

Bachelor Project



**Czech
Technical
University
in Prague**

F3

**Faculty of Electrical Engineering
Department of Cybernetics**

Generating of the Training Data for 2D Segmentation

Jan Ferbr

**Supervisor: Ing. Michal Polic
Field of study: Cybernetics and Robotics
May 2022**

I. Personal and study details

Student's name: **Ferbr Jan**

Personal ID number: **483695**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Generating of the Training Data for 2D Segmentation

Bachelor's thesis title in Czech:

Generování trénovacích dat pro 2D segmentaci

Guidelines:

1. Read the related literature and get familiar with the used reconstruction pipeline COLMAP.
2. Create a custom dataset of factory environment for training 2D segmentation, i.e., record the laboratory B-670 by RGB and RGB-D cameras.
3. Create the 3D reconstruction from the captured data using COLMAP reconstruction software, i.e., utilize both the RGB and RGB-D images to create the 3D model.
4. Run the SPSG to fill the holes in the 3D model.
5. Segment manually (or automatically) the dense pointcloud to objects which will be used for training the 2D segmentation network.
6. Generate training data, i.e., renders of the 3D model and segmentation masks, by AI Habitat.
7. Train and evaluate the error of 2D segmentation by YOLACT with and without using SPSG for tuning the dense reconstruction.
8. BONUS - retrain/finetune the SPSG model on captured data.

Bibliography / sources:

- [1] Savva, Manolis, et al. "Habitat: A platform for embodied ai research." ICCV2019
- [2] Dai, Angela, et al. "Spsg: Self-supervised photometric scene generation from rgb-d scans." CVPR2021
- [3] Schonberger, Johannes L., and Jan-Michael Frahm. "Structure-from-motion revisited." CVPR2016

Name and workplace of bachelor's thesis supervisor:

Ing. Michal Polic Applied Algebra and Geometry CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **21.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Michal Polic
Supervisor's signature

prof. Ing. Tomáš Svoboda, Ph.D.
Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Acknowledgements

I would like to sincerely thank my supervisor Ing. Michal Polic for all the advices and the time he was willing to offer me. Also I would like to thank my family, my friends and my girlfriend for moral support and never ending motivation during the writing process.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

V Praze, 20. May 2022

Abstract

2D segmentation is nowadays used for multiple applications, ranging from medical imaging to object recognition. 2D segmentation requires tens of thousands of annotated images. This paper aims to describe the process of creation of those training images from a 3D model of real environment.

Instance masks are created from a 3D model that was scanned using Hololens 2 and Azure Kinect. The sparse reconstruction was obtained using COLMAP and the dense reconstruction using Azure Kinect SDK and Hololens 2. Afterward, the dense reconstruction was improved by SPSG algorithm to fill in the holes and improve the overall quality of the mesh. The masks were rendered from dense reconstruction by AI habitat. Using those masks a dataset in COCO format was generated and employed for training using YOLACT. The results show that training instance segmentation using this pipeline is possible, however the hypothesis predicting that results will improve while using SPSG, did not prove to be right.

Keywords: SPSG, Azure kinect, Hololens 2, AI Habitat, COLMAP, YOLACT, instance segmentation, optimization of dense reconstruction

Supervisor: Ing. Michal Polic
Contact: michal.polic@cvut.cz,
ferbrjan@fel.cvut.cz

Abstrakt

V dnešní době je 2D segmentace aplikována v mnoha různých částech výzkumu. Tyto aplikace zahrnují například zobrazování medikálních obrazů či rozpoznávání objektů. 2D segmentace ale vyžaduje desetitisíce anotovaných snímků. Tato práce se zaměřuje na popis procesu generování těchto trénovacích snímků z 3D modelu reálného prostředí.

Masky instancí jsou vytvářeny za pomoci 3D modelu, který byl naskenován pomocí Hololens 2 a Azure Kinect. Řídká rekonstrukce byla vytvořena za pomoci softwaru COLMAP a hustá rekonstrukce byla vytvořena za pomoci Azure Kinect SDK a softwaru od Hololens 2. Poté byl 3D model z husté rekonstrukce zdokonalen za pomoci SPSG, s cílem zaplnit díry a zvýšit celkovou kvalitu a přesnost modelu. Z tohoto modelu v podobě husté rekonstrukce byly poté vytvořeny masky, které následně byly použity v datasetu na učení za pomoci projektu YOLACT. Výsledky ukazují, že učení segmentace instancí za pomoci této pipeline je možné, avšak hypotéza předpovídající zlepšení přesnosti při použití SPSG se nepotvrdila.

Klíčová slova: SPSG, Azure kinect, Hololens 2, AI Habitat, COLMAP, YOLACT, segmentace instancí, optimalizace husté rekonstrukce

Překlad názvu: Generování trénovacích dat pro 2D segmentaci

Contents

Project Specification	iii
1 Introduction	1
2 Key concepts	3
2.1 Scanning	3
2.1.1 Azure Kinect	3
2.1.2 Hololens 2	4
2.1.3 Samsung Galaxy S10e	5
2.2 Sparse and dense reconstruction	5
2.2.1 COLMAP	6
2.2.2 Hololens 2	7
2.3 SPSCG	8
2.4 Semantic segmentation, instance segmentation and instance masks	9
2.4.1 AI Habitat	10
2.5 Training	10
2.5.1 Yolact	10
2.6 Convolutional neural networks	11

3 Structure of the thesis summarized using a diagram	13
4 Data collection	17
4.1 B-635 laboratory	17
4.2 Recording devices	18
4.2.1 Azure Kinect	18
4.2.2 Hololens 2	20
4.2.3 Samsung galaxy S10e	22
5 Data processing	23
5.1 Azure Kinect	23
5.1.1 Image processing	23
5.1.2 Depth image alignment	24
5.1.3 Point cloud/mesh	26
5.2 Hololens 2	27
5.2.1 Image processing	27
5.2.2 Point cloud/mesh	27
5.3 COLMAP	30
5.3.1 COLMAP input	30

5.3.2 Sparse reconstruction	30	6.1.2 YOLACT evaluation	59
5.3.3 COLMAP output	32	7 Result analysis	63
5.4 SPSG	33	7.1 Scanning process	63
5.4.1 Installation	33	7.2 COLMAP	64
5.4.2 Testing on data provided by the authors	34	7.3 SPSG	64
5.4.3 Structure of SPSG	34	7.4 YOLACT	67
5.4.4 Datagen	36	7.5 Comparison of non-SPSG pipeline vs. SPSG pipeline	69
5.4.5 Point cloud alignment	39	7.6 Possible improvements	69
5.4.6 Point cloud rotation	42	7.6.1 More scanned data	70
5.4.7 Point cloud to mesh	45	7.6.2 SPSG trained model	70
5.4.8 .sens file creation	45	7.6.3 Number of YOLACT iterations	70
5.4.9 SPSG	47	7.6.4 Other improvements	71
6 Segmentation and training	51	7.7 Conclusion	72
6.0.1 Meshlab segmentation of 3D model	51	8 Code	73
6.0.2 AI Habitat	53	A Bibliography	75
6.1 YOLACT	57		
6.1.1 Training YOLACT	57		

Figures

1.1 Enhancing 3D model using SPSPG [6].....	2	4.3 Azure Kinect - fields of view [20].	19
2.1 Azure Kinect [10].	4	5.1 visualization of k4a transformation depth image to color camera() [25].	25
2.2 Hololens 2 [12].	5	5.2 RGB image and its corresponding point cloud created by Azure Kinect SDK.....	26
2.3 The structure of SfM pipeline [4].	7	5.3 Single point cloud generated by save_pclouds.py.	28
2.4 Hololens 2 pinhole projection RGB frame.	7	5.4 Fused point cloud generated using pymeshlab library.	29
2.5 SPSPG network architecture [6]. ...	9	5.5 Mesh generated using TSDF integration.	29
2.6 Difference semantic segmentation and instance segmentation [18]	9	5.6 Data processing and SPSPG.	35
2.7 YOLACT architecture with $k = 4$ [8]	11	5.7 Additional include directories. ...	37
2.8 Diagram of neural network.	12	5.8 Additional Dependencies.	37
3.1 Project structure diagram part 1.	14	5.9 representation of α, β, γ in geogebra 3D [32].	40
3.2 Project structure diagram part 2.	15	5.10 representation of α and w in geogebra 3D [32].	43
4.1 View of B-635 laboratory - RGB format.	18	5.11 SPSPG reconstruction front view.	49
4.2 View of B-635 laboratory - depth format.	18	5.12 SPSPG reconstruction back view.	49
		5.13 SPSPG reconstruction side view.	49

6.1 Segmented empty map of non-SPSG model.	52
6.2 Segmented object examples.	52
6.3 RGB frame with itscorresponding semantic frame.....	54
6.4 COCO annotations on images from dataset without SPSG.	56
6.5 COCO annotations on images from dataset with SPSG.	57
6.6 YOLACT eval.py results on non SPSG dataset and non SPSG images.	60
6.7 YOLACT eval.py results on SPSG dataset and SPSG images.	60
6.8 YOLACT eval.py results on non SPSG dataset and real images. ...	61
6.9 YOLACT eval.py results on SPSG dataset and real images.	61
7.1 Results presented by SPSG. [6] .	66
7.2 Our results from the same room.	66
7.3 Our results from the same room but different position.	67
7.4 mAP vs. number of iterations. .	71

Tables

4.1 Number of images and length of recordings of Azure Kinect.	20
5.1 zParametersScanMP.txt argumets.	38
7.1 mAP results non-SPSG	68
7.2 mAP results SPSG	68



Chapter 1

Introduction

The instance segmentation of 2D images is a widely researched topic in the computer vision community with many practical applications. These applications can range from object detection and classification in robotics [1] to segmentation of medical images [2]. One of the obstacles to overcome, is the amount of data required for training. These data are in the form of images with annotations and the creation of this data requires a lot of resources (gathering training images, time to create annotations). In these days, there exist a lot of frameworks for 3D model creation (e.g., COLMAP [3], PatchmatchNet [4]). The annotation of 3D mesh is much faster and more efficient than the annotation of thousands of images for each object class. However, the biggest challenge of this approach is creating a sufficiently detailed 3D model, that can be used for generating realistically looking images and instance segmentation masks. Unfortunately, the models generated by the state-of-the-art commercial software (Capturing Reality [5]) are not detailed enough to provide reasonable training data for object detection in complex scenes.

This work aims to enhance the 3D model by employing SPSG [6], which can complete the holes in the mesh obtained from a dense point cloud. The result should be a more accurate model, leading to more realistically looking training images and masks.

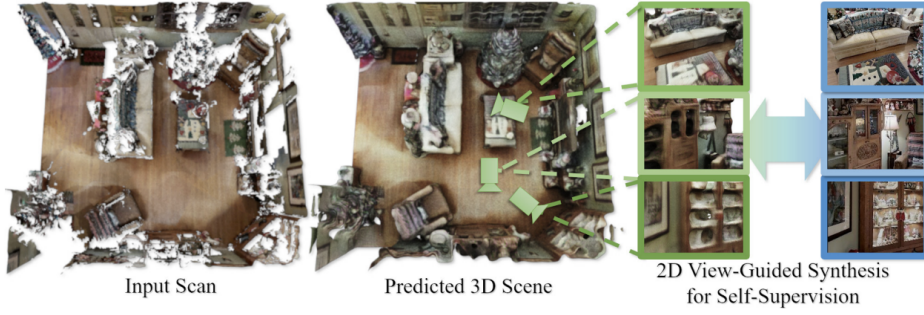


Figure 1.1: Enhancing 3D model using SPSG [6].

This thesis describes the generation of training data, steps to create a reconstruction, improving the 3D model, its segmentation, and lastly, training and the evaluation of the 2D segmentation network. We start with the data collection process, describing each scanning device used. Secondly, the thesis focuses on processing the data to create 3D models. The third step focuses on improving 3D models. The fourth part evaluates the segmentation of the models, annotation of the models and creation of datasets in COCO [7] format for training. The last part describes the training of the data by YOLACT [8] and its evaluation.

Chapter 2

Key concepts

2.1 Scanning

We first focus on scanning a specific environment (e.g., construction site/factory environment). The goal was to obtain a 3D model of some testing area. Multiple scanner devices were used to build a complete and accurate digital copy of the environment usable for rendering.

We start with a brief description of each scanning device. More details are listed in chapter 4.

2.1.1 Azure Kinect

Azure Kinect is a scanner device that contains a depth (time-of-flight (ToF)) camera and RGB camera. The Azure Kinect was chosen as a scanning device because of the ability to record accurate depth data, that are ideal for creating 3D models. Moreover, a software called Azure Kinect SDK [9] provides a function wrapping depth images into RGB images and thus perfectly synchronizes those frames (the usefulness of this function is discussed later). Another helpful feature is the access to camera frame meta-data that enables the selection of frames with given timestamps and getting used camera configurations. The output of Azure Kinect is a .mkv file containing all the

data from a single recording.

More informations regarding hardware and software specifications are in section 4.2.1.

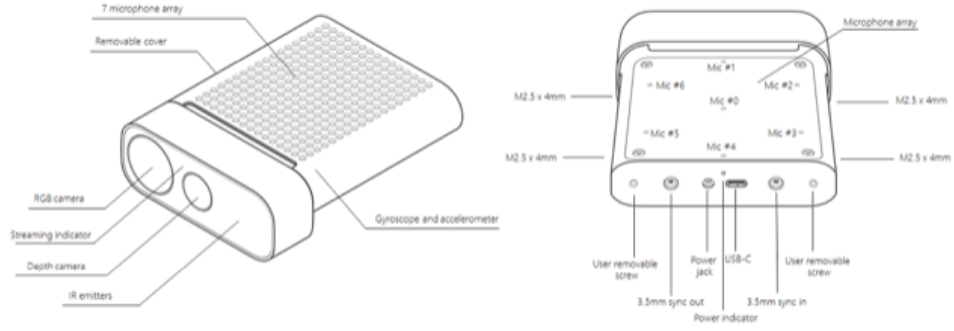


Figure 2.1: Azure Kinect [10].

2.1.2 Hololens 2

Hololens 2 is an augmented reality device equipped with four grayscale cameras, one RGB camera and one ToF sensor. The advantage of this device is the range of different sensors. Another advantage for 3D scanning is a spatial mapping option that allows real-time environment meshing. This feature, that can be found in Hololens 2 Github [11], was employed as the primary method for dense scene reconstruction.

More information regarding hardware and software specifications is in section 4.2.2.

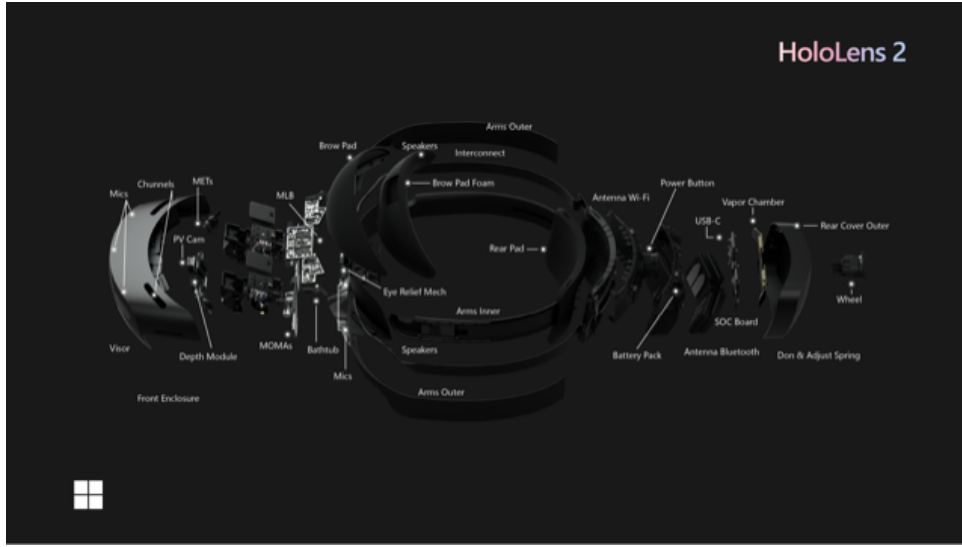


Figure 2.2: Hololens 2 [12].

