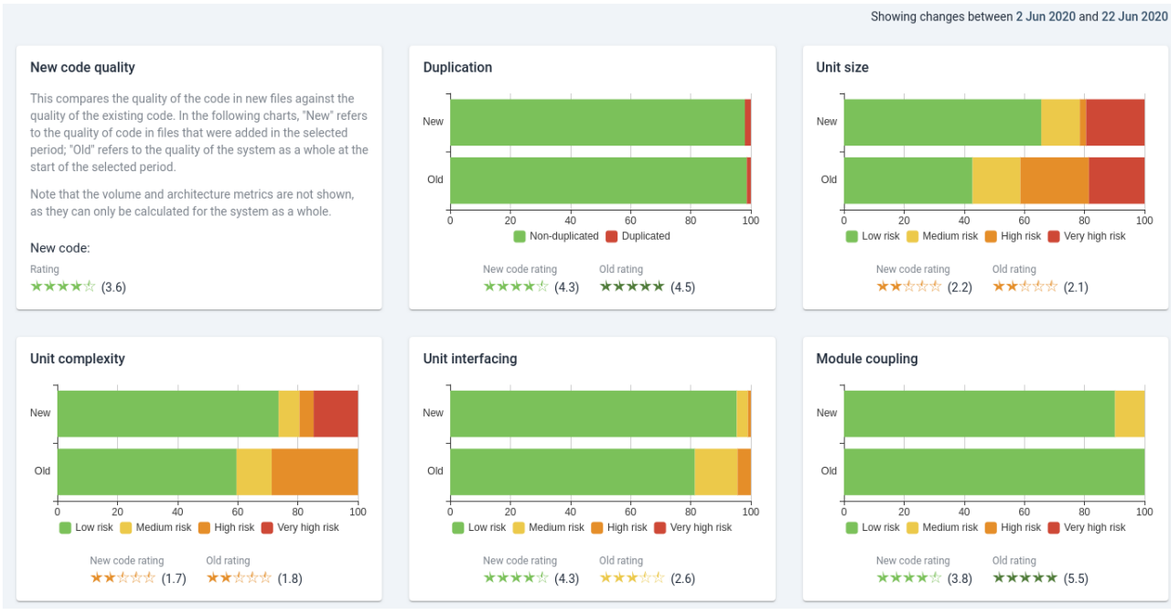


Figure 12.2: Overall and partial score for each of the code quality categories we received in the second SIG code review. The new bars are from the second SIG code review, whereas the old bars are from the first SIG review.

12.3. Experience With SIG

Even though we appreciate the feedback we got from SIG very much and with it we were able to increase our code quality, we also think that SIG could improve the feedback they give on the code. Whereas the feedback we got from SIG is more about code quality and while it is only static analysis, we also would have liked other kinds of feedback, like feedback about the code itself, i.e. if there were bad coding practices in our code, or problems with our random number generator that could cause security issues, but unfortunately we did not receive any kind of feedback regarding this. We also think SIG could improve the feedback they give, since they for example for certain maintainability categories give 5.5 stars, whereas only 5 stars are shown (as can be seen in the figure). This is clearly a bug from their side in which they give more than 5 stars, and from a company who inspects code of other companies and give them feedback we would have expected better than this.

12.4. Code Review and Project Management

Whenever a developer creates some new functionality, he needs to add this functionality in a new branch. After the functionality is implemented and tested, the developer needs to merge his code with all other code eventually. To do this, a merge request is created. This merge request needs to be approved by at least one other developer in the team. The task of reviewers according to Black, Veenendaal and Graham is to inspect the code for any bugs or defects [18]. Conducting code review was a great opportunity for other team members to comment on small bugs and also propose further improvements. Once the appropriate changes were made and the reviewer was satisfied with the code it could then be approved by the reviewer and merged into the production branch. Furthermore, code review was a useful opportunity to transfer knowledge from one person to the next about code that would then be used by the other person developing a feature on top of the current code being merged.

To create and divide the tasks, the developers used GitLab's built-in issue board to keep track of the features being implemented throughout the project. For each feature an acceptance criteria or a "Definition Of Done" was developed through team discussion and meetings with the client. This acceptance criteria was then used to ensure that the features that were implemented were doing exactly what they were supposed to do. Each issue has its own definition of done. This definition of done is also used during the merge request. The reviewer checks if all criteria in the definition of done have been implemented in the branch of the merge request.

13

Performance Measures

Within this chapter we shall discuss the performance of two important components of our system the pre-crushing and the pipeline generated images. The performance measures of these two components are highlighted because they present the largest time bottlenecks and provide insight into the general system performance. It is important for the reader to note that the time it took for the images to be generated and the models to be crushed could have been vastly improved when using a better machine with a dedicated GPU and more powerful CPU.

