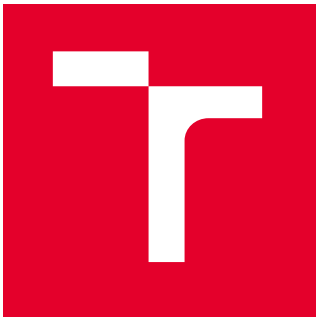

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

3D MODEL REGISTRATION WITH DEPTH CAMERA DATA

REGISTRACE 3D MODELU S DATY Z HLOUBKOVÉ KAMERY

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. Daniel Boháč

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. Roman Adámek

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF MECHANICAL ENGINEERING

Assignment Master's Thesis

Institut:	Institute of Solid Mechanics, Mechatronics and Biomechanics
Student:	**Bc. Daniel Boháč**
Degree programm:	Mechatronics
Branch:	no specialisation
Supervisor:	**Ing. Roman Adámek**
Academic year:	2022/23

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Master's Thesis:

3D model registration with depth camera data

Brief Description:

Depth sensors and cameras are currently widely used in a variety of different industries, from mobile robotics and automation to 3D object scanning. This thesis deals with the challenge of identifying a specific object from depth sensor data based on its known 3D model. And also determine its position and orientation in space. Such data can then be used, for example, to guide a robotic arm to grasp the object.

Master's Thesis goals:

1. Conduct a literature review of the state–of–the–art methods for matching and registration of objects specified by their 3D model and corresponding point cloud.
2. Based on the literature review, select a suitable algorithm that will be able to match a 3D model to data obtained from a depth sensor and determine its position and orientation in space.
3. Validate the result on a set of test objects and assess the functionality and limitations of this method.

Recommended bibliography:

TRUCCO, Emanuele a Alessandro VERRI. Introductory techniques for 3-D computer vision. Upper Saddle River, NJ: Prentice Hall, c1998. ISBN 0132611082.

SOLEM, Jan Erik. Programming computer vision with Python. Sebastopol, CA: O'Reilly, 2012. ISBN 1449316549.

KAEHLER, Adrian a Gary R. BRADSKI. Learning OpenCV 3: computer vision in C++ with the OpenCV library. Sebastopol, CA: O'Reilly Media, [2017]. ISBN 1491937998.

Deadline for submission Master's Thesis is given by the Schedule of the Academic year 2022/23

In Brno,

L. S.

prof. Ing. Jindřich Petruška, CSc. Director of the Institute	doc. Ing. Jiří Hlinka, Ph.D. FME dean

Summary

This thesis deals with 3D object recognition and pose estimation based on depth camera data, specifically point clouds. The only references for potential objects considered are their corresponding 3D models. The goal is to identify an appropriate algorithm through a literature review, develop a solution implementing this algorithm, assess its functionality, and describe its limitations. The work proposes an easily usable extendable library written in C++ and usable within Python. This library incorporates a pipeline derived from the literature review that addresses the problem by utilizing global descriptors. Lastly, the solution is validated in experiments using artificial and real data.

Abstrakt

Tato práce se zaměřuje na úlohu 3D rozpoznání objektu a odhadu jeho transformace na základě dat získaných z hloubkové kamery. Konkrétně jde o mračna bodů, přičemž jedinou referencí pro dané objekty jsou jejich 3D modely. Cílem práce je vybrat vhodný algoritmus na základě provedené rešerše, implementovat řešení využívající tento algoritmus, ověřit jeho funkčnost a identifikovat jeho limity. Práce představuje snadno použitelnou a rozšiřitelnou knihovnu napsanou v C++, která je dostupná také prostřednictvím Pythonu. Tato knihovna využívá postup identifikovaný během rešerše, který řeší danou úlohu pomocí globálních deskriptorů. Nakonec je řešení ověřeno na umělých i reálných datech.