2.1.3 Samsung Galaxy S10e

In order to stick all the images into a more complete scene, a Samsung Galaxy S10e, featuring a 16-MP ultra-wide camera, was used. The wide-angle of the camera provides multiple frames that capture the same area of the scanned environment, allowing for better connection of frames during SfM. Moreover the high resolution is responsible for better details during the MVS.

More information regarding S10e is in section 4.2.3.

2.2 Sparse and dense reconstruction

The idea of sparse and dense reconstruction is described as taking a set of scanned depth and RGB images and aligning them into a 3D representation of the scene. Sparse reconstruction refers to camera extrinsics, camera intrinsics and sparse pointcloud, while dense reconstruction refers to a complete 3D scan of the environment. (dense point cloud or mesh representation) For our sparse reconstruction, we worked with COLMAP software, and for our point cloud/mesh creation, we used software provided by Hololens 2.

2.2.1 COLMAP

COLMAP is a general-purpose, end-to-end image-based 3D reconstruction pipeline (i.e., Structure-from-Motion (SfM)[4] and Multi-View Stereo (MVS)[3]) with a graphical and command-line interface. We utilized it to estimate camera poses or RGB images from all the recording devices. These poses refer to world to camera transformations, which was one of the main inputs for the SPSPG software described later in the thesis. The main parts of SfM are feature extraction, matching, and mapping.

Feature extraction is a process of finding distinguishable patches in images. These patches are described by the coordinates of their center in the image and the feature vector. The main goal is to eliminate redundant data by transforming the image into a set of different feature vectors. Those feature vectors are called features. The features refer to specific patches, and they should be invariant to scale, rotation and translation. A widely used example of a feature detector and descriptor is SIFT [13].

The next step of SfM is matching of features between pairs of images. At first, the tentative matches are found by selecting the pairs of feature vectors with the smallest distance. The tentative matches realize correspondences between the distinguishable patches in pairs of images. Secondly, the tentative matches are verified by relative pose geometry constraints.

COLMAP offers multiple types of matching: exhaustive matching, sequential matching, vocab-tree matching, spatial matching, or transitive matching. We chose a vocab-tree matching [14] method in this thesis. This method speeds up the process, because each feature vector is matched to a database of already existing feature vectors, which allows clustering of similar feature vectors.

The last step of SfM is mapping. Firstly a pair of initial frames is chosen. Then, using the correspondences of patches, new image is registered and triangulated in order to obtain a relative pose. Using the relative pose, the 2D to 3D correspondences are found and the absolute pose is calculated for given frame. After the absolute pose is obtained, another frame is registered, triangulated and the same process is applied again.

More information regarding COLMAP is in the data processing part at section 5.3.

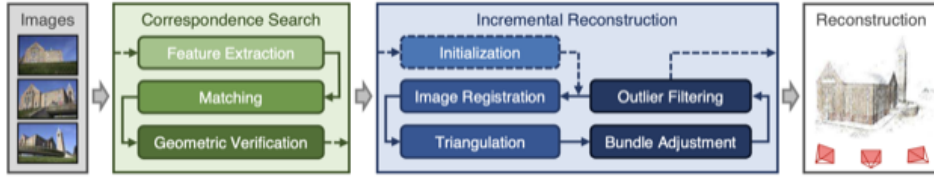


Figure 2.3: The structure of SfM pipeline [4].

2.2.2 Hololens 2

As mentioned in prior, Hololens 2 software [11] also includes an option for real-time environment meshing. Moreover, Hololens 2 track the camera poses and provides extrinsics and intrinsics, as so as the lookup table for converting depth frames into dense point clouds. The extrinsic parameter allows us mapping of all the point clouds into a single coordinate system. After creating such point cloud, it can be easily pruned and meshed using any 3D meshing software. (e.g., Meshlab [15], blender [16])

Hololens 2 preprocessing converts the recorded images and depth maps into a simple pinhole camera model. Such a format can be employed in the Open3D implementation of TSDF. TSDF describes a 3D model representation using voxels, where each voxel represents the distance to the closest surface. The mesh created using this TSDF function was used for our thesis.

More information regarding Hololens 2 software is in section 4.2.2.

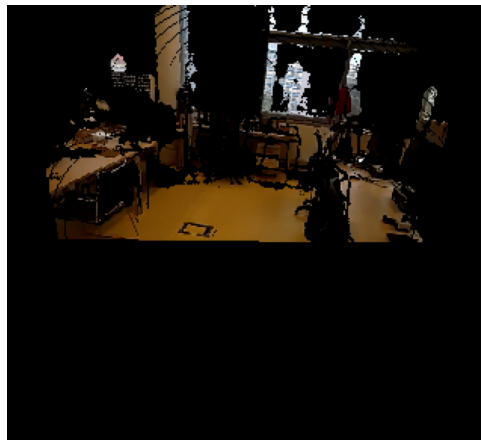


Figure 2.4: Hololens 2 pinhole projection RGB frame.

2.3 SPSG

SPSG [6] is software that tries to complete the missing parts of a 3D model using 2D renderings of the incomplete 3D model of a scene. SPSG uses a self-supervised approach to improve a model by using an incomplete version of the same model and learning to inpaint it back to its original form.

The key idea of the neural network is to formulate an autonomous approach based on 2D view-guided synthesis. Since it is important to learn on realistic data, the program needs to be able to learn from incomplete target scan data, as it is impossible to obtain ground truth for real-world scans. Thus, the learning process is based on the correlation between the incomplete target data and the "more incomplete" version of the same data. In other words, the training process splits input RGB-D images to two subsets. One subset is employed to create TSDF and the second one realizes training outputs for estimation of missing parts.

SPSG formulates 2D rendering losses that guide color and geometry predictions of the output. These losses are obtained by rendering the target RGB-D images in a differentiable fashion, generating color, depth and world-space normal images for a given view. Using these normal images, the adversarial, perceptual and reconstruction losses are obtained. The reconstruction loss is used to fix the geometry and color predictions as can be seen in figure 2.5, as well as the perceptual and adversarial losses are used to provide a more realistic appearance for the final prediction.

The SPSG is fully convolutional and end-to-end trainable. The algorithm first predicts geometry and then colors so that the color predictions can be inspired by the geometric predictions. The data format used for SPSG is mainly .sdf, a TSDF representation of the scene. This data is then standardly processed in chunks of $64 \times 64 \times 128$ voxels, with a voxel size of $2cm^3$.

More information regarding SPSG is in section 5.4.

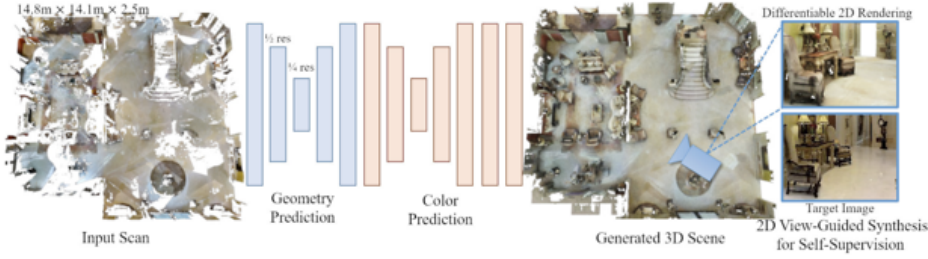


Figure 2.5: SPSG network architecture [6].

2.4 Semantic segmentation, instance segmentation and instance masks

Semantic segmentation is the process of categorizing a scene into sets of objects that belong to the same class. The downside of semantic segmentation is that if there are two objects from the same category in one frame, they are categorized, but they are not distinguishable in the semantic frame.

On the other hand, instance segmentation [17] differs from semantic segmentation by not only categorizing each object into a given class, but also distinguishing between multiple objects in each of those classes. While semantic segmentation only refers to different classes for each pixel, instance segmentation also contains id for each object in a given class.

Instance masks are the product of instance segmentation. Those masks contain information for each object in the frame, including its class, its id and the range of pixels referring to that specific instance.

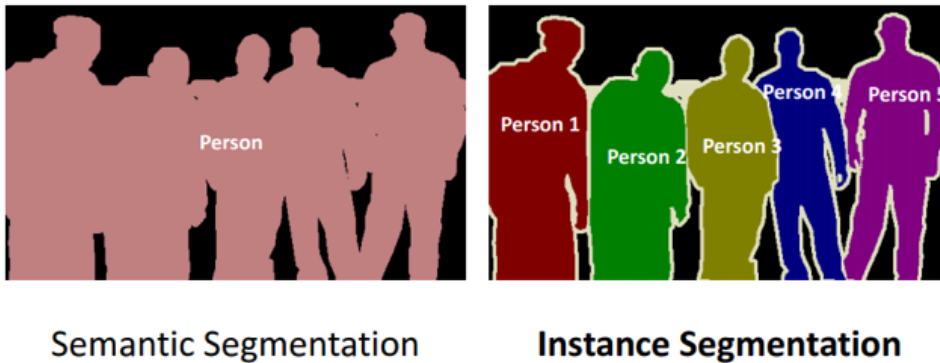


Figure 2.6: Difference semantic segmentation and instance segmentation [18]

■ 2.4.1 AI Habitat

AI Habitat [19] is an agent simulation software from Facebook, that is made for research in embodied AI. It supports 3D scans of various scenes and allows artificial agents to move inside those environments. Moreover, it allows configurations of multiple sensors, including a RGB sensor and a semantic sensor. Using this simulation software, we can easily create a dataset of any size, that will contain RGB images and instance segmentation masks.

More information regarding AI habitat is in section 6.0.2.

■ 2.5 Training

The training part of this thesis focuses on training a model that recognizes environment-specific objects in a 2D picture. By creating a dataset containing images and annotations, we can use software that will learn to carry out instance segmentation by creating instance masks in real-time.

■ 2.5.1 Yolact

YOLACT [8] is a fully-convolutional model for real-time instance segmentation. YOLACT works using two main steps.

The first step is the creation of a dictionary of non-local prototype masks. This step predicts a set of k prototype masks. Usually, prototype masks are used to realize features, but YOLACT uses prototypes to realize masks in the given image.

The second step consists of predicting coefficients for each instance (object) in the image. Compared to standard detectors that only have two branches in their prediction heads, YOLACT contains three branches. One branch predicts the class confidence, the second predicts the four bounding box regressors and the third YOLACT-unique branch predicts the k mask coefficients, one corresponding to each prototype.

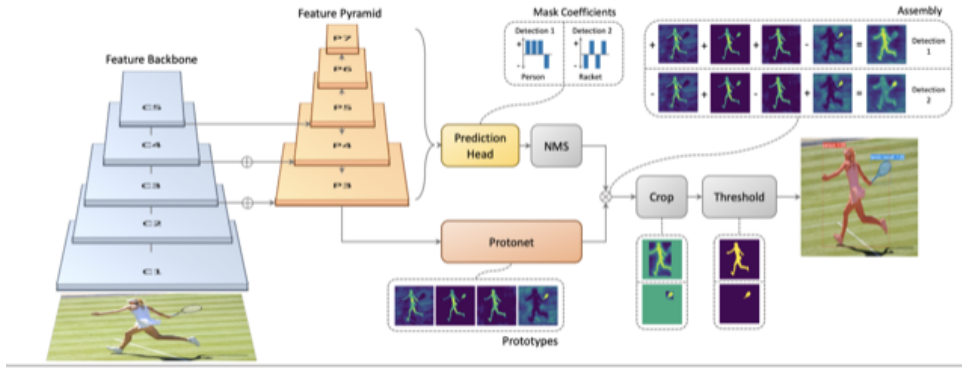


Figure 2.7: YOLACT architecture with $k = 4$ [8]

Lastly, assembling the instance segmentation for the complete image is accomplished by a linear combination of the prototypes and cropping by the predicted bounding box.

More information regarding YOLACT software is in section 6.1.

2.6 Convolutional neural networks

Let us briefly explain what a convolutional neural network (CNN) is, since YOLACT and SPSG work in a fully convolutional way.

A neural network is a structure of layers used for learning AI. It should represent a set of layers, each containing input and output. Each node in each layer can receive an input, process it, and send it to the next node in the next layer. The goal of the neural network is to produce an output or a set of outputs from an input or a set of inputs by sending the inputs through multiple layers that process them. In other words, a neural network is a function created as a concatenation of simpler functions.

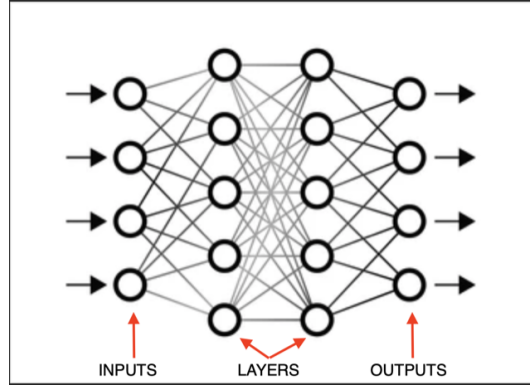


Figure 2.8: Diagram of neural network.

A specific type of neural network is a convolutional neural network. It is necessary to define what discrete convolution is. Convolution of f and g is written as $f * g$ and is defined as:

$$(f * g)(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k f(x - i, y - j)h(i, j) \quad (2.1)$$

where, x is the x coordinate in an image, y is the y coordinate and k defines the dimensions of the kernel. In practice, convolution can be used in order to detect features in images by defining specific kernels. Each kernel is used to find a particular feature, and combining multiple convolutional layers in a network makes it possible to obtain complex features in each image.

Chapter 3

Structure of the thesis summarized using a diagram

In order to summarize all the necessary steps made during this thesis a diagram was created. This diagram mentions all the processes/software used in this project, and all the inputs/outputs of these processes.

This diagram shows not only the paths that have been used in this project, but also tries to show other possible methods for solving the problematic. Every dotted arrow in this diagram shows a possible path, while the classic arrows depict our used paths.

Every circle depicts a data format and the bounded squares refer to software used in the process.

The diagram is divided into 3 main parts that can be seen on the left in figures 3.1 and 3.2. Those parts correspond to chapters in the thesis, mentioning data collection 4, data processing 5 and segmentation & training 6.

3. Structure of the thesis summarized using a diagram

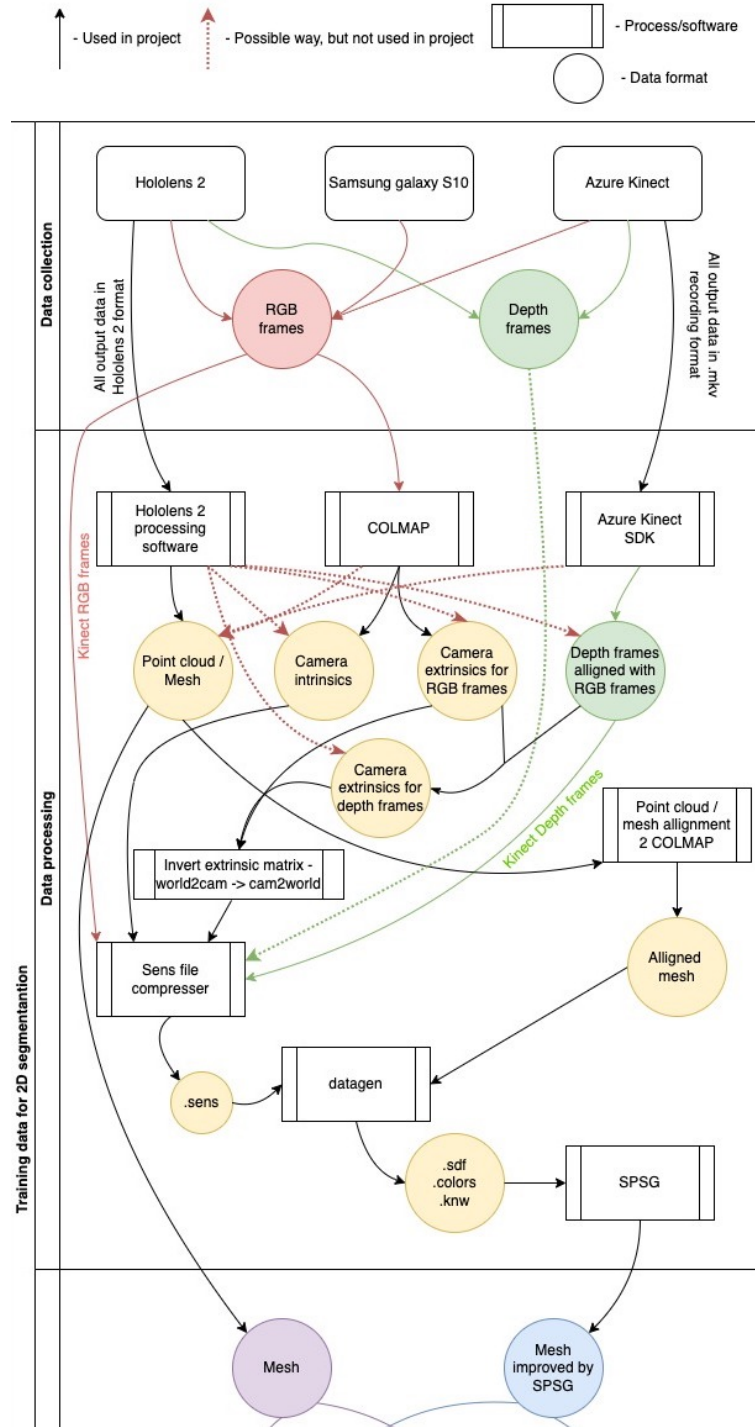


Figure 3.1: Project structure diagram part 1.