13.1. Performance Measure of Pre-crushing

As described in section 8.3 the software solution requires pre-crushing to speedup. The pre-crushing was executed on a base Late 2016 Apple MacBook Pro (dual-core Intel i5, without GPU). See Figure 13.1. The only parameter that varied is the amount of objects being pre-crushed for all eligible objects (See section 8.2.1 for limitations of crushing objects).

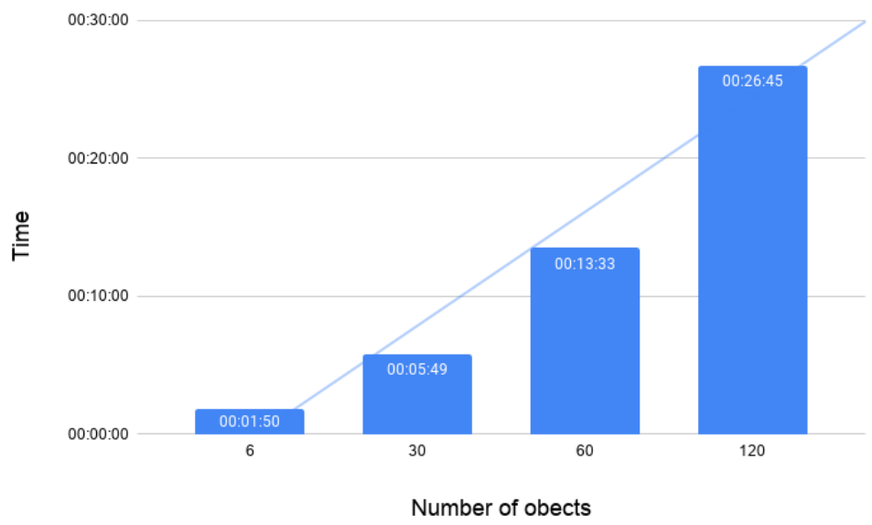


Figure 13.1: Total crushing time in seconds vs. number of objects. The trend line is the blue line going through the bars.

It takes approximately 26 minutes to crush 120 different objects. However, the amount of performance of the crushing can vary quite a bit as the amount of frames needed for the crushing is dependent on the speed of hitter cubes (see 8.2). The random hitter speed varies depending on the user settings. Within the pre-crushing measurement of 13.1 a hitter speed of 50 was used. A slower hitter speed improves performance as it decreases the amount of frames needed for the crushing.

13.2. Performance Measure of Pipeline Image Outputs

As described in section 6 the main output of our system is the generated image output. For some examples of image outputs see section 6.7. The images were generated on a Asus GL553VD (quad-core Intel i7 7700HQ, with mobile NVIDIA GeForce GTX 1050). See Figures and 13.2. The only parameter that was varied is crowding (the number of objects in a scene).

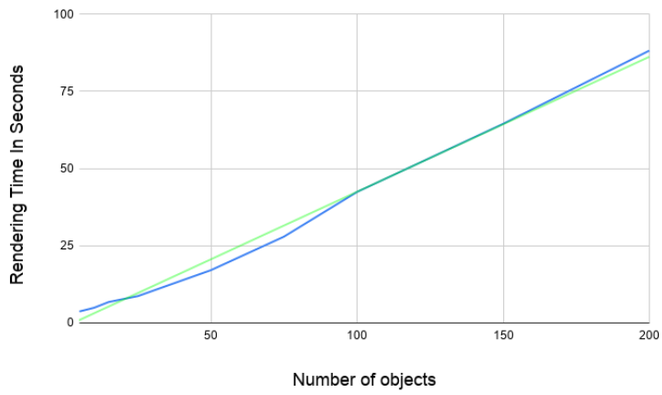


Figure 13.2: Rendering time in seconds (using Blender Render) vs. number of objects. The linear trend line is the green line.

Within Figure 13.2 we can clearly see that the rendering time difference for scenes with a small amount of objects. We note that its performance speed is linear in the amount of objects that are in a given scene. In fact, with between roughly 40 and 80 objects the pipeline image generation performs sub-linearly.