Keywords

3D object recognition, pose estimation, point cloud, 3D model, global descriptor

Klíčová slova

3D rozpoznání objektu, odhad transformace, mračno bodů, 3D model, globální deskriptor

Rozšířený abstrakt

Probíhající automatizace ve všech odvětvích průmyslu si žádá řešení mnoha dílčích výzev. Jednou z těchto výzev je i rozpoznávání objektů a získání informace o jejich přesné poloze a orientaci (transformaci). Řešení této úlohy umožňuje tyto objekty uchytit a přesunout, či založit na přesné místo. Potenciálně lze pak vytvořit univerzální pracoviště vhodné pro vícero objektů, bez nutnosti jeho drahé specializace v oblasti manipulace.

Tuto úlohu je vhodné řešit metodami počítačového vidění. Ty lze rozdělit na dva hlavní proudy: 2D a 3D. Právě druhý zmíněný čelí v poslední době rostoucímu zájmu, mimo jiné, díky stále lepší dostupnosti hloubkových kamer. Tento zájem je navíc podpořen potenciálem robustnějších metod, díky většímu množství informací o scéně (hloubka).

Tato práce se zabývá problematikou 3D rozpoznání objektu z dat hloubkové kamery, konkrétně mračna bodů, a odhadu jeho transformace. Konkrétně je uvažována situace, kdy se objekty nachází volně na posuvném pásu a zároveň je možná přítomnost pouze jednoho objektu ve snímáné scéně v daném okamžiku. Zároveň mají být jedinými referencemi k možným objektům pouze jejich 3D modely, bez textur či informace o barvě.

Nejprve je v práci uvedena uvažovaná situace. Poté jsou popsány typy 3D dat, které jsou relevantní pro řešení dané úlohy. Tudíž jsou uvedeny základní typy reprezentací 3D modelů, následované popisem mračen bodů a způsoby, jak je získat. V případě mračen bodů je důležité zdůraznit, že pokud využíváme jednu scénu zachycenou hloubkovou kamerou, jsou objekty v této scéně reprezentovány pouze takzvaným částečným mračnem bodů. To je způsobeno nedostatkem informací o částech objektu, které nejsou viditelné z daného umístění kamery. Dále je v práci provedena základní rešerše možných řešení dílčích problémů, jako je 3D rozpoznávání objektů, odhad transformace mračna bodů a vhodných nástrojů. Z této rešerše vyplývá, že pro daný scénář je vhodné použít přístupy založené na rysech pro 3D rozpoznávání objektů a přístup hrubého odhadu a jemného dopřesnění pro odhad transformace. Dále je specifikován způsob hodnocení přesnosti odhadu transformace a jeho nedostatky, které jsou způsobeny především existencí nejednoznačných pohledů. Navíc je určena nejvhodnější knihovna pro implementaci řešení, a to Point Cloud Library pro jazyk C++.

Na základě zpřesnění oblastí zájmu, je provedena hlubší rešerše, zaměřující se na postupy využívající zmíněné dva přístupy a srovnávací práce. S ohledem na dosavadní práce provedené v této oblasti, je extrahován obecný postup řešení celé úlohy. Tento postup je založen na využití databáze částečných mračen bodů objektů a globálních deskriptorů. Dále jsou popisovány jednotlivé kroky tohoto postupu a algoritmy, které využívají.

Následně je navrhnuto řešení, které integruje postup uvedený v rešeršní části. Toto řešení je ve formě knihovny, která je navíc modulární a relativně jednoduchá na použití, s širokými možnostmi konfigurace. Umožňuje jednoduché testování, diagnostiku a řešení případných chyb při konfiguraci. Knihovna zajišťuje celý proces, od generování částečných mračen bodů jednotlivých objektů s možností přidání šumu, až po testování úspěšnosti rozpoznávání a přesnosti odhadnutých transformací.