3. Structure of the thesis summarized using a diagram

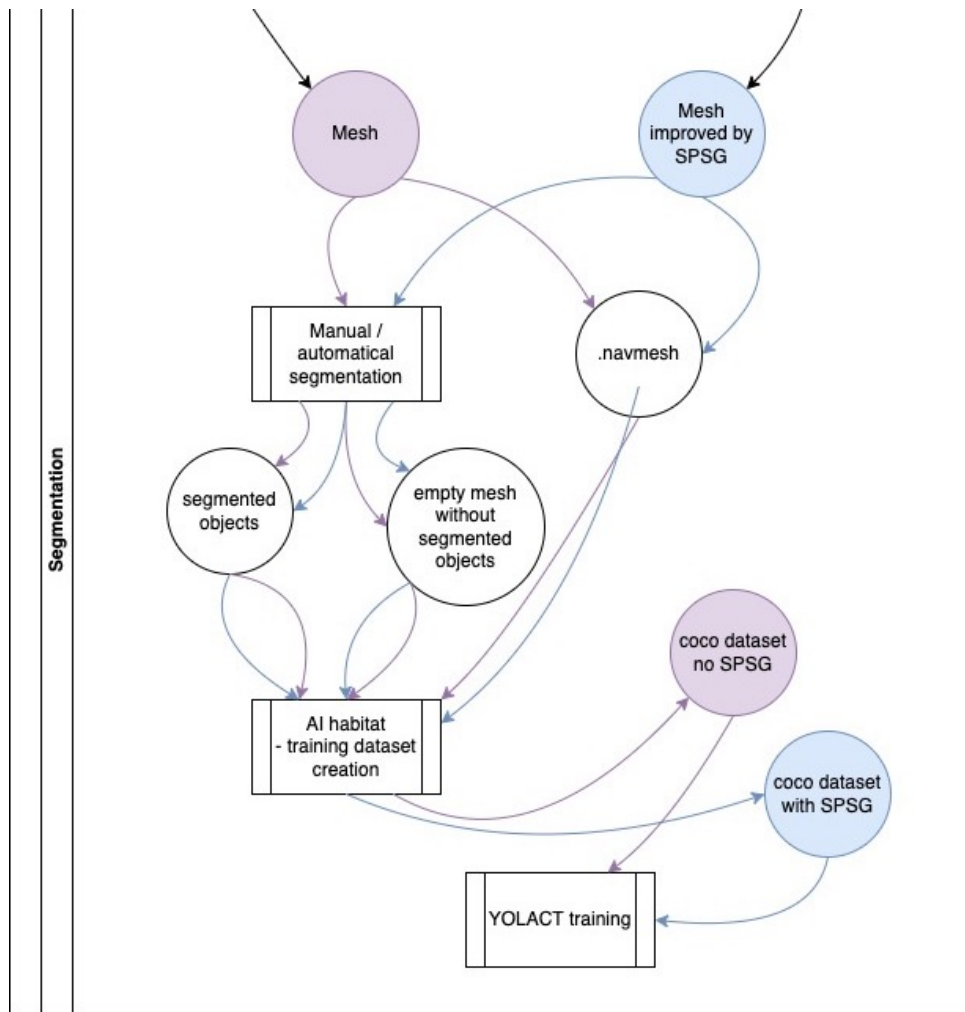


Figure 3.2: Project structure diagram part 2.

Chapter 4

Data collection

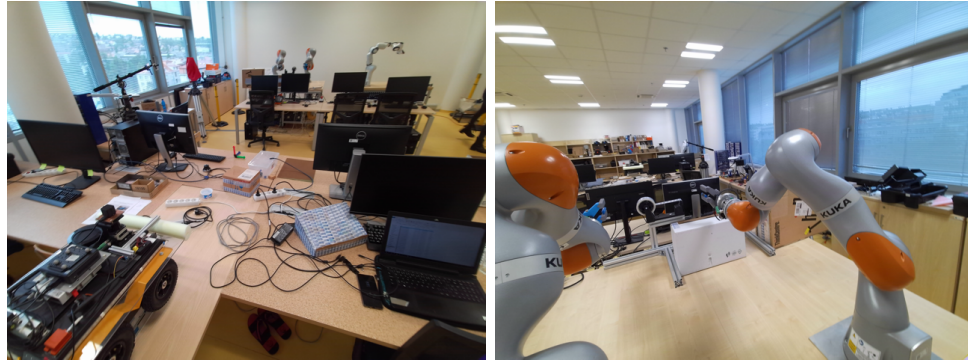
This section of the thesis describes the process of collecting the recordings. First of all, the scanned area is described. Next, the output format of each device is summarized, and the "usefulness" of the data is estimated based on our findings.

The diagram describing the data collection process is in figure 3.1.

4.1 B-635 laboratory

The scene chosen for scanning was a B-635 laboratory located in the Czech Institute of Informatics, Robotics and Cybernetics (CIIRC) in Prague. This scene was chosen because of its complexity and amount of details. We believe, that if we could scan this kind of complex environment, then there should not be any major issue with no other environment.

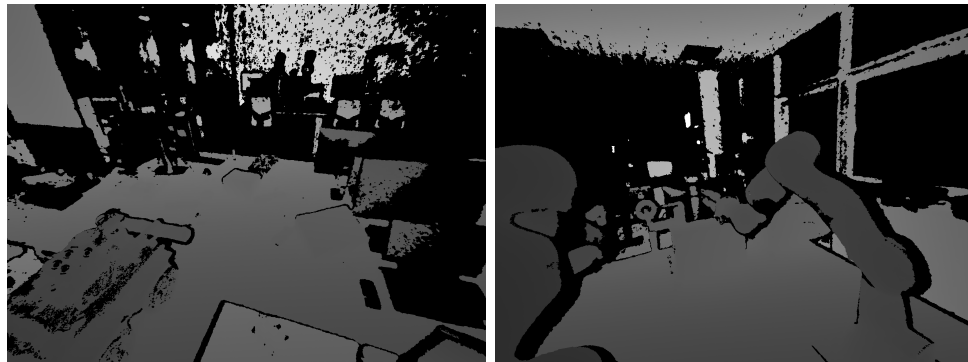
The scene contains a lot of different objects that should be ideal for instance segmentation. These objects can be categorized easily. The scene contains multiple chairs, tables, computers, and shelves. This diversity of objects should challenge the object detection training, but the scene's complexity also makes the reconstruction process more complex and potentially less accurate. Hence, we used multiple scanning devices and produced a significant amount of RGB and RGB-D images from different viewing angles.



(a) : Front view

(b) : Back view

Figure 4.1: View of B-635 laboratory - RGB format.



(a) : Front view

(b) : Back view

Figure 4.2: View of B-635 laboratory - depth format.

An example of recording images can be seen in figures 4.1 and 4.2.

■ 4.2 Recording devices

■ 4.2.1 Azure Kinect

The first recording device we used for the scanning process was Azure Kinect.

■ Hardware specifications

Azure Kinect contains two cameras. The depth camera is a one-megapixel Time-of-flight camera. The regime of depth camera used for scanning was WFOV (wide field of view) unbinned regime. This regime supports 1024x1024 resolution, up to fifteen frames per second (5FPS was used), and the operating range is 0.25 meters up to 2.21 meters. NFOV (natural field of view) or passive IR are other possible regimes.

The RGB camera is a twelve-megapixel CMOS sensor with a rolling shutter. For the RGB camera, a 4096x3072 resolution was used. This resolution supports up to 15 FPS, but only 5FPS were for this thesis. RGB camera supports six different resolution modes and two different aspect ratios. Other resolutions support up to 30FPS.

Field of views of Azure Kinect are shown in figure 4.3.

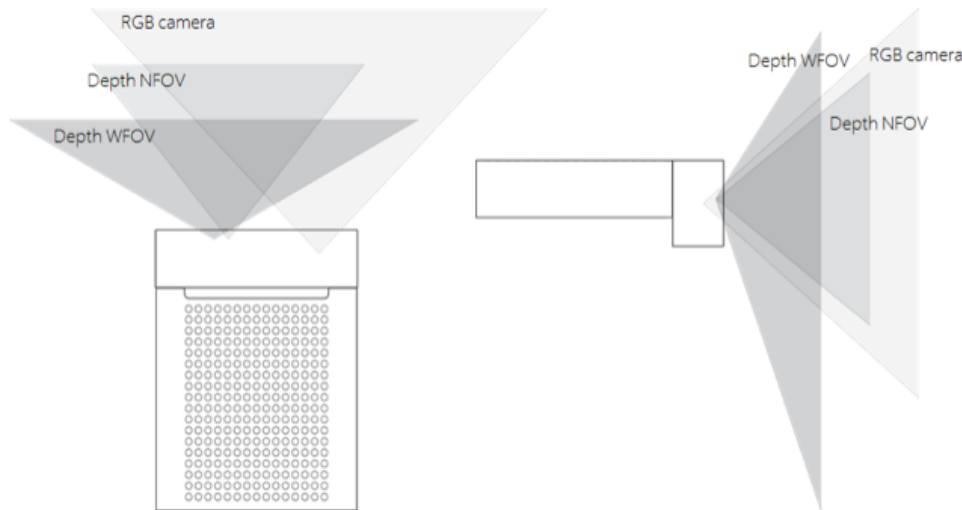


Figure 4.3: Azure Kinect - fields of view [20].

■ Data format

The output from azure Kinect is in the form of .mkv recordings. The depth and RGB frames can be easily extracted into two directories, one for each recording. Depth frames are in .png format and 1024x1024 resolution, and RGB frames in .png format and 4096x3072 resolution. In total, three recordings were made for this project.

The table below shows the number of frames for each recording and the corresponding time of each recording.

Recording	Number of images	Length of recording
1	2642	$2642 \text{ frames} \times 0.2 \frac{s}{\text{frames}} \approx 528s$
2	2306	$2306 \text{ frames} \times 0.2 \frac{s}{\text{frames}} \approx 461s$
3	2020	$2020 \text{ frames} \times 0.2 \frac{s}{\text{frames}} \approx 404s$
Total	6968	1373s

Table 4.1: Number of images and length of recordings of Azure Kinect.

■ Usefulness of data

The downside with the output data format from Azure Kinect was that the dimensions of the depth frames were not matching the dimensions of the RGB frames. Therefore the whole recording required further post-processing in order to be used by SPSP (see section 5.4.8 that describes this problematic). On the other hand, the RGB frames produced by Kinect could have been directly inputted into COLMAP for reconstruction. Also, the depth frames produced by Kinect could have been immediately used for point cloud creation by Kinect SDK software.

■ 4.2.2 Hololens 2

The second recording device we used for the scanning process was Hololens 2.

■ Hardware specifications

Hololens 2 features six different sensors. Firstly it has an 8-MP RGB camera. The regime that has been selected for our thesis was a Videoconferencing, 100 BalancedVideoAndPhoto, 120 regime, that outputs 15 frames per second in 760x428 resolution. Secondly, four head tracking visible light cameras are present on Hololens 2. They record frames in 640 x 480 resolution. Those cameras are located on the sides of the device to scan all the surroundings, and they scan the scene in grayscale format. Lastly, a 1-MP ToF depth sensor was employed. This sensor operates in two modes. The first mode is a high-frequency mode, usually used for hand tracking. The second mode, used

for this thesis, is a long throw low-frequency mode that captures up to five frames per second. This is the advised mode for spatial mapping processes.

All these devices and configuration options can be accessed by running Hololens 2 in research mode.

■ Data format

The output format for each Hololens 2 recording is a folder containing multiple .tar and .bin files. The whole directory can be processed using stream recorder code provided in [21].

For converting all images into their .png or .pgm format, a script `convert_images.py` was used. Another option is to use `process_all.py` in order to convert images into a visualizable format, but also create point clouds from depth maps and meshes from pinhole camera representation. More information about mesh creation can be found in section 5.2.2.

In total, four recordings were captured during the recording session. The first recording contained approximately 10 000 non-depth frames from the four visible light cameras. The second recording also contained about 10 000 non-depth frames, and the last two recordings contained around 5000 non-depth frames each.

■ Usefulness of data

The data from Hololens 2 were a crucial component for the dense reconstruction. Using Hololens 2, the meshes were created. Yet, the format of the depth frames was not matching the format of the RGB frames, so further processing was necessary for SPSC. Also, Azure Kinect's RGB and depth frames were in much higher resolution and therefore were more suitable for SPSC. In conclusion, Hololens 2 data was used only for meshing and COLMAP SfM.

■ 4.2.3 Samsung galaxy S10e

The third recording device we used for the scanning process was Samsung galaxy S10e.

■ Hardware specifications

Samsung Galaxy S10e features a 16MP ultrawide camera.

■ Data format

The number of acquired frames for this project was 450 RGB images. The dimensions of each frame are 4608x3456 px. The images are saved in .jpg format. The FOV of the ultrawide camera is 123°.

■ Usefulness of data

Data from this device were mainly used in COLMAP SfM, because of the wide angle of capturing. Therefore it helped COLMAP in order to complete the scene more precisely. However, this device does not have any IR sensor for depth, and therefore it could not be used for SPSSG. The pure intention of using this device was to have a wider variety of cameras for COLMAP reconstruction and to add wider images that capture a bigger area of the environment.

Chapter 5

Data processing

The diagram describing the data processing is in Figure 3.1.

5.1 Azure Kinect

5.1.1 Image processing

The total number of RGB images obtained from Azure Kinect added up to 6968 frames. Because of the scanning method the data contained a high rate of blurry images.

The first step in image processing was removing the blurry images from the dataset. For this, we created a simple python script that used the Laplacian function from the OpenCV library[22]. For each photo, a variance of the Laplacian operator serves as a simple blurriness description. This method works on the principle of measuring the second derivative of an image and then finding its standard deviation. The Laplace operator is also used for edge detection. Higher score from the laplacian function means that the image contains more edges. If the variance of the Laplace operator is below a given threshold, the image is considered blurry. Our experimentally selected threshold was chosen as 700. (i.e., `blur_threshold = 700`). Fraction of code from Github repostiory [23] is listed below.

```
score = cv2.Laplacian(image, cv2.CV_64F).var()
if score < blur\_threshold:
    os.remove(image)
```

■ 5.1.2 Depth image alignment

Another challenge that has been faced during this project was the alignment of depth images with RGB frames. Azure Kinect records depth frames in 1024 x 1024 format, and RGB frames are in 4096x3072 format. For SPSPG, the input format consists of RGB frames and Depth frames with equal dimensions and equal camera intrinsics. So, one of the options was to obtain depth frames in the format of RGB frames.

This has been achieved by using Azure Kinect SDK [9]. This software allows access to depth camera recording and RGB camera recording, so it was used for the initial extraction of data from the recording file. Moreover the Azure Kinect SDK library contains multiple useful functions, one of which is the `k4a_transformation_depth_image_to_color_camera()`.

Using this function and an example transformation script from Azure Kinect Sensor SDK, it was possible to extract the depth images in RGB format. The steps were:

Step 1: Installing the OpenCV library into the visual studio project containing the script.

Step 2: Adding a saving command using openCV's `imwrite()` function for `transformed_depth_image`.

The example transformation script can be obtained at [24].