14

Conclusion and Future Work

Before the creation of this tool, it was very difficult for engineers to quickly create suitable data for training a computer vision model for classifying waste in waste facilities. This tool has therefore made it much easier for engineers to create a custom dataset for their particular needs. The software has achieved this goal by meeting the specified evaluation criteria and requirements that were set at the beginning of the project by the client and the team. With this being said, throughout this report we have shown the technologies, tools, and methodologies involved in making the automated synthetic data generation pipeline possible. We have also discussed in detail various parts of the pipeline and expressed the core responsibility of the pipeline parts. With this report we hope to have enlightened readers on the possibilities of such a system and have created a basis that can be further be built upon. In chapter 3, the must-haves, should-haves, could-haves, won't-haves, and non-functional requirements are listed for this software project. The team has successfully implemented the must-haves, should-haves, and could-haves according to all the non-functional requirements, meaning that all requirements in the MoSCoW document are satisfied. The client has expressed his appreciation with this result and has since already started using SWaG in his own pipeline to train his computer vision model.

14.1. Image Quality Rubric Filled In

As discussed in chapter 4 the image quality rubric is a tool used for evaluation. The developers graded a collection of synthetically produced images by SWaG and averaged the results (8.7/10) as shown in graded rubric below.

	Image Quality Rubric				Score
Evaluation characteristic	**Excellent (8-10)**	**Good (7-8)**	**Average (6)**	**Low Quality (1-5)**	
Proportion	The objects are well proportioned to one another	The objects are proportionate to one another	The objects are somewhat proportionate to one another	The objects are not proportionate	8
Background	The objects are placed on a realistic looking background, and there are 10+ backgrounds	The objects are placed on a decent looking background and there are enough (5) backgrounds	The objects are placed on a background, but there are not a lot of them, and they are not very realistic	The background does not fit the scene and there are not a lot of backgrounds	10
Realism	The image has GTA V level rendering	The image is somewhat realistic	The image is sufficiently realistic	The image looks like a cartoon	8
Textures	The objects have their associated textu	The objects have their associated textures fitted	The objects have sufficient textures	The textures are not applied to the objects correctly	8
Resolution	The image quality is high: 1200 x 800	The image quality is decent: 800 x 600	The image quality is sufficient	The image quality is ins	10
Results matching expectations	Image has matching charecteristics to what was specified in the user request	The image has some characteristics missing	Only some of the characteristics are matching	The resulting image does not match the parameters set	9
Lighting	Lighting is similar to what is used at waste facility. For example bright lights are used with minimal shadows.	The lighting results in some shadows that are not realistic	The lighting results in many shadows that are not realistic	The lighting is too dark	9
Camera positioning and aspect ratio	The camera is positioned in a location that is similar to what is used at the waste facility. The camera is directly above the conveyer belt at a reasonable height	The camera postioning is suitable. It could be further away/closer to the conveyer belt to look a bit more realistic	The camera position should be improved by moving it to a new location	The camera is way too close/far away from the objects	8
Object crushing	Objects look crushed in a lifelike manner	Objects look crushed. The crushing is not very realistic i.e(Completely crumpled can that has had too much force applied to it)	Objects are crushed way too much or not at all and the result does not look realistic at all	The objects are not crushed	8
Object overlap	Objects overlap in a similar way to reality	Objects overlap is mostly realistic. No objects are breaking the laws of physics i.e going through one another	Objects overlap mostly realistically. Some objects do not overlap well	The objects do not overlap in a realistic way at all	8
Masking	The image masking is suitable for insta	Image masking is suitable for instance segmentation	Image masking is not really suitable for instance segmentation	The image is not suitable for instance segmentation	10
				Final Score	96
				Grade	8.7272727

Figure 14.1: Image quality rubric filled in

14.2. Future Work

There are many further synthetic data generation developments that can be researched. After having discussions with the client, In this section the developers list future possible features that they believe are the most impactful.

14.2.1. Animations

A variation of generating synthetic images would be to create an animation so that a classification model could be trained on a video. This idea could be further researched by developing a moving conveyor belt and then placing objects on it. Animations would be particularly useful for real world applications as a camera has a frame per second rate that needs training.

14.2.2. Train a Generative Adversarial Network

A Generative Adversarial Network requires a large amount of data to be trained (of the order of thousands). Therefore, once a synthetic dataset of tens of thousands of images has been developed it will be possible to start training a Generative Adversarial Network for real world applications. Developing a Generative Adversarial Network will enable even more data being generated as it shall combine both real and synthetic images as discussed in section 2.1.

Figure 14.2: Example of conveyor belt with dirt and scraps. Image courtesy of our client

14.2.3. Paper Scraps and Dirt (Domain Adaptation)

By adding dirt and things like paper scraps to the scene it is possible to make the images look even more photo-realistic. This will enable higher quality data to train a Computer Vision model to ensure that it performs well once it has been deployed in the real world at the recycling plant.