Po představení návrhu následují detaily implementace tohoto řešení. Výsledkem je knihovna napsaná v jazyce C++, skládající se z několika modulů, které umožňují potenciální rozšíření o další postupy. Pro tuto knihovnu byl vytvořen i adaptér, poskytující jednoduché rozhraní pro její použití, který je navíc začleněn do Python modulu. Veškerá konfigurace je předávána ve formě JSON souborů, které lze upravovat v libovolném textovém editoru.

Po implementaci řešení je přistoupeno k jeho ověření prostřednictvím tří druhů experimentů se třemi skupinami objektů. Experimenty byly navrženy tak, aby bylo možné z jejich výsledků obecně posoudit chování vytvořeného řešení. Druhy experimentů proto přesdstavují: ideální generovaná data bez šumu, uměle generovaná data s šumem, a reálně zachycená data. Výsledky ukázují funkčnost řešení i pro data zachycená reálnou hloubkovou kamerou, navzdory značné deformaci geometrických rysů objektu v důsledku charakteristiky šumu použité kamery. V rámci experimentu s reálně zachycenými daty, který zahrnuje tři objekty rozpoznávané v celkem 72 scénách, dosahuje řešení úspěšnosti správného rozpoznání objektu 40,28 %, přičemž u 27,59 % z těchto správných rozpoznání je přesně odhadnuta i transformace objektu. Experiment provedený na uměle generovaných datech s šumem dosahuje daleko lepších výsledků. Při zahrnutí šesti objektů rozpoznávaných v celkem 252 scénách dosáhuje řešení úspěšnosti správného rozpoznání objektu 74,21 %, přičemž u 67,91 % z těchto správných rozpoznání je přesně odhadnuta i transformace objektu.

Na základě výsledků z experimentů je možné identifikovat některé aspekty chování navrženého řešení a identifikovat jeho limity. Úspěšnost řešení se ukazuje být závislá jak na konfiguraci, tak i na konkrétních objektech ve skupinách. Některé objekty vykazují mnohem horší rozpoznatelnost i přesnost odhadů transformace, než objekty jiné. Navíc je zaznamenána možnost nesprávného rozpoznání menšího objektu ve větším, pokud oba disponují velmi podobnými geometrickými rysy. Mezi omezení řešení patří nedostatečná robustnost v odhadu transformace, což je ovlivněno i přítomností nejednoznačných pohledů. Další limitací je nutnost úpravy parametrů vždy dle charakteru daných vstupních dat. V závěru práce jsou uvedena možná vylepšení navrženého řešení a návrhy pro další práce s navazující tématikou.

Bibliographic citation

BOHÁČ, Daniel. *3D model registration with depth camera data* [online]. Brno, 2023. Available from: `https://www.vut.cz/studenti/zav-prace/detail/149546`. Master's Thesis. Brno University of Technology, Faculty of Mechanical Engineering, Institute of Solid Mechanics, Mechatronics and Biomechanics. 88 pages, Supervisor Roman Adámek.

I declare that this thesis has been composed solely by myself under the supervision of Ing. Roman Adámek. All references, literary sources, and other resources I have utilized to develop this work have been appropriately cited and acknowledged.

Bc. Daniel Boháč

Brno, May 26, 2023

I want to acknowledge my thesis supervisor, Ing. Roman Adámek, who has provided valuable guidance and significant motivational support, contributing greatly to the completion of this thesis. In addition, my sincere gratitude goes to my girlfriend for her unwavering support throughout this process. I must also extend my appreciation to my family, whose understanding and encouragement have been vital to my perseverance.

Bc. Daniel Boháč

Contents

1 Introduction

The ongoing automation in manufacturing and related industries demands solutions to various challenges. One of these challenges is recognizing objects and determining their accurate pose. This is vital for automating manipulation tasks, such as picking, transferring, and settling with a robotic manipulator, without costly and highly specialized workstations. While these tasks are intuitively simple for humans, replicating this ease in an automated setup has proven computationally and algorithmically complex. Currently, this problem is being solved thanks to state-of-the-art computer vision solutions in combination with the qualities of modern sensors.