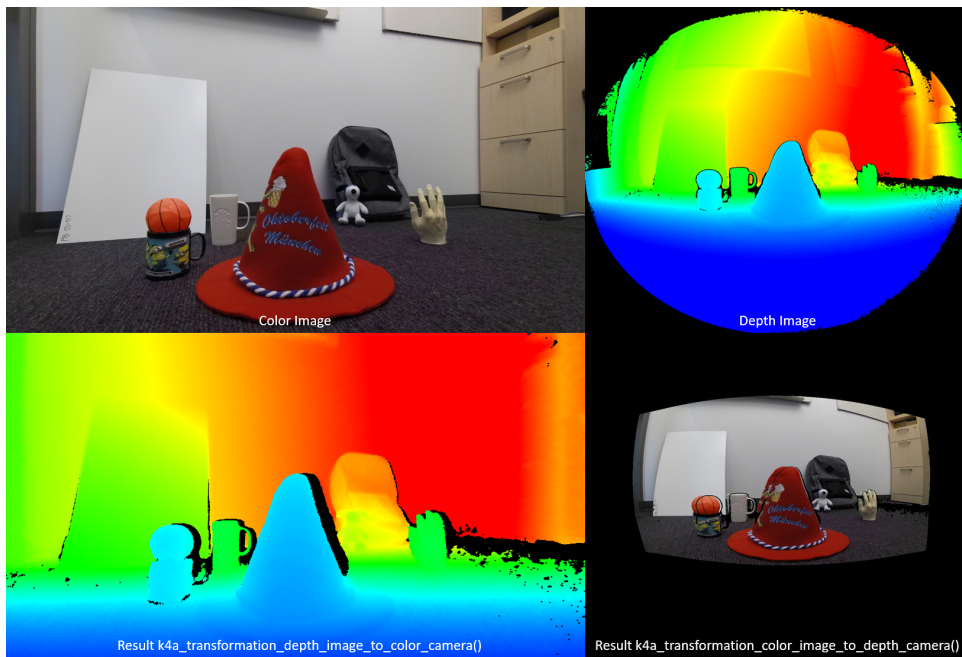


Figure 5.1: visualization of k4a transformation depth image to color camera() [25].

Afterward, a simple script in python was written in order to extract the transformed depth images for each desired timestamp. The transformation example (depth_to_color.exe in our case) takes four arguments.

1. The first one defines playback or capture mode. In our case, we used playback mode, because the goal was to extract frames from a recording.
2. The second argument is the path to the recording .mkv file.
3. The third argument is the timestamp from which the depth image should be extracted.
4. The last argument is the output path for the transformed image to be saved.

The script example is listed below:

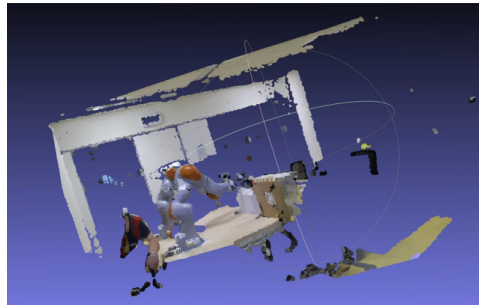
```
cnt = 1
for i in range (init_timestamp, end_timestamp, 1/FPS_in_milliseconds):
    command = f"path/_to/_depth/_to/_color.exe playback"\
    "path\_to\_recording.mkv {i} {cnt}.png"
    os.system(command)
    \indent cnt += 1
```

The timestamp for the first recording starts at 216ms. For the second recording the initial timestamp was 416ms.

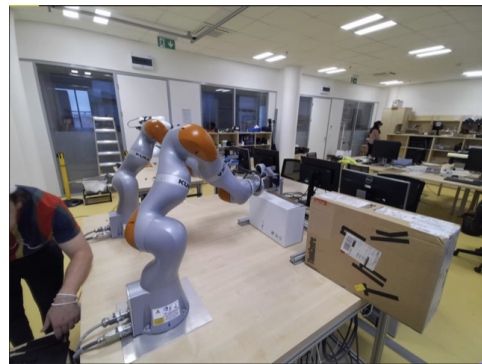
5.1.3 Point cloud/mesh

Another feature that Azure Kinect SDK offers is the point cloud creation from transformed depth images. The name of the function is `k4a_transformation_depth_image_to_point_cloud`.

The quality of those point clouds can be seen in Figure 5.2a, however the point clouds were not aligned with each other. This requires aligning each point cloud based on the RGB camera pose obtained from COLMAP. We developed codes for aligning dense point clouds into a single coordinate system. However this solution was not used in practice because an easier solution for dense point cloud creation using Hololens 2 was discovered.



(a) : Point cloud



(b) : RGB frames

Figure 5.2: RGB image and its corresponding point cloud created by Azure Kinect SDK.

■ 5.2 Hololens 2

■ 5.2.1 Image processing

From the four recordings, a total of 32 084 non-depth frames were captured. This number includes frames from all five cameras (i.e., four grayscale tracking and one RGB photometric camera).

All the four tracking cameras are rotated by 90 degrees. Therefore the first step was rotating all their frames. All the frames from the tracking cameras only consist of one color channel (mono). It was decided that the amount of data is redundant and therefore it needs subsampling.

The method chosen for subsampling the dataset was to keep all the PV camera frames and take every fourth frame from the tracking cameras. So, the total number of frames was reduced to 12 533 frames. All the Hololens 2 frames were not as blurry as the Azure Kinect images, so it was not necessary to remove blurry images using the same method as was used for Kinect.

Thus, an easy python script for rotating all images and erasing $\frac{3}{4}$ of the images was written. The code featured two functions, one for rotating and the second for erasing. Those functions can be found here [23].

The main part consisted of reading the image using the OpenCV library, creating the desired rotation matrix and applying the rotation matrix to the image using the warpAffine function (included in the CV2 library). The erasing process consisted of a modulo operation for selecting every fourth image.

■ 5.2.2 Point cloud/mesh

The last step in processing data from Hololens 2 was to obtain a mesh from the scanned scene. This could have been done in two ways. The first option was to compute the point cloud for each depth frame captured by Hololens 2. The second option was to use the TSDF function provided by Hololens 2. [21]

We started with the point cloud transformation into a common coordinate system, in order to recreate the scene. Using the `save_pclouds.py` [26] script, one point cloud was generated for each depth image in a recording. In total, more than 2000 point clouds were generated. Those point clouds were then fused together.

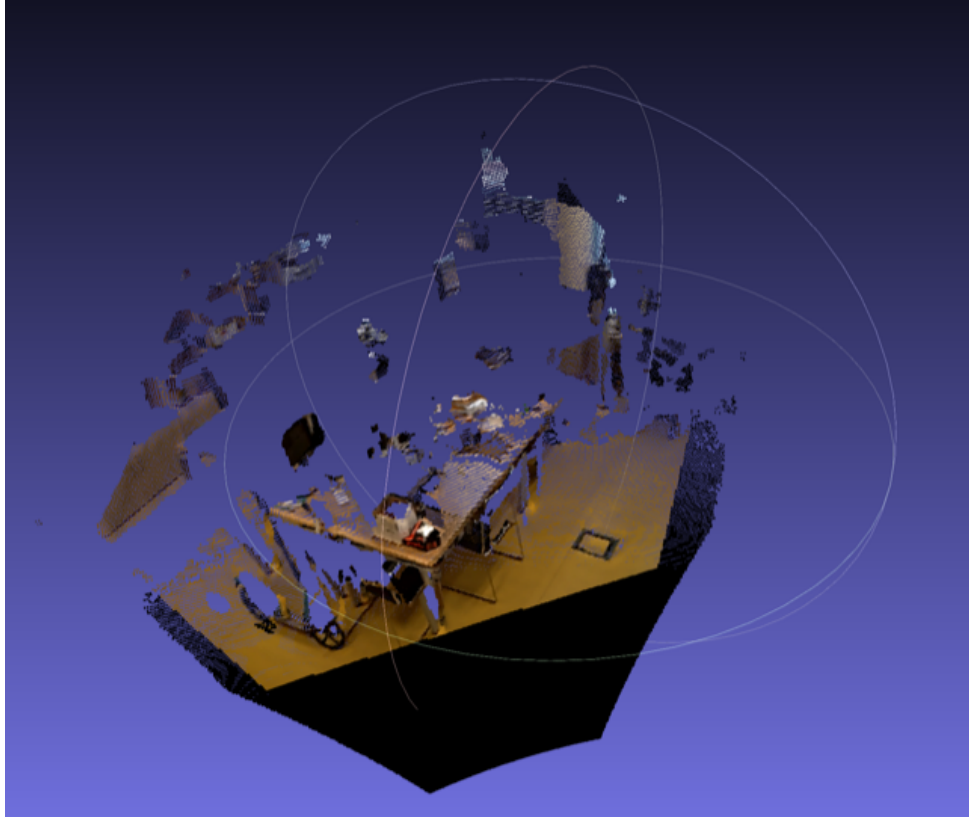


Figure 5.3: Single point cloud generated by `save_pclouds.py`.

A simple python script has been written in order to fuse the point clouds together. It used the `pymeshlab` library [15] in order to work with each of those point clouds. Each of the point clouds has been loaded using `ms.load_new_mesh(file_path)` function. After all the point clouds were loaded into separate layers, `ms.flatten_visible_layers (mergevisible=False)` was called and then the final point cloud was saved using `ms.save_current_mesh(output_path)`.

The downside of this dense reconstruction option is that the size of the generated point cloud was 1.74GB. Even though it could be subsampled and transformed into a mesh, the TSDF function allows generating mesh automatically and with a much smaller size.

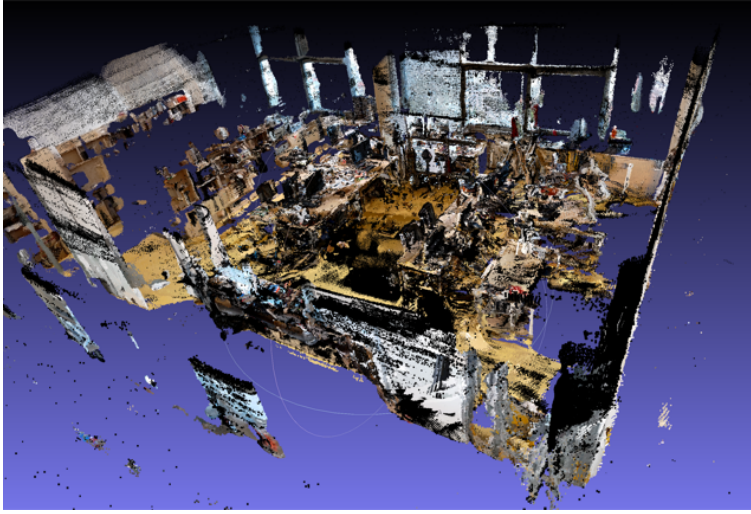


Figure 5.4: Fused point cloud generated using pymeshlab library.

The sample script for TSDF integration allows for reconstructing the scene using the depth and RGB frames and also head poses. The outputs are in the form of a point cloud but also a mesh. The fusion works using the Open3D library. Open3D is a library for manipulating 3D models [27]. The size of the output mesh was 12.9MB which is approximately 130 times smaller than the result of the first method. Smaller data are an advantage because the computing time for SPSPG and AI habitat will decrease. Moreover, by reducing the size, the quality remained the same.



Figure 5.5: Mesh generated using TSDF integration.

5.3 COLMAP

The next step after extracting and pre-processing all the frames from all devices was COLMAP. The goal of COLMAP was to create sparse reconstruction, in order to have camera extrinsics in the world to camera format, that are required for SPSPG. The following section describes the process of using COLMAP.

5.3.1 COLMAP input

We decided to use RGB frames as the input for COLMAP and to use the `-single_camera_per_folder` argument. This argument ensures that each image is categorized to its corresponding recording device. Each recording device therefore has its own camera model. The type of matcher we chose was a `vocab_tree_matcher`. In total, 19791 RGB images were used during COLMAP reconstruction.

The COLMAP work directory, before starting any work, looked like this:

```
COLMAP_work_dir
├── images
│   ├── HL_LF
│   ├── HL_LL
│   ├── HL_PV
│   ├── HL_RF
│   ├── HL_RR
│   ├── Kinect
│   └── S10
└── vocab_tree_flickr100K_words1M.bin
```

5.3.2 Sparse reconstruction

The description of COLMAP can be found in section 2.2.1. COLMAP allows multiple approaches to complete the sparse reconstruction, including an automated version of reconstruction pipeline using a feature extractor, exhaustive matcher, and mapper. The command for this automated reconstruction is:

```
colmap automatic_reconstructor
--workspace_path $DATASET_PATH
--image_path $DATASET_PATH/images
```

We chose to use `vocab_tree_matcher` instead of `exhaustive_matcher`. Therefore we did not use the `automatic_reconstructor`. This part mainly focuses on the commands and arguments used for sparse reconstruction in this thesis.

The first step was `feature_extractor`. Initially, the `$DATASET_PATH` is set to aim to the working directory seen in section 5.3.1. The script for the `feature_extractor` can be run using:

```
colmap feature_extractor
--database_path $DATASET_PATH/database.db
--image_path $DATASET_PATH/images
```

The second step was to use the `vocab_tree_matcher`. In order to use the vocab tree matching, a database needs to be downloaded from <https://demuc.de/colmap/>. Then, we can execute.

```
colmap vocab_tree_matcher
--database_path $DATASET_PATH/database.db
```

Finally, the sparse reconstruction part could be done using the `mapper`. Some important values were used for this part, so let us briefly explain them. Firstly the argument `--Mapper.ba_global_images_ratio` has been increased to 1.3 and the `--Mapper.ba_global_points_ratio` has also been increased to 1.3. This is because the bundle adjustment takes place less frequently and therefore speeds up the reconstruction process. Secondly, the `--Mapper.ba_global_max_num_iterations` has been reduced to 20 from the default 50 in order to reduce computation time. Thirdly, `--Mapper.ba_global_max_refinements` has been decreased to 2, and lastly, `--Mapper.ba_global_points_freq` has been doubled to 500000 points. All these modifications were used in order to decrease the time needed to complete the sparse reconstruction. The overall reconstruction with these arguments took around four days. According to our initial experiments, the COLMAP with using the default arguments, would take more than ten times more time to finish the reconstruction process.


```
colmap mapper
--database_path $DATASET_PATH/database.db
--image_path $DATASET_PATH/images
--output_path $DATASET_PATH/sparse
--Mapper.ba_global_images_ratio 1.3
--Mapper.ba_global_points_ratio 1.3
--Mapper.ba_global_max_num_iterations 20
--Mapper.ba_global_max_refinements 2
--Mapper.ba_global_points_freq 500000
```

5.3.3 COLMAP output

The main output after the sparse reconstruction were two text files called `cameras.txt` and `images.txt`.

The file `cameras.txt` contains the camera parameters (intrinsics) for each camera used during the reconstruction. The first parameter refers to the camera distortion model, which in our case has always been the `SIMPLE_RADIAL` camera model. We chose the model based on the experimental evaluation. The next two parameters refer to the image dimensions of each camera. Those dimensions are all mentioned in chapter 4. The next parameters define the single focal length of the camera and principal point location. The structure, therefore, looks like this:

```
CAMERA_ID CAMERA_MODEL WIDTH HEIGHT FOCAL_LENGTH
PRINCIPAL_POINT_X PRINCIPAL_POINT_Y DISTORTION
```

Since the distortion for each camera has been less than 0.05, it has been neglected in all future processing.

The `images.txt` contains the parameters for each image (extrinsics) used in the reconstruction. For us, the most important parameter has been the extrinsic parameter that defines the world to the camera position. This parameter is important because SPSG requires the camera to world matrix. Two lines in the `images.txt` file describe each image file. The first line is structured like this:

```
IMAGE_ID qw qx qy qz Tx Ty Tz CAMERA_ID FILE_NAME
```


The quaternion $q = [q_w, q_x, q_y, q_z]$ realizes rotation of the camera. T_x , T_y and T_z describe the translation of the camera in the world coordinate system. The second line contains information about each image key point. These key points have not been further used in our thesis, therefore they are ignored in this description.

■ 5.4 SPSG

Running the SPSG software was one of the main goals of this thesis. It should create the main difference between our approach and the SoTA approach. However, installing and running the software was much harder than expected.