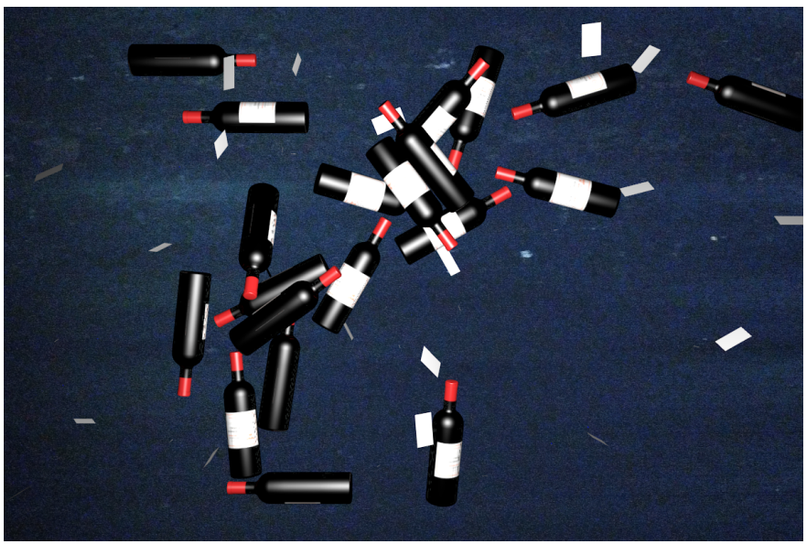


Figure 14.3: Example of wine bottle with paper scraps.

14.2.4. Depth

Generating information about depth will further enable the training of a robotic arm to pick up particular waste for recycling. Research [40] has shown that it is possible to use depth images for training the grasp of a robot.

14.2.5. Translucency and Transparency

A visual feature that will improve the system is the ability to have transparent and translucent objects, see figure 14.4. Some objects that could appear in waste are of plastic and/or glass. This would mean that underlying objects would be partly visible and also light would be refracted in a different manner allowing for even more realistic images to be rendered. Implementation wise there are a few limitations. One of them is the fact that object materials can not be exported/imported, at least not in Blender. Object material is the information that makes the object look like plastic or glass, i.e. transparent or translucent. One way to get around this problem is to work with .blend files rather than .obj files. Furthermore, from the research within the scope of this project it looks like these types of materials are only rendered with the Cycles rendering engine. In fact, the rendered image breaks if these materials are rendered with the Blender Render engine. This is likely due to the Shader, which is used to create these kinds of materials, being only available with Cycles. That these materials are only rendered correctly with Cycles consequently makes the system slower as Cycles is slower than Blender Render.

Figure 14.4: Example of transparent plastic

A

Appendices

A.1. Info Sheet

Project Information Sheet TI3806 Bachelor End Project

SWaG: Synthetic Waste Generator

This software solves the challenge of quickly finding suitable data to train a machine learning model. It provides an easy-to-use API to generate synthetic data. After conducting multiple research experiments, the developers have learned how to rapidly provide custom images to the user. In order to ensure high-quality data, the developers have used the state-of-the-art programming conventions, automated testing techniques, and developed multiple acceptance criteria for the images in the form of an Image Quality Rubric. Furthermore, having followed the Agile development philosophy the team has been able to develop a product specifically tailored to our client's needs. In the future, this software product can be developed even further by using its output to train a General Adversarial Network or by adding depth information to train a robotic arm for example.

Team Roles:

Khalid El Haji, *Contributions:* Setup development environment and CI/CD with Sonarcloud integration. Developed random texturing systems, assisted with background images as plane in scene implementation, and scene preloading optimization. Refactored key parts of code. Developed object scattering system that uses physical simulations to realistically scatter objects on a plane. Developed a crushing system that is able to take objects and generate a realistic crushing of the objects in numerous ways.

Hakan Ilbaş, *contributions:* Selected 3D models and background images. Created input parser. Add the ability to choose the material of the generated object, vary the number of items in a scene, select specific models to render, change lighting conditions. Various optimizations to speed up rendering. Work on texturing, changing backgrounds and crushing of objects. Refactor the majority of the code to lower Cyclomatic Complexity and LOC for functions and write documentation.

Sergen Karpuz, *Contributions*: Selection of 3D models and background images, Research masking in Blender, 'Mask generation and coloring' algorithm, Research Blender camera frustum workings, Variable camera height, Variable size surrounding box, Variable placement box, Research into transparency + translucency options.

Noah Posner, *Contributions*: Parameter customization, Image Quality Rubric, User & Developer Guide, Static Analysis.

Victor Wernet, *Contributions*: Initial working prototype architecture for the synthetic data generation pipeline. Containerization using Docker. Auto-texturing technique. Tests for image segmentation, the dataset manager, and category classes. User guide. Better feedback to the user when an error occurred. Auto-generate categories. Server for handling requests. Nvidia GPU ready container. Algorithm to segment objects in an image, functions to generate COCO datasets, and verified the COCO datasets using Jupyter Notebook. Dataset Manager that handles multiple requests so that COCO Datasets can be created and extended without data-loss.