Generally, computer vision solutions can be divided into two main groups: 2D and 3D data-based. The 3D data-based approach shows great potential in object recognition due to the additional dimension of spatial and geometric information of the captured scene. Moreover, the increasing availability of depth cameras stimulates the adoption of 3D data-based computer vision solutions in industrial applications.

To ensure the overall automation process is efficient, adaptable, and flexible, the latency between incorporating a new object into the database and its recognition during production should be negligible. In this regard, Computer Aided Design (CAD) models of parts, generally accessible to manufacturers, can be utilized. Ideally, adding a 3D model to the possible object database would be followed by its correct recognition in the production line. Thus, the factory employing such a solution will be able to adapt workstations quickly, which is demanded in an increasingly dynamic environment of modern manufacturing and associated industries.

This thesis explores possible solutions to the problem in a specific industrial application scenario, where only a single object at a time is located in an area captured by a depth camera. Chapter 2 outlines the problem, specifies the scope of this thesis, itemizes the relevant requirements, introduces relevant data formats, presents introductory research on object recognition and pose estimation methods based on 3D data, and draws relevant conclusions to narrow subsequent chapter. Chapter 3 describes relevant related work, extracts a general pipeline for 3D object recognition and pose estimation, and describes each step regarding this solution pipeline. Chapter 4 proposes a solution without implementation details. Chapter 5 describes the implementation details of the solution. Chapter 6 is dedicated to validating the solution, assessing its functionality, identifying its limitations, and suggesting further work.

2 Base research

This chapter forms the basis for the entire thesis. First, Section 2.1 introduces the details of a specific application scenario, defines and evaluates the problem to be solved, and illustrates the scope of this thesis. Section 2.2 introduces the relevant 3D data representations to be used. Sections 2.3 and 2.4 provide an overview of approaches to solving parts of the problem at hand. Section 2.5 briefly introduces the appropriate libraries available for 3D data processing. Finally, Section 2.6 draws the relevant conclusions for conducting a more specific literature review in the following, more detailed chapter.

2.1 Scenario details and assessment

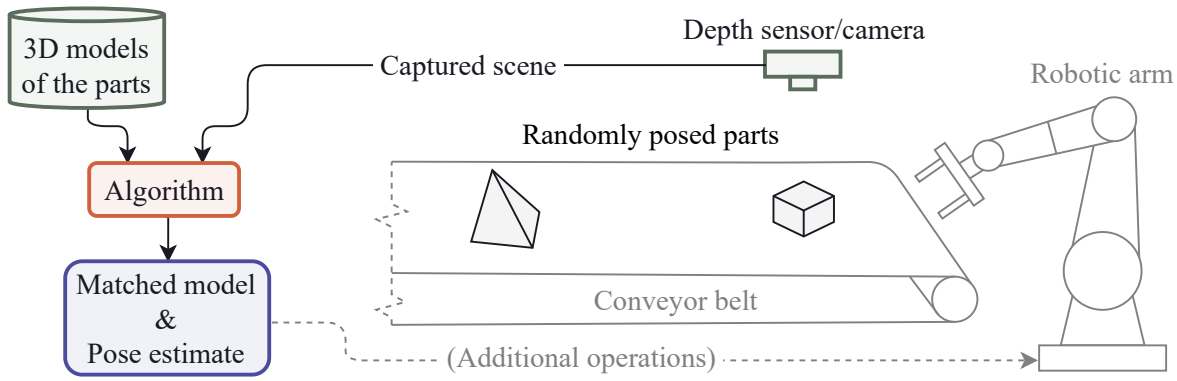

Figure 2.1: Illustration of the problem scenario. The scope of this work is highlighted. Green are the inputs, orange is the main part, and the output is blue.

A real-life workplace scenario inspires the problem addressed in this thesis, an illustration of which can be seen in Figure 2.1. This scenario contains a robotic arm, a conveyor belt, and a depth sensor/camera. The intended task is to transfer parts from the conveyor belt using the robotic arm. These parts are randomly placed with a considerable gap between them and are randomly oriented. Furthermore, assume that the only reference for possible parts apart from the depth sensor data is their CAD models (3D models) and that these models lack texture (and color).