Firstly the lack of any documentation made this task extremely difficult. The correct inputs and outputs were achieved experimentally by trying different options, adding missing parts of the source code and comparison of different results.

Secondly, many mistakes and missing parts of the code were discovered during the process and hence many changes needed to be implemented. The only solution for resolving these issues was to communicate with the authors, which was not optimal because of the time it took to receive a response and because of the abstraction of the answers. On the other hand, it would not be possible to get to the final results without their help and therefore their time is highly appreciated.

In the following section, the whole process of running SPSG is explained. This section might contain some ambiguities, but as mentioned above, that is mainly caused by the lack of documentation and many inconsistencies in the SPSG software. Note that we provide the first available description of how to install, run and use custom datasets with SPSG [6].

■ 5.4.1 Installation

The installation process is straightforward. SPSG is running on python 2.7 and Pytorch 1.2.0. The optimal way for installing is to create an anaconda environment with the correct version of python.

After enabling the environment it is necessary to install the correct version of Pytorch. This part has been a bit challenging, because while installing Pytorch with CUDA, PyTorch is dependent on NVIDIA drivers installed. The SPSG software was supposed to run on a remote server where multiple other applications are installed, so while trying to update to the correct version of Pytorch, the NVIDIA drivers were updated, which caused SPSG to work, but other applications crashed.

Therefore the goal was to find an optimal version of the NVIDIA driver that supports both PyTorch and all the other software installed on the server. After experimenting, the NVIDIA-SMI 440.33.01 with driver version: 440.33.01 was found. This driver was running CUDA version 10.2 and allowed all processes on the server to run correctly.

After successfully installing PyTorch and enabling the conda environment with python 2, all the extension modules were installed by running the `install_utils.sh` script provided by the authors.

■ 5.4.2 Testing on data provided by the authors

The next step was to test the software on the example data provided by the authors. After successfully downloading all the data, we managed to run `training.py` and `eval.py` scripts on the example data without any issues. Since everything was running as it should, we moved on to the next step, which was running SPSG on our custom dataset.

■ 5.4.3 Structure of SPSG

To correctly explain the process of running SPSG on a custom dataset, it is necessary to refer back to figure 3.1

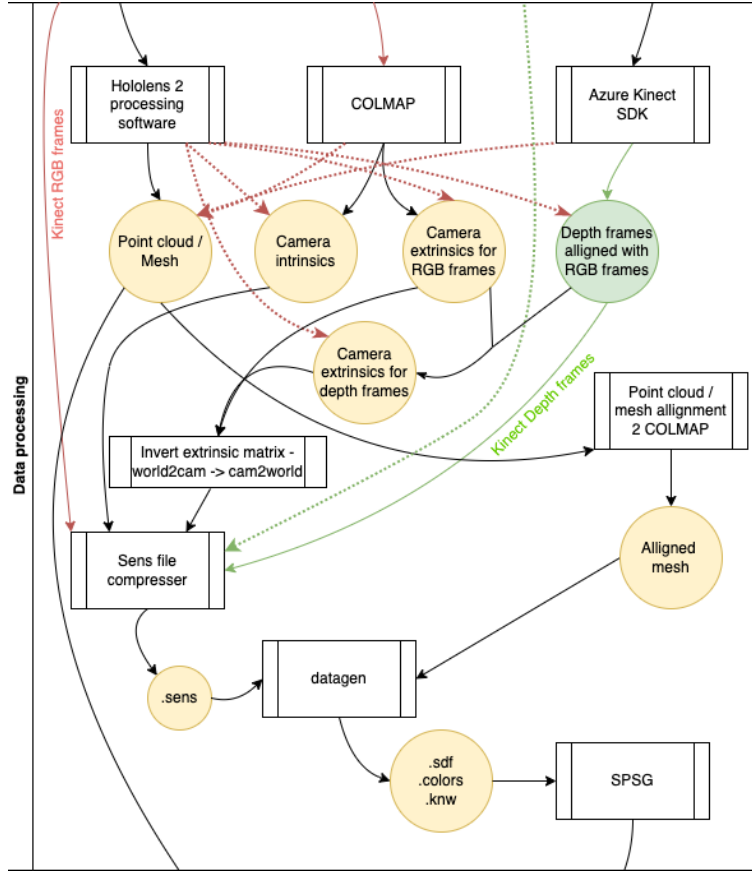


Figure 5.6: Data processing and SPSG.

In this state, we already obtained a mesh, camera intrinsics, extrinsics and depth frames aligned with RGB frames. The input for SPSG is in form of .sdf, .knw and .colors files. All information about creating these three files is missing and is not available on the official Github page nor in the short documentation.

After some investigation and communication with the authors, it was made clear that there is another part of the software that has been used for the creation of .sdf files in the past. This part of the software is called datagen and was available on the Github page of the SGNN project [28]. SGNN is another scene completion project written by the same authors.

When the datagen was installed, another obstacle was discovered. Datagen requires specific input data. However, it was unclear where to obtain them. Luckily, we found that the example data are from the Matterport 3D dataset [29]. That allowed us to examine the datagen input format. The datagen requires a mesh as the input. This mesh has to be in some "reduced" form.

Since there was no mention of what the "reduced" form is, we found out the number of faces smaller than the full number of faces leads to working code.

Finally, datagen generated .sdf data. For SPSG however, .colors and .knw files are required. After another communication with the authors, they provided the correct version of datagen, that has been used for SPSG. This version was finally producing correct data, that could be used by SPSG, so the only missing part was to run datagen on our custom dataset.

Since there was no mentioned requirement about the orientation of the mesh, we had to find out that datagen needs to have its principal axis aligned in a way that the Z-axis is aiming up towards the roof of the scan. Moreover datagen required all the data provided in the form of a .sens file, but there exists no script or code that would compress data into this format.

To summarize things, a list of necessary steps to run SPSG is provided:

Step 1: Gather all necessary data (intrinsic, extrinsic and RGB-D frames).

Step 2: Install SPSG.

Step 3: Install datagen [30].

Step 4: Align principal axis.

Step 5: Create .sens file.

Step 6: Run datagen in order to create .sdf/.knw/.colors files.

Step 7: Run SPSG.

■ 5.4.4 Datagen

When the correct version of datagen was obtained from the authors, the first step was to install it correctly.

Datagen requires a few additional dependencies to be installed and linked to the project. Firstly the mLib and mLibExternal libraries need to be installed and linked to the project. However, we found out that the version of the libraries provided is incorrect and they need to be updated. After downloading

the correct version [31] of libraries and installing them, we needed to add them to the additional include directories in the Visual Studio project.

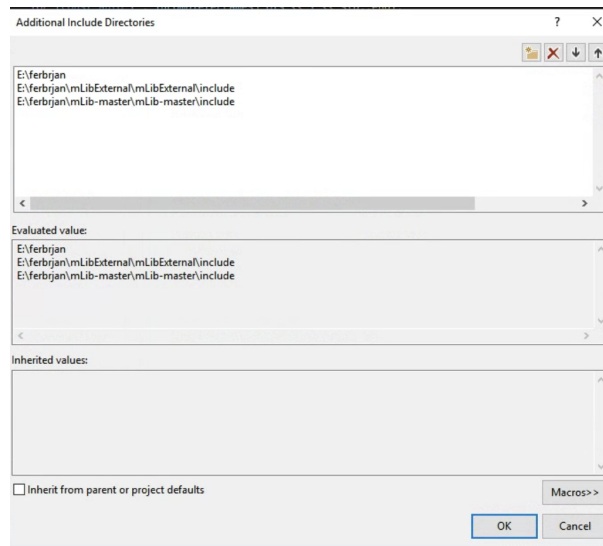


Figure 5.7: Additional include directories.

Secondly, two other libraries not mentioned in the datagen documentation needed to be installed. These are FW1FontWrapper.lib and Freeimage.lib. Those needed to be written into additional dependencies like this:

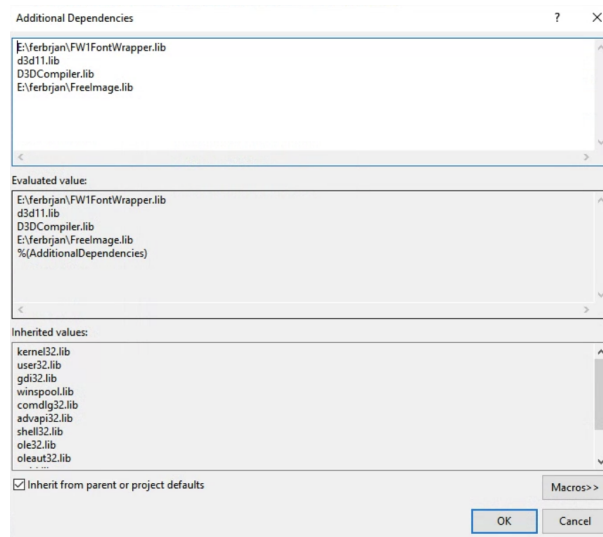


Figure 5.8: Additional Dependencies.

The input for datagen comprises of .sens file, which is a file containing RGB frames, depth frames, camera extrinsics, camera intrinsics (all in binary format) and the reduced mesh. Reduced mesh is a mesh that has its face attributes removed. Another input part is the zParametersScanMP.txt file, which defines all the required paths and optional arguments. All the arguments from zParametersScanMP.txt are listed below:

argument	value
s_bDebugVis	True
s_sceneFileList	path_to_scene_file_list
s_scanPath	path_to_scanpath
s_scanLabelFile	not required
s_labelName	not required
s_labelIdName	not required
s_incompleteFramePath	path_to_output_incomplete_frames
s_outputCompletePath	path_to_output_complete
s_outputIncompletePath	path_to_output_incomplete
s_maxNumScenes	0 //no maximum
s_maxNumSens	3 //depends on your number of .sens
s_renderWidth	320
s_renderHeight	240
s_BRDF	0
s_cameraFov	60.0f
s_minDepth	0.4f
s_maxDepth	6.0f
s_addNoiseToDepth	false
s_depthNoiseSigma	0.01f
s_filterDepthMap	true
s_depthSigmaD	5.0f
s_depthSigmaR	0.1f
s_edgeNeighborhoodThresh	0.7f
s_edgeDepthThresh	0.25f
s_voxelSize	0.02f
s_renderNear	0.1f
s_renderFar	10.0f
s_scenePadding	6
s_heightPad	3
s_bSaveSparse	true
s_bUseRenderedDepth	false
s_trajCachePath	./output/traj_cache
s_chanceDropFrames	0.8f
s_bGenerateSdfs	true
s_bGenerateKnown	true
s_bGenerateColors	true

Table 5.1: zParametersScanMP.txt arguments.

The `s_sceneFileList` parameter points to the file that should summarize all scene parts. In this project, we have one scene with one room. Thus the file only includes one line (`B-635_room0`).

The `s_scanPath` parameter defines the path to the directory with the `.sens` files and the reduced mesh. The scan directory should have form:

```
scans
├─ scene_name (B-635 in our case)
│   └─ region_segmentations
│       └─ region0.reduced.ply
│           └─ sens
│               ├── scene_name_0.sens
│               ├── scene_name_1.sens
│               └─ scene_name_2.sens
```

The rest of the parameters are advised, by the authors, to be set as it is in Table 5.1. The specification refers to three `.sens` files in case we have three different cameras with different intrinsics. In case only one camera was used, the remaining two `.sens` files can be empty, or the data can be distributed into multiple `.sens` files or the number of `sens` files in `zParametersScanMP.txt` can be set to 1.

Since the `nyu40` dataset used for labeling is unnecessary, line 14 in `visualizer.cpp` [30] using the `s_scanLabelFile` can be commented.

5.4.5 Point cloud alignment

The mesh used for `datagen` has been generated by Hololens 2. However, the camera extrinsics were generated by COLMAP. Those two softwares have different coordinate systems that need to be aligned first.

This has been done by aligning camera centers from Hololens 2 and the corresponding camera centers from COLMAP and using the `Matlab procrustes()` function. This function provides us with rotation, translation, and scale parameters. As a result, we can use extrinsics provided by COLMAP and employ images captured by Kinect, because they have the best accuracy, resolution and overlap between depth and RGB frames.

Firstly, it is necessary to explain how to obtain camera centre from the projection matrix. Projection matrix is in the form:

$$\begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_x \\ r_{2,1} & r_{2,2} & r_{2,3} & t_y \\ r_{3,1} & r_{3,2} & r_{3,3} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.1)$$

Where $\mathbf{R} = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} \\ r_{2,1} & r_{2,2} & r_{2,3} \\ r_{3,1} & r_{3,2} & r_{3,3} \end{bmatrix}$ is the rotation matrix and $\mathbf{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$ is the translation vector. To obtain the camera centre we can use the :

$$\mathbf{C} = -\mathbf{R}^T \mathbf{t} \quad (5.2)$$

Where $\mathbf{C} \in \mathbb{R}^3$ represents the camera center and \mathbf{R}^T represents the transpose of \mathbf{R} .

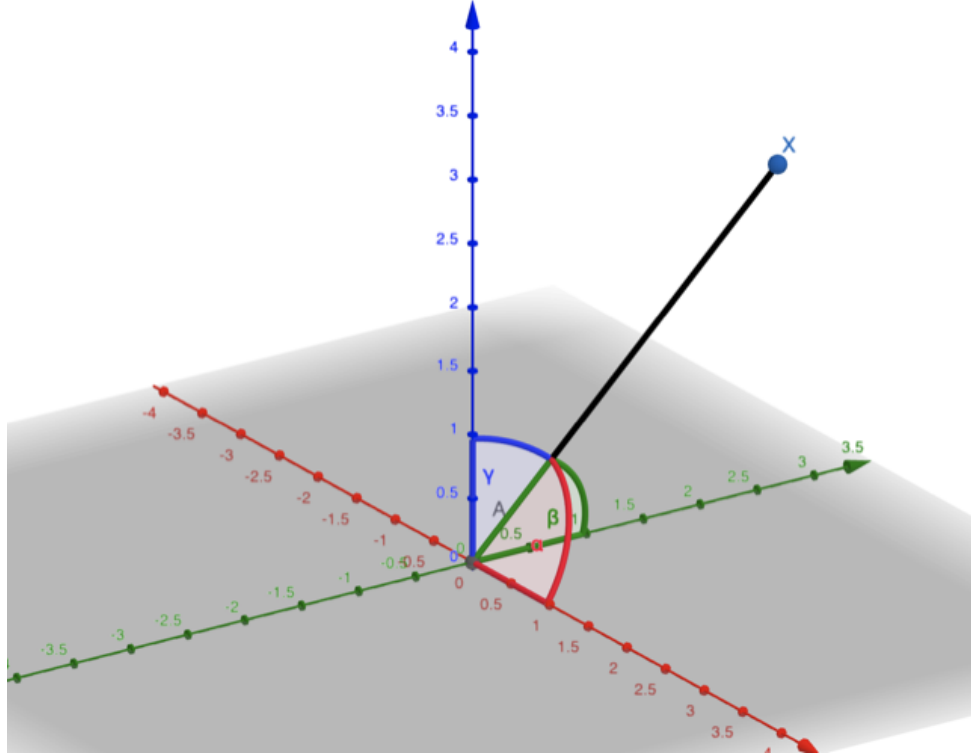


Figure 5.9: representation of α, β, γ in geogebra 3D [32].

Hololens 2 uses this format of extrinsic parameters. However, COLMAP uses a slightly different format ,i.e., a quaternion and translation. To explain

how quaternions work, we need to define two values. The first value represents the angle of rotation. Lets define this value as θ . The second value is the axis of rotation around which we want to rotate. Let us define this value as vector \mathbf{b} consisting of 3 values representing the angles between the vector and the three positive coordinate axes (seen in figure 5.9). We can also call the components of the vector direction cosines. From the figure above we can see the axis of rotation \mathbf{x} (visualized by line connecting origin and point X) and the corresponding direction cosines.