All team members contributed to software testing, code reviews, preparing the reports and the final presentation.

Supervisor

Associate Professor L. Y. Chen: EEMCS, Distributed Systems

Client

Peter Hedley

Contact Details

V.G.A.Wernet@student.tudelft.nl, H.Ilbas@student.tudelft.nl, S.Karpuz@student.tudelft.nl, K.ElHaji@student.tudelft.nl, N.Posner@tudelft.nl

The final report for this project can be found at: http://repository.tudelft.nl

A.2. User Guide

SWaG - a Synthetic Waste Generation pipeline used for object detection

This repo contains a synthetic data generation pipeline that can be run locally, in a container or on the cloud. For more information on Object detection using COCO have a look here.

This software guide contains instructions on how to install and use the software. Although there are other ways to run the software. We advise for ease of use to install Docker in order to run the code and Postman or cURL to send requests for images in this initial release.

If you have any questions please contact the team:
Victor Wernet: v.g.a.wernet@student.tudelft.nl
Sergen Karpuz: s.karpuz@student.tudelft.nl
Noah Posner: n.b.posner@student.tudelft.nl
Khalid El Haji: k.elhaji@student.tudelft.nl
Hakan Ilbas: h.ilbas@student.tudelft.nl

Setting up Nvidia drivers/CUDA toolkit

The appropiate Nvidia GPU drivers and toolkit needs to be installed. For more information have a look here.

Setting up Docker

Users will need to download and install the Docker Engine. Visit the following link to install the appropriate Docker engine for your operating system.

Steps to build the image and run the container

Before everything, make sure your terminal is pointing to `/recycleye$` . Where the `Dockerfile` is located. We can now proceed and run the following steps.

Run the following command `docker build -t recycleye .` to build the image (note the . is part of the command). You can check and see if the images have been built by using the following command `docker images` , if all is correct you should see the following two images **recycleye** and **nvidia/cudagl**.

Now to run the container, execute the following command assuming you have GPU on your device and you updated the `/your/path/` to a file path on your system: `docker run -v /your/path/:/usr/src/recycleye/output/ --gpus all -p 5000:5000 -d -t --name recycleye.container recycleye` If your system does not have any GPU remove `--gpus all` from the above command.

This should now run the container, to see if it is running execute the following command `docker container ls` , you should now see your container present in the list.

> Note: You might have to run the docker commands using sudo and make sure your path has a / at the end of it. Otherwise the generated COCO dataset will not be visible.

Preparing the request

Example showing how one can use the JSON.

```
{
  "seed": 3,
  "material_types": ["aluminium", "cardboard", "glass"],
  "material_composition": [0.2, 0.4, 0.4],
  "crowding": 10,
  "rendering_engine": "CYCLES",
  "enable_gpu": "true",
  "background": "backgrounds/b2.JPG",
  "resolution": [800, 400],
  "camera_height": 2,
  "surrbox_size": 1,
  "surrbox_hide": "true",
  "light_location": [0, 0, 5],
  "texturing": {
      "random_texture_coords_type": "UV",
      "enable_random_texture": true
  },
  "light_strength": 2,
  "version": 1.0,
  "dataset_name": "Your_dataset_name"
}
```

In the above example the GPU is enabled. Cycles rendering is more accurate but is slower than blender render. To enable faster rendering, you can simply remove the `enable_gpu` and `engine` parameters. For more information about the parameters have a look here.

> Note: The naming convention for the categories is simply the name of a folder inside of models and afterwards the folder object name. For example if the path

> is this: models/glass/coca_bottle/. The category name needs to be called glass_coca_bottle as seen in the above JSON example.

Routes

| Route | Purpose | Result | | ----- | ----- | ----- | `/` | The main route for rendering images and creating coco datasets. | A dataset containing images and a single JSON file containing information about the dataset and the segmentations and bbox's. |

> The first initial call can be slower than the subsequent calls. This is due to the scene.blend file is being created. This allows a faster loading time for subsequent calls. It is important to remove the scene.blend if new models have been added to the models folder. One can simply delete the scene.blend file and a new updated one will be created.

Using cURL

By default windows does not come with cURL, but this can be downloadedhere.

```
curl --location --request GET '0.0.0.0:5000' --header 'Content-Type: application/JSON' --data-raw 'YOUR_JSON'
```

> Note: Both GET and POST are valid for the API.

Using Postman

Download and install Postman. Afterwards simply create a new request, and add 0.0.0.0:5000 in the url bar. Afterwards click on `body`, click on the button with the title `None` and select `raw`, afterwards instead of `Text` click on `JSON`. Press `Send`.