For transferring a specific part, it must be held. However, ideally, its pose (position and orientation) in the space must be known in advance. Thus, when dealing with multiple possible individual parts, the part in the scene must first be recognized. In other words, the corresponding 3D model must match the captured scene. Therefore, the algorithm sequence is as follows: match the 3D model to the captured scene (recognize the object), obtain the pose of the part, and perform additional operations that lead to the transferring of the part.

The scope of this thesis includes only the matching of the captured scene to the 3D model and the obtaining of the pose of the part, not the additional operations or the transfer itself. These two problems are well known in 3D computer vision and are more commonly called 3D object recognition and pose estimation [1]. Because the solution should serve in a robotic perception system, both should take as little time as possible with reasonable accuracy. In addition, the entire solution should be fully automated, ideally including only the initial setup.

Thus, the main tasks to be solved are:

- Match the depth data of the captured scene to the corresponding 3D model (recognize the object).
- Estimate the pose of the matched object relative to the depth sensor/camera and retrieve it.

While the conditions are:

- The captured scene includes only one object at a time.
- The only references for the objects are their 3D models without texture/color.

2.2 3D data representations

In this thesis, two types of 3D data representation are considered. The first is a 3D model, and the second is a point cloud. Currently, it is relatively standard to have CAD models of parts available. However, to use the 3D models designed in CAD software, they must be exported to a universal file format. On the other hand, a point cloud is a common output of most depth sensors and is essential for 3D computer vision.

The following subsection covers the two most widely used universal 3D model file formats. The subsequent subsection provides a closer look at the point cloud representation and its acquisition.

2.2.1 Considered 3D model file formats

The two most widely used universal 3D model file formats are Standard for the Exchange of Product Data (STEP) and STereo Lithography (STL) [2]. The differences between these two file formats are more than formal, as they use two different representations of 3D models. Illustration of these two representations can be seen in Figure 2.2.

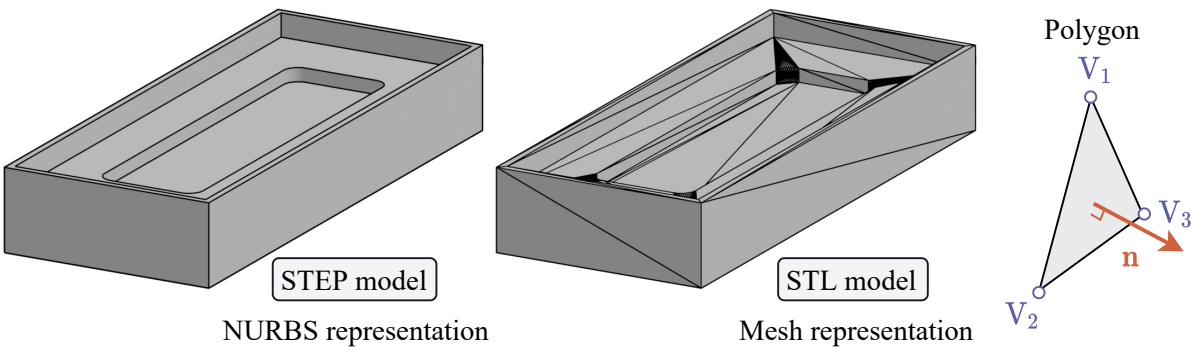

Figure 2.2: Representations of STEP and STL file formats.

The **STEP** file format uses the same mathematical model for geometry description as the 3D CAD software programs[1]. Thus it maintains high precision [2]. However, the nature of the description results in slower rendering and processing. This file format can also include texture, material types, and other product data. It is the preferred file format when geometry modifications are expected or when the high accuracy of the 3D models matters, and slower processing is not an issue.