Using values θ and $\mathbf{b} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$ we can define quaternion as:

$$\begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos(\theta/2) \\ \sin(\theta/2)\cos(\alpha) \\ \sin(\theta/2)\cos(\beta) \\ \sin(\theta/2)\cos(\gamma) \end{bmatrix}. \quad (5.3)$$

The formula for conversion between quaternion and rotation matrix is defined as [33]:

$$\mathbf{R} = \begin{bmatrix} 2(q_w^2 + q_x^2) - 1 & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) \\ 2(q_x q_y + q_w q_z) & 2(q_w^2 + q_y^2) - 1 & 2(q_y q_z - q_w q_x) \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 2(q_w^2 + q_z^2) - 1 \end{bmatrix}. \quad (5.4)$$

After transforming both extrinsics from COLMAP and Hololens 2 to the camera centres we are ready to use the `procrustes()` function in matlab. Lets label camera centres from COLMAP as C_{COLMAP} and camera centres from Hololens 2 as $C_{hololens}$. The function can then be used as:

$$[d, Z, transform] = procrustes(C_{COLMAP}, C_{hololens})$$

The output of `procrustes()` then returns the transformation in the form of $s = \text{transform.b}$ representing the scale, $\mathbf{R} = \text{transform.T}$ representing the rotation and $\mathbf{t} = \text{transform.c}$ representing the translation. The formula for transforming each point $X_H \in \mathbb{R}^3$ of the Hololens 2 mesh into the COLMAP coordinate system (i.e. $X_C \in \mathbb{R}^3$), is:

$$X_C = s \times X_H \times \mathbf{R} + \mathbf{t}. \quad (5.5)$$

Saving the transformed model as a point cloud was necessary after transforming each point of the original mesh, because we only transformed points and not the normals. The script used for point cloud alignment is called `align_holo2colmap.m` and it can be found here [23].

■ 5.4.6 Point cloud rotation

After aligning the point cloud to the correct coordinate system, the next necessary step was rotating the model to the correct orientation. Each point of the point cloud has been rotated using a rotation matrix that has been obtained using simple linear algebra. Moreover, the extrinsics in the COLMAP model also needed to be rotated to be used by datagen. The following section defines the mathematics behind this process.

First of all, 2 points were extracted from the point cloud to create a reference vector \mathbf{v} . This vector \mathbf{v} consisted of three values, each representing a difference of two points in the desired axis. $\mathbf{v} = \begin{bmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \end{bmatrix}$ Then, we needed to create a vector that would define our new coordinate system. Since we needed the z coordinate to point upwards we chose a vector $\mathbf{u} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

Using those two vectors, we obtain the axis of rotation \mathbf{w} and the angle α .

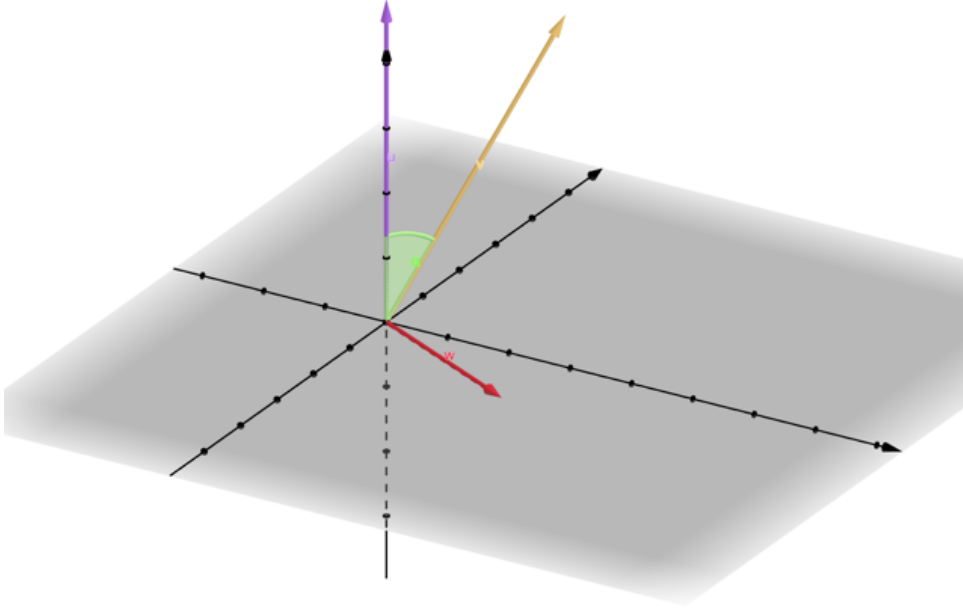


Figure 5.10: representation of α and \mathbf{w} in geogebra 3D [32].

Axis of rotation \mathbf{w} is perpendicular to \mathbf{u} and \mathbf{v} , therefore we can use the cross product of normalized \mathbf{u} and \mathbf{v} . In the thesis we specifically used formula

$$\mathbf{w} = \mathbf{u} \times \frac{\mathbf{v}}{\|\mathbf{v}\|} \quad (5.6)$$

because \mathbf{u} is already a normalized vector.

The angle α can be obtained using the dot product:

$$\alpha = \frac{\mathbf{v}^T \mathbf{u}}{\|\mathbf{v}\|}. \quad (5.7)$$

Now, the goal was to create a rotation matrix \mathbf{R} using $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$ and α . The formula for generating the rotational matrix using [33] is:

$$\mathbf{R} = \cos(\alpha)\mathbf{I} + (1 - \cos(\alpha))\mathbf{w}^T \mathbf{w} + \sin(\alpha)[\mathbf{w}]_{\times}, \quad (5.8)$$

in other words:

$$\mathbf{R} = \cos(\alpha) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + (1 - \cos(\alpha)) \begin{bmatrix} w_1 w_1 & w_1 w_2 & w_1 w_3 \\ w_2 w_1 & w_2 w_2 & w_2 w_3 \\ w_3 w_1 & w_3 w_2 & w_3 w_3 \end{bmatrix} + \sin(\alpha) \begin{bmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{bmatrix}. \quad (5.9)$$

Each point \mathbf{p} of the point cloud can be rotated by using:

$$\mathbf{p}_{rot} = \mathbf{R}\mathbf{p}. \quad (5.10)$$

This process rotates the whole point cloud, so the z-axis is aimed upwards. The script that made this transformation is called `align_scene_to_main_axis.m` [23]. However, rotating the point cloud was not the only necessary script since the camera extrinsics from COLMAP needed to be transformed. Therefore we also load the COLMAP positions, transform them, and write them into a new `images.txt` file.

In order to derive the equations 5.13, 5.14, 5.15 we need to state the definition of projection equation:

$$\lambda \begin{bmatrix} \mathbf{u} \\ 1 \end{bmatrix} = \mathbf{K} [\mathbf{Q}x + \mathbf{t}] = \mathbf{K} [\mathbf{Q}_{rot}x + \mathbf{t}_{rot}] \quad (5.11)$$

where \mathbf{K} is the calibration matrix. From original `images.txt` the extrinsics are loaded as rotation \mathbf{Q} and translation \mathbf{t} . Using those two parameters it is possible to define a camera centre \mathbf{C} using:

$$\mathbf{C} = -\mathbf{Q}^\top \mathbf{t}. \quad (5.12)$$

This camera centre \mathbf{C} can be then rotated using the rotation matrix \mathbf{R} :

$$\mathbf{C}_{rot} = \mathbf{R}\mathbf{C}. \quad (5.13)$$

The updated rotation matrix \mathbf{Q}_{rot} used in extrinsics is then:

$$\mathbf{Q}_{rot} = \mathbf{Q}\mathbf{R}^T. \quad (5.14)$$

The updated translation vector is then:

$$\mathbf{t}_{rot} = \mathbf{t}. \quad (5.15)$$

■ 5.4.7 Point cloud to mesh

The previous two sections describing point cloud alignment and point cloud rotation work with point clouds and not meshes. So, the next step is to transform the point cloud back to mesh. For this process, we used meshlab. [15]

The conversion from point cloud to mesh can be done in 3 phases: computing normals, optimizing the point cloud and converting the optimized point cloud into a mesh.

The first phase can be done using the filter called Compute Normals for Point Sets and selecting the default settings.

Optimizing the point cloud can be obtained by point cloud simplification and reducing the number of samples. In this thesis, it was not necessary to drastically reduce the number of samples because our point cloud was not that big and detailed. The number of vertices of our mesh is approximately 300 000.

Lastly, using the "surface reconstruction: ball pivoting" filter and using the default settings makes it possible to recreate the surfaces and obtain a mesh.

■ 5.4.8 .sens file creation

As previously mentioned, one of the necessary inputs for datagen is a .sens file. It describes sensor data type, and it consists of multiple parts. Each .sens file needs to contain a struct that is structured as follows (according to ScanNet [34]):

```

struct SensorData {
    unsigned int m_versionNumber;
    std::string m_sensorName;

    CalibrationData m_calibrationColor; //4x4 intrinsic matrix
    CalibrationData m_calibrationDepth; //4x4 intrinsic matrix

    COMPRESSION_TYPE_COLOR m_colorCompressionType;
    COMPRESSION_TYPE_DEPTH m_depthCompressionType;

    unsigned int m_colorWidth;
    unsigned int m_colorHeight;
    unsigned int m_depthWidth;
    unsigned int m_depthHeight;
    float m_depthShift; //conversion from float[m] to ushort

    std::vector<RGBDFrame> m_frames; // <= Main data
    std::vector<IMUFrame> m_IMUFrames;
}

```

This whole file is encoded in binary format. If the scan does not contain any IMU frames, those IMU frames can be left out. Also, each RGB-D frame should contain the following data: RGB frame in .jpg format, depth frame compressed using zlib, camera extrinsic (i.e., the camera pose) for a given frame, and timestamp for each frame (that can be left as 0).

The calibration data defines the camera intrinsics. In order to make things more simple, our depth frames were converted into the format of our RGB frames, therefore the intrinsics for depth and RGB frames are the same. The calibration matrix can be obtained by using the focal length and principal point coordinates. Lets label our focal lengths as f_x and f_y and principal points as c_x, c_y . Since we chose a camera mode that only specifies a single focal length let us assume that $f_x = f_y$.

$$calibration_matrix = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.16)$$

No official code for compression into .sens has been published by the authors. Therefore we publish a script called compressor.py [23] that has been created

to compress data into .sens files. The input for compressor.py needs to be structured as follows:

```

./
├── color
│   ├── RGB_frame_1
│   └── ...
├── depth
│   ├── depth_frame_1
│   └── ...
├── intrinsic
│   ├── intrinsic_color
│   ├── intrinsic_depth
│   ├── extrinsic_color
│   └── extrinsic_depth
├── pose
│   ├── camera_extrinsic_1
│   └── ...

```

It is unclear what is meant by `extrinsic_color` and `extrinsic_depth`, but in the Matterport 3D dataset, those files were each a four-by-four identity matrix. Therefore, the same was implied for our dataset, and an identity matrix was used.

All the camera extrinsics from modified colmap COLMAP `images.txt` and `cameras.txt` files have been extracted and transformed into matrixes using `extract_poses.py` script [23]. Note that compared to COLMAP, SPSG requires the matrixes in camera to world format so an inverse of the matrix needs to be computed.

5.4.9 SPSG

Finally, after producing all the necessary data for running datagen, it is possible to move on and use the SPSG. Datagen generates two output directories, with complete and incomplete reconstructions. Those two reconstructions are in the form of .sdf, .colors, and .knw files, which is the required input for SPSG. Therefore, those two obtained directories are used as the primary data for SPSG.

SPSG software can be divided into two parts, training, and testing. The training part trains the model on example data, and the testing part takes care of the actual reconstruction using the pre-trained model. In our thesis, the

provided pre-trained model has been used. The pre-trained model spsg.pth can be found on the official Github repository of SPSG [30]. Using this pre-trained model, we run the testing part on our dataset. We believe that retraining may improve the results, however it is unclear how to run it.

Since there is no exact documentation describing all the possible arguments, the default arguments provided by the example command were used. The incomplete output from datagen has been used as input, and the complete data has been used as the target. The command that we have used for reconstruction is:

```
python test_scene_as_chunks.py
--gpu 0 --input_data_path path/to/datagen/incomplete
--target_data_path path/to/datagen/complete
--test_file_list ../filelists/your_sdf_files.txt
--model_path path/to/SPSG.pth
--output output/directory
--num_to_vis 20
```


The following pictures show the reconstructed scene using SPSG:



(a) : Front view SPSG



(b) : Front view without SPSG

Figure 5.11: SPSG reconstruction front view.

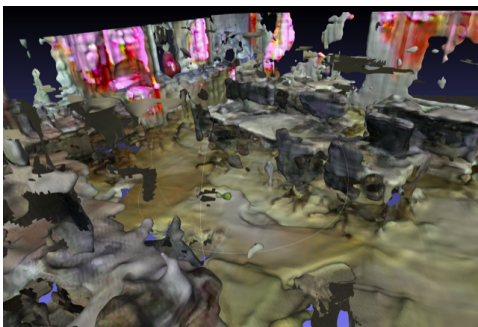


(a) : Back view SPSG

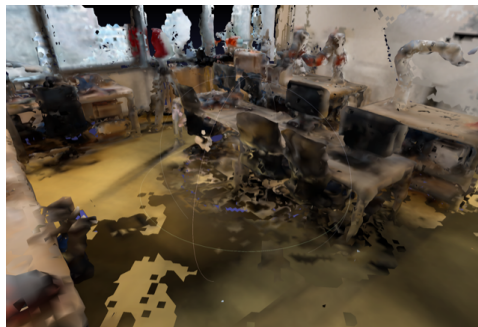


(b) : Back view without SPSG

Figure 5.12: SPSG reconstruction back view.



(a) : Side view SPSG



(b) : Side view without SPSG

Figure 5.13: SPSG reconstruction side view.

Even though SPSG was able to fill in the holes, the overall quality of the reconstruction decreased. Many artifacts and flying objects can be found in the SPSG version, and also the whole roof is missing. Initially, it was thought that the SPSG was running incorrectly. However, after multiple tests, it has been concluded that this is the correct output that SPSG should produce. More details about the quality of the reconstruction and the "usefulness" of the software are discussed in section 7.3. We observe the same behaviour on the official example as can be seen in Figures 7.1 7.2 and 7.3.

Chapter 6

Segmentation and training

This chapter describes the process of 3D segmentation, the creation of instance masks used in the COCO dataset, training YOLACT and the evaluation of the training of our instance segmentation.

6.0.1 Meshlab segmentation of 3D model

After generating the adjusted mesh using SPSG, the next step was to segment different objects manually. This process has been done using Meshlab [15].

Our goal was to segment two different meshes, the one with SPSG reconstruction and the second one without the SPSG reconstruction. Since SPSG rescales the mesh used, the SPSG mesh had to be rescaled approximately back to its original size.

For the SPSG model, the mesh was segmented into four object categories. Those categories are chair, PC, table, and wardrobe. The non-SPSG model has been segmented into seven object categories: chair, PC, pole, robot, table, unidentifiable, and wardrobe. The non-SPSG model has more categories because by using SPSG, many of the objects merged and created weird-looking objects that could not be segmented. Moreover, the robots segmented in the non-SPSG model disappeared during the reconstruction and thus could not be segmented. However, the main categories for segmentation which are chair, PC, wardrobe and table, remained the same for comparison of results.

In the Figures 6.1 6.2a and 6.2b examples of the 3D segmentation can be seen. It is visible that the model does not look realistic, however there is a possibility of using the original scanned images with the mask, if the masks did not contain any holes.