> All requests either using cURL or Postman should be sent to the host address 0.0.0.0 on port 5000 or host address 127.0.0.1 on port 5000 (Windows).

Can I hook up a front-end technology on the pipeline?

The communication protocol is HTTP(s). Meaning any front-end technology can be connected to the pipeline.

Parameters and their meaning

These are the current valid parameters.

Parameter	Values	Description
seed	A single value in the range of `[0, 2147483647]`	An `int` value that specifies how objects on the scene should be placed/dropped. A seed can be used to produce an identical result.
material_types	`"aluminium"`, `"cardboard"`, `"glass"`, `"hdpe"` or `"pet"`	A `list of strings` containing what mix of types the image should contain.
material_composition	Values in the range of `[0, 1]` in such a way the list must sum up to 1.0	A `list of floats` (or int for a single type) representing the proportions of how much of a given item we want and is based off of the `crowding` parameter.
crowding	A single value in the range of `[0, 500]`	An `int` value specifying the total amount of items an image should contain. This is not type dependent, but rather the sum of all objects.
rendering_engine	`"CYCLES"` or `"BLENDER_RENDER"`	A `string` value representing the render engine. CYCLES is slower but more accurate and can be run on CPU or GPU. BLENDER_RENDER is faster but less accurate and can only be run on CPU.
file_name	A single value specifying the name of a specific object	A `string` value that when specified, no material_types or material_composition parameter need to be specified.
enable_gpu	A single value being either `false` or `true`. Can either be specified as a string or a boolean in the JSON request	GPU rendering only works for cycles. Also Enabling the GPU and not having one will throw an error.
background	A single value for the conveyor belt with following format `backgrounds/<your_file>.png`	A `string` value selecting the background image to be used as the conveyor belt.
resolution	A single value representing the resolution of the image with the following format `widthxheight`	A `string` value for specifying the resolution of the output image(s). A bigger the resolution means a slower output.

Parameter	Values	Description
camera_height	An integer value in the range of `[1, 3]` representing the camera height. The default value is `1`	An `integer` value for specifying the camera height, with `1` being the closest to the scene, `2` a bit further away and `3` being the furthest away from the scene.
surrbox_size	An integer value in the range of `[1, 3]` representing the size of the surrounding box which defines the field of placement for the objects. The default value is `1`	An `integer` value for specifying the size of the surrouding box, with `1` being the smallest, `2` a bit larger and `3` being the largest.
surrbox_hide	A boolean value either `true` or `false` that defines whether to show the surrounding box or not. The default value is `true`	`true` hides the surrounding box in the rendered image, `false` makes the surrounding box visible in the rendered image.
dataset_name	A single value specifying the dataset name	A `string` value that uniquely defines the dataset. A dataset contains 1 single COCO JSON file and the images belonging to it.
version	A single `float` value	This specifies the current version of the dataset. If not given the dataset will receive a default value of 1.0
light_strength	A single value specifying the light_strength of the light that illuminates the scene	A `int` value that must be positive, that defines the light_strength. A higher light_strength, means the light will shine brighter. The default value is 10.
light_location	A list specifying the location to place the light, where the list represents the x, y and z coordinates (which must all be positive)	A `list` value that defines the location of the light. With this parameter you can simulate different lighting conditions by letting the light shine from different locations. The default value is `[0, 0, 10]` . The list must have a length of 3, and all values of it must be positive.
texturing	A object containing two attributes random_texture_coords_type and enable_random_texture	Texturing object specifies the textures settings that will be used in scene. The texturing object has two attributes: random_texture_coords_type and enable_random_texture. The first attribute can be set to 'UV' or 'GLOBAL' to indicate texture coords type. The second attribute can be set to `true` or `false` enabale_random_textures for object that have no default texturing.

Tech

The synthetic data generation pipeline uses a number of software to work properly: * Docker - Containerize applications * Python3.6x - Python Programming Language. * Flask - Flask is a lightweight WSGI web application framework. Fast and easy to extend. * Blender - Blender Engine for rendering. * COCO API - COCO is a large-scale object detection, segmentation, and captioning dataset. * Nvidia - Nvidia drivers for the GPU. * CUDA - Cuda Toolkit. * Ubuntu 18.04 - Base image used for CUDA & OpenGL