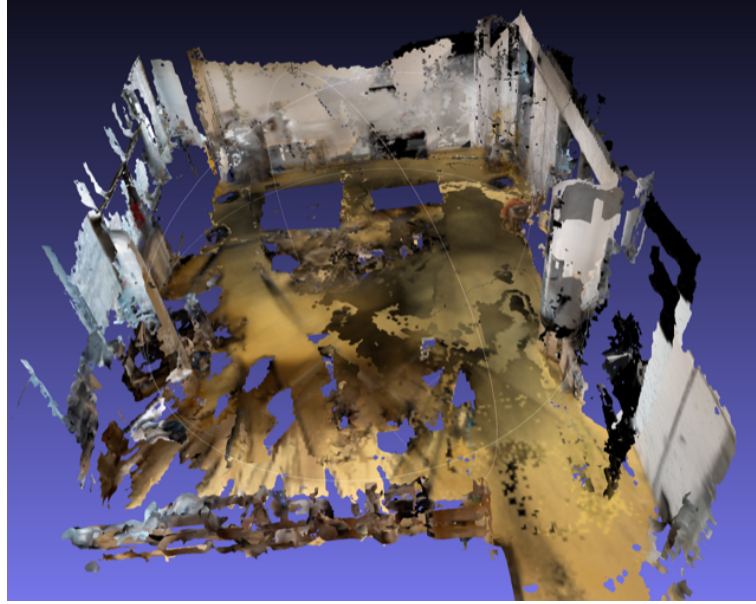
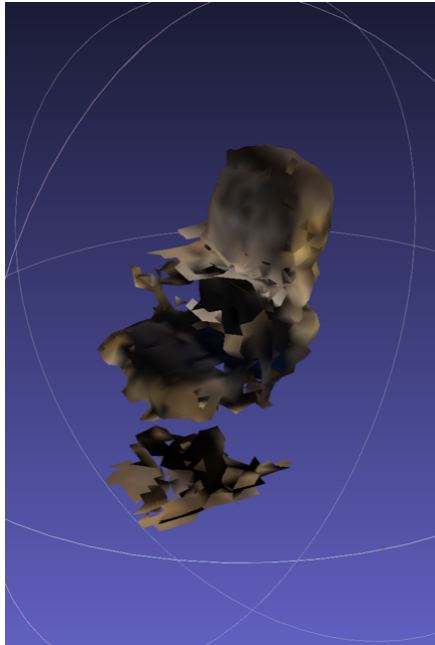
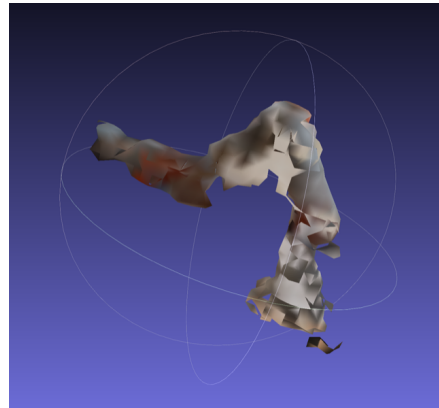


Figure 6.1: Segmented empty map of non-SPSG model.



(a) : Segmented chair



(b) : Segmented robot

Figure 6.2: Segmented object examples.

■ 6.0.2 AI Habitat

To generate the training data (i.e., RGB images and instance segmentation masks), AI habitat was used. The script that has been written is a modified version of a previously written [35] script used for experiments in BROCA hospital.

The objects were converted using obj2glb.py script [23], and the training data have been generated using generate_dataset.py [23].

■ Conversion to .glb

Firstly, all the segmented models needed to be converted into .glb format. This means that both scenes and all objects needed to be transformed from .ply/.obj into .glb. This has been achieved using obj2gltf software [36]. The essential thing is that all the data used by AI habitat need to have their roof aiming towards the positive y-axis. To do that, all the objects and meshes were rotated 90° around the x-axis.

Additionally, a corresponding .json file needs to be created for each object. It is enough if this .json contains only the name of the .glb file.

■ Generation of NavMesh

Another important step is to generate .navmesh for both scenes. The NavMesh represents the area where the center of the agent can move. This has been done using datatool software provided by habitat. For the generated .glb scene file with the correct orientation, it was sufficient to run:

```
./datatool create_navmesh  
input_file.glb  
output_file.navmesh
```

After the .navmesh file is created, the only necessary steps are to load

the correct scene file with its corresponding .navmesh file and add all the segmented objects into the scene in AI habitat.

■ Semantic camera

Since all the segmented objects have been sorted into directories based on their category, each object has been added to the scene with a label corresponding to its original directory. Addition of files from corresponding category directories can be seen at [23] in script `generate_dataset.py`, on lines 161 - 178.

After using AI habitat to add all the objects into the scene with their corresponding category ids, it is possible to start using the semantic camera. The semantic masks display colors based on the category id of the object that is visible in the frame. Visualization of the semantic camera can be seen in Figure 6.3.

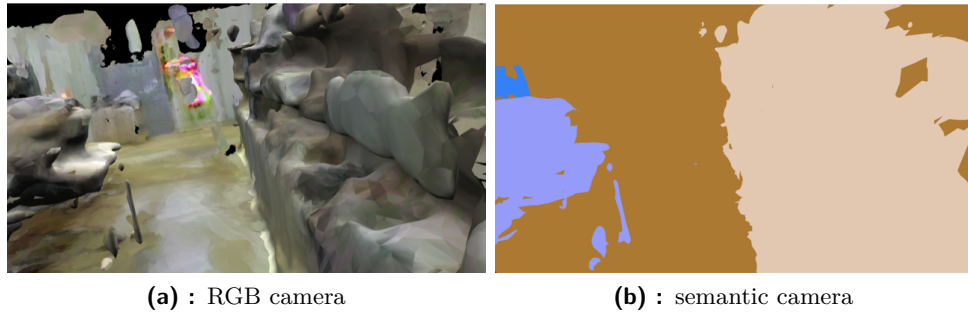


Figure 6.3: RGB frame with its corresponding semantic frame.

■ Generating instance masks from semantic camera

The next step was to generate training data for YOLACT. It is necessary to have RGB frames with corresponding segmentation masks in the COCO dataset format [7]. This has been achieved by placing the agent in the AI habitat into a random navigable point using `env.sim.sample_navigable_point()` and assigning him a semi-random camera rotation. This rotation has been adjusted to not point fully to the ground or the ceiling.

We generated an RGB image and its corresponding instance mask using the semantic image. All the necessary information has been written into a

corresponding .json format in the COCO dataset format.

For both models, 2000 training and 1000 evaluation images have been generated with their corresponding annotations.

■ COCO dataset

COCO dataset is a standard format used in object detection training. It is a dictionary that consists of: "info", "licences", "images", "categories" and "annotations". Each section contains specific information about the given section. For our purpose of training YOLACT, the "info" and "licenses" sections were left out.

Firstly, the "images" dictionary item is a structure containing four values for each image. Those values are "id," which represents the id of the given image, "file_name," which defines the name of the RGB frames file, and "height" and "width," which represent the image dimensions. The image structure, therefore, looks like this:

```
image[n]
├── id
├── file_name
├── height
└── width
```

Secondly, the "categories" dictionary item is a structure containing two values for each category in the list. These values are "id" representing the category id and "name" representing the category name. The structure for each category follows this structure:

```
category[n]
├── id
└── name
```

Lastly, the "annotations" dictionary item contains seven values for each object annotation. The first value is the annotation "id" required to be unique for each new annotation. The second value is the "image_id" depicting the image in which the annotation is. The third value is the "category_id" that describes the category of the object in the given image. The next three values define the segmentation of the object in the picture. Those values are "b_box" representing the bounding box of the segmentation, "segmentation," which is an array defining the exact location of the object, and "area" defining the

area that the object covers in the frame. The bounding box has been located using a simple function written in python (part of `generate_dataset.py` [23]). The segmentation and area were obtained using a `COCOstuffAPI` library. [7].

The structure for each annotation therefore looks like this:

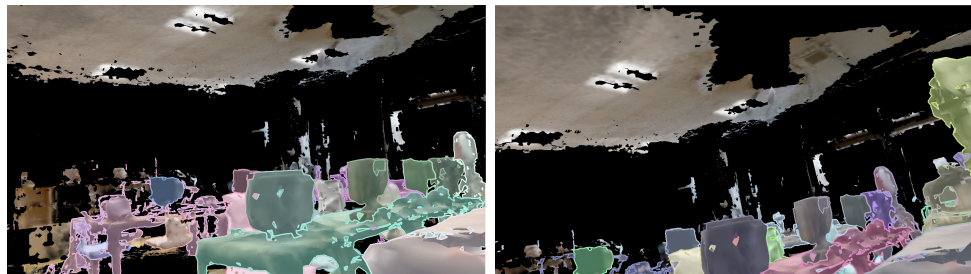
```
annotation[n]
├── id
├── image_id
├── category_id
├── bbox
├── segmentation
├── area
└── iscrowd : 0
```

The full .json annotation file has a following structure:

```
.json
├── images
│   ├── image[n]
│   └── ...
├── categories
│   ├── category[n]
│   └── ...
└── annotations
    ├── annotation[n]
    └── ...
```

This .json has been created using a dictionary format in python with exactly the same structure as the output .json file. After completing the dictionary the .json file was created using `outfile.write(json.dumps(dictionary))`.

Using `COCO_Assistant API` [37] it was possible to visualize these two datasets:



(a) : example 1

(b) : example 2

Figure 6.4: COCO annotations on images from dataset without SPSG.

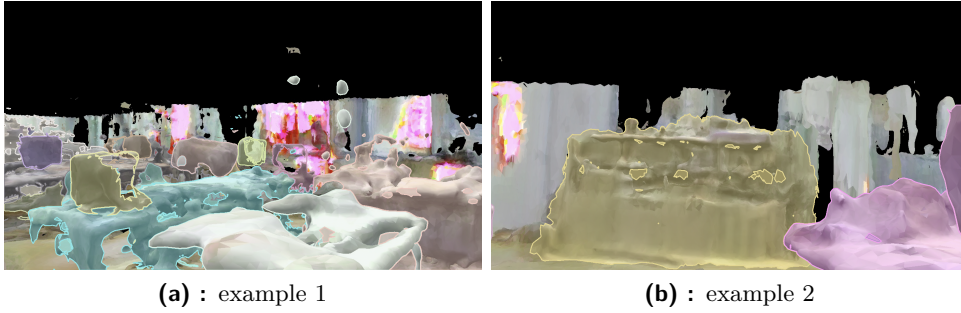


Figure 6.5: COCO annotations on images from dataset with SPSG.

6.1 YOLACT

6.1.1 Training YOLACT

For training object detection, we used YOLACT with two custom datasets. The two custom datasets consist of a set of images and an annotation .json file. It was expected that the dataset made from a scene reconstructed with SPSG would produce better results during the evaluation of the YOLACT training.

To train YOLACT, the software was installed according to installation instructions available on Github [8]. The only necessary step was to modify the config.py file to specify the custom dataset. We specified the paths to test, validation datasets, and corresponding categories for both datasets and a custom config called example_base_config was created using those defined datasets. Examples of custom datasets specifications are written below.

```
spsg_dataset = dataset_base.copy({
    'name': 'SPSG Dataset',
    'train_images': 'data/images/SPSG_train',
    'train_info': 'data/annotations/SPSG_train.json',
    'valid_images': 'data/images/SPSG_val',
    'valid_info': 'data/annotations/SPSG_val.json',

    'has_gt': True,
    'class_names': ("wardrobe", "chair", "table", "PC")
})
```

```
no_spsg_dataset = dataset_base.copy({
    'name': 'NO-SPSG Dataset',
    'train_images': 'data/images/NO_SPSG_train',
    'train_info': 'data/annotations/NO_SPSG_train.json',
    'valid_images': 'data/images/NO_SPSG_val',
    'valid_info': 'data/annotations/NO_SPSG_val.json',

    'has_gt': True,
    'class_names': ("PC", "wardrobe", "unidentifiable",
                    "chair", "robot", "table", "pole" )
})
```

Example of the custom config that has been used can be seen below.

```
example_base_config = coco_artwin_base_config.copy({
    'name': 'no_spsg',
    # Dataset stuff
    'dataset': spsg_dataset, #kuka_env_pybullet_dataset
    'num_classes': len(spsg_dataset.class_names) + 1,
    #The +1 stands for "background" class

    # Image Size
    'max_iter': 10000,
    'lr_steps': lr_steps
})
```

The command for running YOLACT training used is:

```
train.py
--batch_size=2
--config=example_base_config
```

The batch size has been reduced to 2 because of the amount of available memory. Note, that all the masks smaller than 16 pixels (4 width, four height) have to be removed from the dataset. When trying to specify the image size, some other bugs occurred because the size of RGB frames was larger than the recommended 550 (i.e., 756 x 1344). By experimental evaluation, we

discovered that it is better not to specify the image size because YOLACT resizes the frames automatically.

■ 6.1.2 YOLACT evaluation

We trained the YOLACT using 10 000 iterations for each dataset. Three different directories with testing data were created and used during the evaluation. Those directories were images from the SPSG dataset, the non SPSG dataset, and real photos captured with Kinect. Each dataset was evaluated on real data and the corresponding testing images.

The command used for evaluations is:

```
eval.py
--score_threshold=0.15
--trained_model=path/to/trained/model
--images=path/to/test/images:path/to/output
--config=example_base_config
```

The results were obtained in the form of images with masks and bounding boxes. The images below show examples of the output data for each evaluation. The in-depth results are discussed in the next chapter, section 7.4.



Figure 6.6: YOLACT eval.py results on non SPSG dataset and non SPSG images.

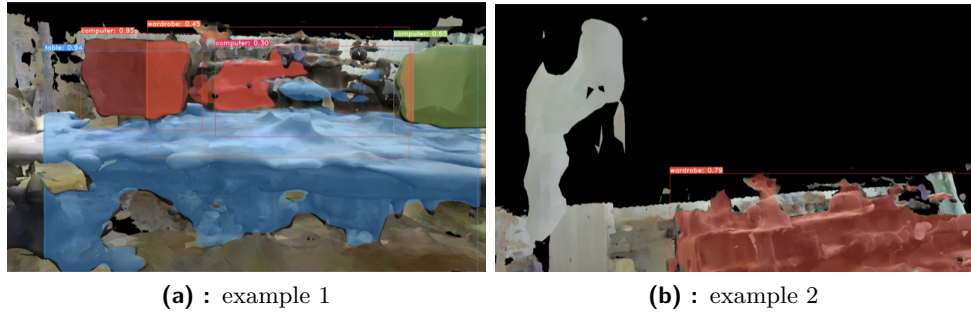
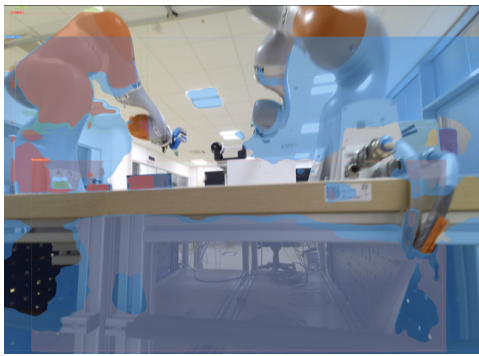
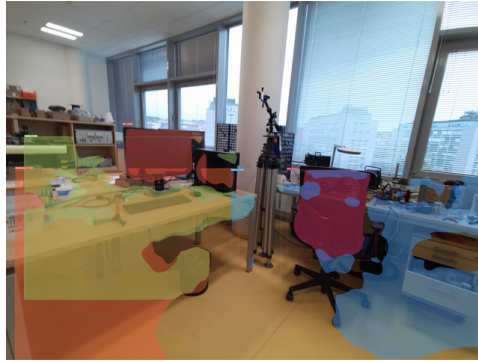


Figure 6.7: YOLACT eval.py results on SPSG dataset and SPSG images.

Figures 6.6 and 6.7 show results on non-real images. We can see that the results are relatively accurate based on the quality of the input dataset. The following two figures show the results on real images, where the accuracy is much lower. We can conclude that object detection does not work on real images from Azure Kinect, because the masks and rendered images do not look realistically.



(a) : example 1



(b) : example 2

Figure 6.8: YOLACT eval.py results on non SPSG dataset and real images.

(a) : example 1



(b) : example 2

Figure 6.9: YOLACT eval.py results on SPSG dataset and real images.



Chapter 7

Result analysis

In the following chapter, we discuss all the experiments, conclude results, and propose improvements. Some strengths and weaknesses of the process are listed, and the standard pipeline without SPSG is compared with the new pipeline (with SPSG) that we proposed as a better alternative.

An important thing to mention is, that the primary goal of the thesis was to evaluate the possibilities of training 2D instance segmentation from the segmented 3D scene. We expected that the process of running all the software will not be that time consuming. Most of the tasks consisted of enhancing and creating multiple codes in order to run the key parts of the thesis. Thus, from a project that was supposed to study a specific topic, it transformed into a more software project.



7.1 Scanning process

First of all, let us discuss the scanning process. The number of scanned images used for reconstruction was sufficient. In the case of having fewer images, the models would be less detailed. On the other hand, having more images might slightly improve the quality of models, but the computing times for reconstruction would increase. We believe that for our purpose, approximately 20 000 images were enough.

Moreover, we employed multiple devices because each device allowed us to enhance the different properties of the 3D model and added a unique feature to the project. By having only one scanning device, we would lose the opportunity to choose the most accurate device or for example lose most of the tracks by not having wide-angle images at the input. This variability helped us choose the most straightforward paths and sped up the process.

In conclusion, the scanning process was successful. Examples of the scanning and processing data can be found at [23].

7.2 COLMAP

Secondly, we discuss the COLMAP reconstruction. COLMAP was a handy software that perfectly reconstructed the scene.

However, one issue was encountered during the process. During the first run of COLMAP we did not specify the argument `-single_camera_per_folder` and it caused some deformations and bad geometry of the scene.

When using the correct parameters, the camera intrinsics and camera extrinsics were relatively accurate and while working with data from COLMAP, it was relatively easy to find all the needed information because of the available documentation for all the parts of the software.

All the data obtained from SPSPG did not contain any errors, and therefore it can be concluded that COLMAP is a reliable software. The COLMAP database might have included some errors and artifacts, but this would have been only a small number of samples used. No actual errors were found during the project.

7.3 SPSPG

The biggest issue we encountered was bounded with SPSPG. First of all, the lack of documentation made it difficult to understand the software and its usage. The article about SPSPG perfectly describes the theoretical part. However, all

the practical information was missing. This caused a considerable delay in the proposed schedule, and it took more than six months to run all the parts necessary correctly.

Secondly, the presented results in the article are not depicting the reality of the reconstructions. After running the software on the data provided by the authors and comparing the results with the results presented, it was surprising to find out that, in reality, the results are not as accurate as presented.

The results are presented in the following figures, compared to our results with the same scenes but looking from different angles and positions. Firstly, let us look at the presented results and the results obtained by running the program.



Figure 7.1: Results presented by SPSG. [6]



Figure 7.2: Our results from the same room.

As it can be seen, the results look very similar, if not the same. The reconstruction appears to work well, filling all the holes and repairing all the inconsistencies. However, this is not true while observing the model from a different position.

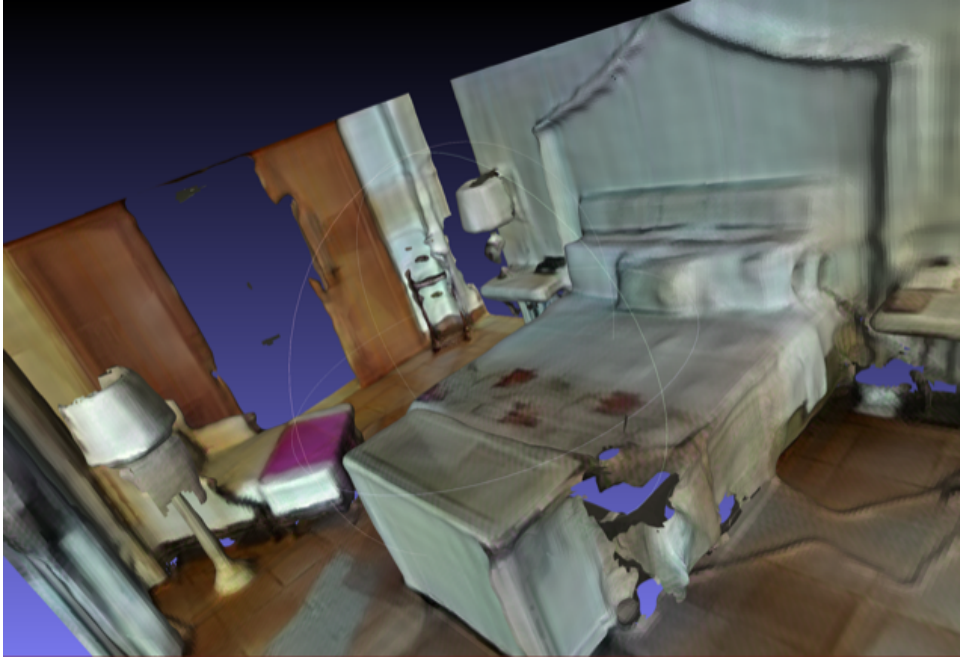


Figure 7.3: Our results from the same room but different position.

We can see that the scan contains new artifacts and holes that were not present in the input scan that was used. We can conclude that the software is not working perfectly as presented at the oral presentation of CVPR 2021.

We initially believed, that the software was not working correctly because of some mistake in our dataset. However, from the comparison of the data presented by the authors and the output of SPSG on this data, we can see the same artifacts and missing parts of the 3D model. Our observation is that the software is not working as expected.

7.4 YOLACT

The results from YOLACT are defined by mean average precision (mAP) [38] for the bounding boxes and the predicted segmentation masks. Firstly let us summarize the results from the dataset without SPSG.

Category	bbox mAP [%]	mask mAP [%]
All categories together	29.58	28.93
Pole	29.22	32.64
Table	30.10	12.62
Robot	19.23	10.24
Unidentifiable	9.04	5.46
Wardrobe	60.28	67.66
PC	26.37	37.32
Chair	32.82	36.58

Table 7.1: mAP results non-SPSG

The results show that the mAP of the bounding box estimation was approximately 30%, and the mask was estimated with mAP of approximately 29%.

Secondly let's introduce the results for the SPSG dataset:

Category	bbox mAP [%]	mask mAP [%]
All categories together	24.81	23.86
Table	21.96	17.96
Wardrobe	54.32	54.92
PC	19.95	19.19
Chair	3.02	3.38

Table 7.2: mAP results SPSG

These results show that the mAP of the correct bounding box and mask is lower than in the non-SPSG model. They are equal to approximately 25% and 24%.

The category with the highest mAP for both bounding boxes and masks was wardrobe. This can be caused by the fact that it was usually the largest object in the model, and therefore it was the easiest to detect.

These results were evaluated using rendered 3D images from the 3D model as the testing images. We can observe that better results are obtained on the images from the AI habitat than on the real images. However, we cannot calculate exactly the mAP for the real images since no segmentation masks (ground truth) were created.

Moreover, our hypothesis that evaluation would have higher accuracy on the SPSG enhanced model did not prove right. This was mainly caused by the SPSG approach and the pretrained model.

7.5 Comparison of non-SPSG pipeline vs. SPSG pipeline

The last note compares the classic SfM pipeline featuring COLMAP and the enhanced pipeline composed of COLMAP and SPSG.

The COLMAP pipeline works well for creating the 3D representations of the scene. However, for object detection training it is not accurate and realistical enough, and it would need another software to improve the overall quality, e.g., IBRNET [39]. Such experiments are, however out of the scope of this thesis.

We expected the SPSG model to be more suitable for semantic segmentation, but it did not prove accurate enough. The SPSG did not meet our expectations, because the quality of both models was almost the same. The only benefit of using SPSG found is that it can close all the holes in the floor of the scan and therefore allow better navigation of agents in AI habitat.

Therefore, there is no real point in using this version of the SPSG.

7.6 Possible improvements

This section discusses the possible approaches that could improve the results of YOLACT evaluation. These ways are purely theoretical, and are planned to be evaluated in future work.

7.6.1 More scanned data

The first suggestion would be to have more images used for the dense reconstruction. Even though the amount of data was sufficient, we should achieve higher precision at the places that were not observed by any device. The best approach would be to see what the models produced and scan the missing inaccurate areas that did not reconstruct perfectly.

On the other hand, having more data would lead to longer processing and computation times. Moreover, the size of all the files we operated with would increase, leading to much more time spent on the project itself. This could be solved by having better computers for all the tasks, but since this project was mostly done on servers with high CPU and GPU, the computation time cannot be optimized much more. Therefore, an advanced keyframe selector would be required to have well spread images with the best possible coverage of the environment.

7.6.2 SPSPG trained model

Another possible improvement is to train the SPSPG model on our data. The SPSPG model used in this thesis was trained on a Matterport 3D dataset composed of images with different resolutions, different camera sensors, and different scanning methods. Higher reconstruction accuracy might have been achieved by training the model on our data.

7.6.3 Number of YOLACT iterations

Thirdly, the number of YOLACT iterations might also improve the results of real-time segmentation. Even though this suggestion is the least efficient one and it should improve the results only slightly.

The downside of this suggestion is that at a certain amount of iterations, the mAPs almost stop increasing. However we cannot say that the increase of those values entirely stops. In Figure 7.4, the trend of increasing mAP vs. the number of iterations can be found. From this graph, we can observe the mAP dependence on the number of iterations.

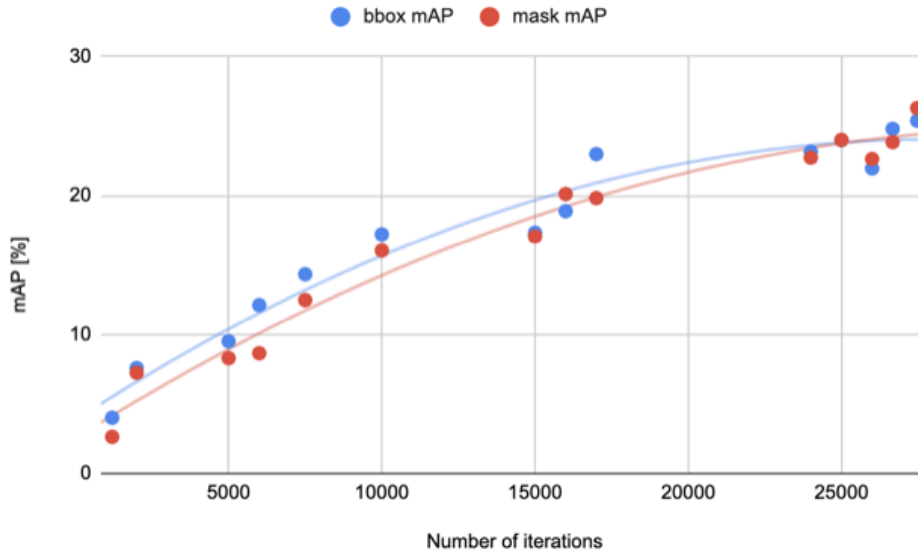


Figure 7.4: mAP vs. number of iterations.

7.6.4 Other improvements

Another possible improvement is the employment of the latest SoTA algorithms for SfM and MVS. Not all of those algorithms are employed in COLMAP. Using this tactic it might be possible to employ pixel-perfect SfM to optimize the 3D model and camera poses.

Another improvement might be the usage of NERF-based methods. That would allow us to generate new images that look realistically and can be used for training. (i.e., usage of IBRNET [38].

Also, we could use a different method for point cloud completion. Especially, we could try to fill the missing parts in-depth maps estimated by MVS, from depth maps captured by ToF cameras. After training such neural network we could then focus on depth maps instead of point clouds.

7.7 Conclusion

The goal of this work was to evaluate the 2D instance segmentation on data generated from 3D models. The biggest challenge is to have a 3D model that is realistic enough to produce sufficient data for instance segmentation training. We tried to optimize the 3D model by employing SPSG, but discovered that it is not really possible.

Training of 2D instance segmentation using YOLACT did not work on real images captured by Azure Kinect. Moreover the scene enhanced by SPSG was still not accurate enough to produce realistically looking data. The hypothesis stating that data from the SPSG model will have higher YOLACT mAPs, did not prove to be correct.



Chapter 8

Code

All codes used in this thesis are attached in file codes.zip. These are also available in Github repository [23]

Appendix A

Bibliography

- [1] Xiaoke Shen. A survey of object classification and detection based on 2d/3d data. *CoRR*, abs/1905.12683, 2019.
- [2] Shuchao Pang, Anan Du, Zhenmei Yu, and Mehmet Orgun. 2d medical image segmentation via learning multi-scale contextual dependencies. *Methods*, 05 2021.
- [3] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. Pixelwise View Selection for Unstructured Multi-View Stereo. In *European Conference on Computer Vision (ECCV)*, 2016.
- [4] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-Motion Revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [5] Tapio Hellman and Mikko Lahti. Photogrammetric 3d modeling for virtual reality, 08 2018.
- [6] Angela Dai, Yawar Siddiqui, Justus Thies, Julien Valentin, and Matthias Nießner. Spsg: Self-supervised photometric scene generation from rgb-d scans. In *Proc. Computer Vision and Pattern Recognition (CVPR)*, *IEEE*, 2021.
- [7] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014.
- [8] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: Real-time instance segmentation. In *ICCV*, 2019.

- [9] Microsoft. Azure-kinect-sensor-sdk. <https://github.com/microsoft/Azure-Kinect-Sensor-SDK>, 2020.
- [10] Microsoft. device-wire. <https://docs.microsoft.com/en-us/azure/kinect-dk/media/resources/hardware-specs-media/device-wire.png>, 2021.
- [11] Microsoft. Hololens2forcv. <https://github.com/microsoft/HoloLens2ForCV>, 2020.
- [12] Microsoft. hololens2-exploded-view-diagram. <https://docs.microsoft.com/en-us/hololens/images/hololens2-exploded-view-diagram.png>, 2021.
- [13] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.
- [14] San Jiang, Wanshou Jiang, and Bingxuan Guo. Leveraging vocabulary tree for simultaneous match pair selection and guided feature matching of uav images. *ISPRS Journal of Photogrammetry and Remote Sensing*, 187:273–293, 2022.
- [15] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Ugo Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.
- [16] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.
- [17] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.
- [18] Patrick Langechuan Liu. Single stage instance segmentation — a review. 2020.
- [19] Manolis Savva*, Abhishek Kadian*, Oleksandr Maksymets*, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.
- [20] Microsoft. camera-fov. <https://docs.microsoft.com/en-us/azure/kinect-dk/media/resources/hardware-specs-media/camera-fov.png>, 2021.
- [21] Dorin Ungureanu, Federica Bogo, Silvano Galliani, Pooja Sama, Xin Duan, Casey Meekhof, Jan Stuhmer, Thomas J. Cashman, Bugra Tekin, Johannes L. Schonberger, Bugra Tekin, Pawel Olszta, and Marc

- Pollefeys. HoloLens 2 Research Mode as a Tool for Computer Vision Research. *arXiv:2008.11239*, 2020.
- [22] OpenCV. Open source computer vision library, 2015.
- [23] Jan Ferbr. Thesis 2022. https://github.com/ferbrjan/thesis_2022, 2022.
- [24] Microsoft. Azure kinect transformation example. <https://github.com/microsoft/Azure-Kinect-Sensor-SDK/tree/develop/examples/transformation>, 2020.
- [25] Microsoft. image-transformation. <https://docs.microsoft.com/en-us/azure/kinect-dk/media/how-to-guides/image-transformation.png>, 2020.
- [26] Microsoft. Hololens2forcv. <https://github.com/microsoft/HoloLens2ForCV/tree/main/Samples/StreamRecorder>, 2020.
- [27] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.
- [28] Angela Dai, Christian Diller, and Matthias Nießner. Sg-nn: Sparse generative neural networks for self-supervised scene completion of rgb-d scans. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2020.
- [29] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D data in indoor environments. *International Conference on 3D Vision (3DV)*, 2017.
- [30] Angela Dai. Spsg. <https://github.com/angeladai/spsg>, 2021.
- [31] Prof. Matthias Nießner. mlib. <https://github.com/niessner/mLib>, 2015.
- [32] M. Hohenwarter, M. Borchers, G. Ancsin, B. Bencze, M. Blossier, A. Delobelle, C. Denizet, J. Éliás, Á Fekete, L. Gál, Z. Konečný, Z. Kovács, S. Lizelfelner, B. Parisse, and G. Sturr. GeoGebra 4.4, December 2013. <http://www.geogebra.org>.
- [33] Tomáš Pajdla. Elements of geometry for robotics. 2021.
- [34] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2017.
- [35] Stanislav Steidl. habitatros. <https://gitlab.com/ssteidl/habitat-ros>, 2021.

- [36] CesiumGS. obj2gltf. <https://github.com/CesiumGS/obj2gltf/commits/0.1.0>, 2015.
- [37] Ashwin Nair. Coco-assistant. <https://github.com/ashnair1/COCO-Assistant>, 2019.
- [38] Paul Henderson and Vittorio Ferrari. End-to-end training of object class detectors for mean average precision. *CoRR*, abs/1607.03476, 2016.
- [39] Qianqian Wang, Zhicheng Wang, Kyle Genova, Pratul P. Srinivasan, Howard Zhou, Jonathan T. Barron, Ricardo Martin-Brualla, Noah Snavely, and Thomas A. Funkhouser. Ibrnet: Learning multi-view image-based rendering. *CoRR*, abs/2102.13090, 2021.