doc v0.1.1

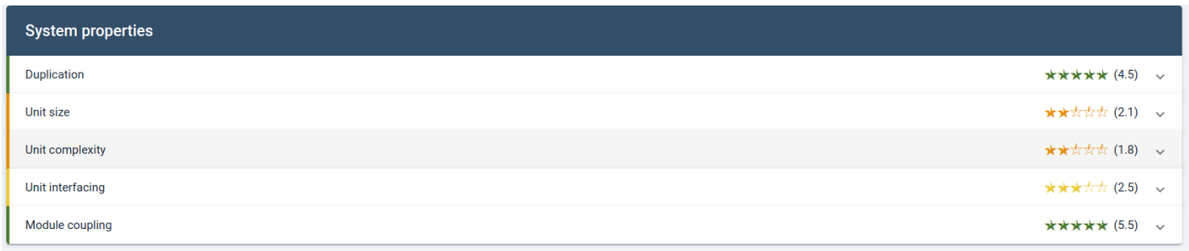
Not relevant after this point, but will be included later.

Steps to run the pipeline without using a container (will be organized soon)

Instead of object name we use the typename_folderobjectname for the name in the categories Make sure you have python installed (version 3.6+) Make sure you also have installed flask using pip `export BLENDER_PATH=/your/blender/location export FLASK_APP=/YOURPATH/recycleye/app/server.py export PYTHONPATH=/YOURPATH/recycleye/app/FOLDERNAME` and to add multiple you can always use `:` to append more folders to the `sys.path` (e.g. export PYTHONPATH=/path1:path2:path3). make sure blender 2.8x is installed and path has been added to the server.py install requirements.txt Open up a terminal and make sure your terminal is pointing to `recycleye$` . Run the following command `flask run -h 0.0.0.0 -p 5000` .

A.3. Project Plan and Timeline

A.4. Software Improvement Group



A.4.1. McCabe Complexity

Name	Lines of code	McCabe complexity
render.py	83	15
json_parser.py:set_parameters(json_request,rootdir)	46	14
json_parser.py:material_types_and_composition_validator(material_types,mat	27	10
json_parser.py:parse(json_request,mt,rootdir)	26	8

A.5. Project Description

A.5.1. The Client

The client is a computer vision company in the recycling/waste industry. The industry at present uses a large amount of manual labour to sort waste, often in poor conditions, high staff turnover and expensive operationally for a waste plant. The client is aiming to replicate human vision through algorithms and replace human hands with robotics. The main problem is data, not only must the model detect different types of waste but also in numerous crushed permutations. The project specification aims to alleviate this by generating unlimited synthetic data for training a detection model to classify waste.

A.5.2. Project Spec

Given a photo of a collection of products, synthetically create numerous crushed permutations of these objects on a conveyor. Train a detection model on the synthetic data and optimise for the best detection model performance for detection.

A.5.3. Extensions

1. Use a model to take photos and render a 3D CAD model which can then be crushed an unlimited number of ways via physical deformations (proposed approach).
2. Alter generated synthetic data to improve detection model's ability to detect real data.
3. Alter generated synthetic data to improve detection model's ability to detect real data.

A.5.4. Technologies

1. Tensorflow / deep learning (GANS?)
2. Unreal Engine or some kind of physics engine

A.5.5. Other Information

The client is based out of Microsoft Startups in London, we have allocated a travel budget to ensure we can meet the project needs. If students wish we can provide office space in London, or we will visit on a regular basis to ensure project is delivered and resources available. We are formed from machine learning and computer vision engineers so we know how to frame a good project and have data ready to do so. NDA perspective - we don't need one, but we would like access to the code post-project and full rights to use it in a profit-making scenario and as a part of the clients technology stack.

Bibliography

[1] Introduction. URL http://builder.openhmd.net/blender-hmd-viewport-temp/render/blender_render/introduction.html.

[2] Introduction. URL http://builder.openhmd.net/blender-hmd-viewport-temp/render/cycles/introduction.html.

[3] Cython. URL http://people.duke.edu/~ccc14/sta-663-2018/notebooks/S13B_Cython.html.

[4] Gpu rendering. URL https://docs.blender.org/manual/en/2.79/render/cycles/gpu_rendering.html.

[5] A. R. Abdul Rajak, S. Hasan, and B. Mahmood. Automatic waste detection by deep learning and disposal system design. *Journal of Environmental Engineering and Science*, 15(1):38–44, 2019. URL www.scopus.com.

[6] Michel Anders. *Blender 2.49 Scripting: Extend the Power and Flexibility of Blender with the Help of Python, a High-level, Easy-to-learn Scripting Language*. Packt Publishing Ltd, 2010.

[7] Anton Kovalyov. Jshint. URL https://jshint.com.

[8] Apache Software Foundation. Maven. URL https://maven.apache.org/.

[9] Blender Foundation. Blender. URL https://www.blender.org/.

[10] Brian Okken. Pytest. URL https://docs.pytest.org/en/latest/.

[11] MinSu Chae, HwaMin Lee, and Kiyeol Lee. A performance comparison of linux containers and virtual machines using docker and kvm. *Cluster Computing*, 22(1):1765–1775, 2019.

[12] Yulia S. Chernyshova, Alexander V. Gayer, and Alexander V. Sheshkus. Generation method of synthetic training data for mobile OCR system. In Antanas Verikas, Petia Radeva, Dmitry Nikolaev, and Jianhong Zhou, editors, *Tenth International Conference on Machine Vision (ICMV 2017)*, volume 10696, pages 640 – 646. International Society for Optics and Photonics, SPIE, 2018. doi: 10.1117/12.2310119. URL https://doi.org/10.1117/12.2310119.

[13] Daniel Marjamäki. Cppcheck. URL http://cppcheck.sourceforge.net/.

[14] Unity Documentation. Unity user manual, 2019. URL https://docs.unity3d.com/Manual/FrustumSizeAtDistance.html.

[15] eCola. @eColaSftDrink. Relax this weekend with an ice-cold ecola!, Feb 2018. URL https://twitter.com/eColaSftDrink/status/962417873716039680.

[16] Epic Games. Unreal engine. URL https://www.unrealengine.com/.

[17] Facebook Open Source. Jest. URL https://jestjs.io/.

[18] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[19] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[20] Hans Dockter, Adam Murdoch, Szczepan Faber, Peter Niederwieser, Luke Daley, Rene Gröschke, Daz DeBoer. Gradle. URL https://gradle.org/.

[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[22] Javier Luis Cánovas Izquierdo and Jordi Cabot. Discovering implicit schemas in json data. In *International Conference on Web Engineering*, pages 68–83. Springer, 2013.

[23] Jim Hugunin. Numpy. URL https://numpy.org/.

[24] JogAmp Community. Jogl. URL https://jogamp.org/jogl/www/.

[25] Kent Beck, Erich Gamma, David Saff, Kris Vasudevan. Junit. URL https://junit.org/.

[26] Khronos Group. Opengl. URL https://www.opengl.org/.

[27] Kirill Klenov. Pylama. URL https://github.com/klen/pylama.

[28] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[29] TY Lin and P Dollar. Ms coco api, 2016.

[30] Mantra Malhotra. Unreal engine vs unity 3d games development: What to choose?, Dec 2019. URL https://www.valuecoders.com/blog/technology-and-apps/unreal-engine-vs-unity-3d-games-development/.

[31] Nvidia, Documentation. Nvidia docker opengl, 2017. URL https://docs.nvidia.com/ngc/ngc-user-guide/graphics/software_stack_zoom.png. [Online].

[32] Oliver Burn. Checkstyle. URL https://checkstyle.sourceforge.io/.

[33] OpenJS Foundation. Electron. URL https://www.electronjs.org/.

[34] Opensource Software. Pyhint. URL https://github.com/codeq/pyhint.

[35] Andrew Price. 4 easy ways to speed up cycles, Nov 2012. URL https://www.blenderguru.com/articles/4-easy-ways-to-speed-up-cycles.

[36] Bernadette M Randles, Irene V Pasquetto, Milena S Golshan, and Christine L Borgman. Using the jupyter notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–2. IEEE, 2017.

[37] Kenneth Reitz. URL https://docs.python-guide.org/writing/tests/.

[38] Rene Rivera. Boost. URL https://www.boost.org/.

[39] Robert Bradshaw, Stefan Behnel. Cython. URL https://cython.org/.

[40] Daniel Seita, Nawid Jamali, Michael Laskey, Ron Berenstein, Ajay Kumar Tanwani, Prakash Baskaran, Soshi Iba, John Canny, and Ken Goldberg. Robot bed-making: Deep transfer learning using depth sensing of deformable fabric. *arXiv preprint arXiv:1809.09810*, 26, 2018.

[41] SideFX. Houdini. URL https://www.sidefx.com/products/houdini/.

[42] SonarSource. Sonarcloud. URL https://sonarcloud.io/.

[43] Szczepan Faber, Brice Dutheil, Rafael Winterhalter, Tim van der Lippe. Mockito. URL https://site.mockito.org/.

[44] Alexander van Tol. Python code quality: Tools amp; best practices, Aug 2018. URL https://realpython.com/python-code-quality/.

[45] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. *arXiv preprint arXiv:1809.10790*, 2018.

[46] Unity Technologies. Unity. URL https://unity.com/.

[47] Vojta Jina. Karma. URL https://karma-runner.github.io/latest/index.html.

[48] Andy G Ye and David M Lewis. Procedural texture mapping on fpgas. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 112–120, 1999.