The **STL** file format uses an approximate triangular mesh (polygons) to describe the surface of the model (see Figure 2.2) [2]. In the description of a whole model, each polygon is defined by a normal vector aim and coordinates of three vertices in counterclockwise order relative to the normal vector (this is referred to as the right-hand rule). The two descriptions of normal orientation may seem redundant, but their mismatch is used to indicate corrupted data [3]. The number of polygons directly depends on the level of detail of curved geometry and corresponds to the file size. This file format cannot store information about the texture or color of the model but has the advantage of having a relatively small footprint. Furthermore, the specific description of the approximated surface enables faster and more straightforward processing than in the case of the STEP file format. Moreover, the STL file format is the most widely supported 3D file format because of its simplicity.

2.2.2 Point clouds and their acquisition

A point cloud is a type of 3D data representation that is described by a set of points in a 3D space, defined by their coordinates [4]. Moreover, each point can also contain color information (RGB values), normal vector direction, and even more additional values. The point cloud representation allows for various processing operations, which forms the basics of 3D computer vision. These operations include, among others, downsampling, filtering, and normal estimation.

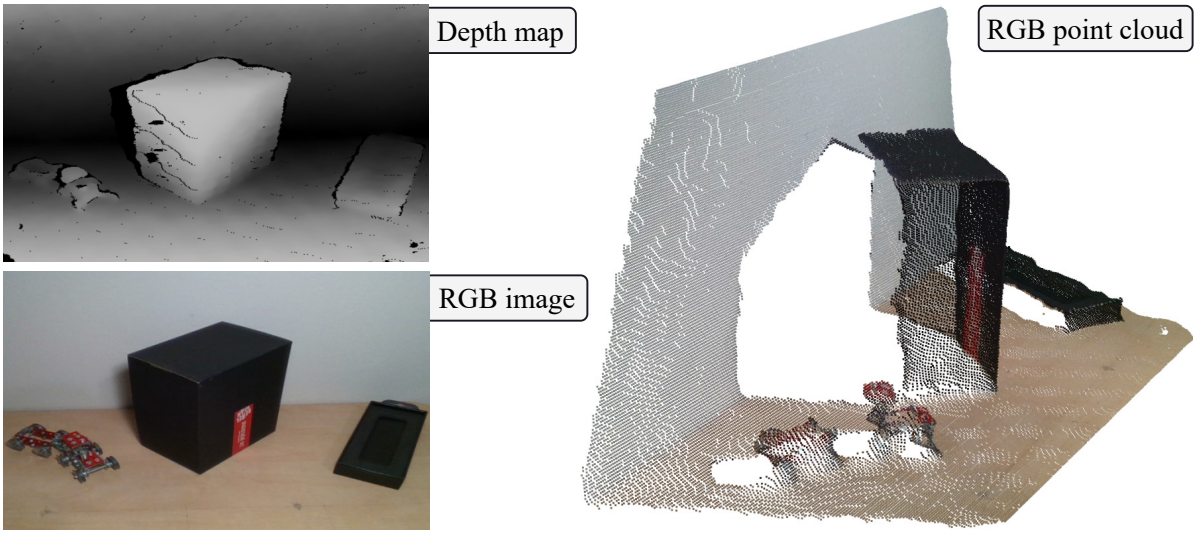

Figure 2.3: Depth map, RGB image, and RGB point cloud captured using an RGB-D camera.

[1]The mathematical model is called Non-Uniform Rational Basis Spline (NURBS).

Currently, three main approaches are used to acquire depth data for 3D computer vision [5]. These are structured light, stereo vision (passive or active), and time of flight. The choice depends mainly on the specifics of the application. However, the most widely used depth sensors are based on structured light or active stereo vision. Both contain standard camera chips and thus are more commonly called depth cameras or 3D cameras. Moreover, both of them use a projection of patterns, which in the case of active stereo vision makes it more robust, and in the case of structured light, is the very essence of the method. Depth cameras that can capture point clouds and fuse them with color information from standard images are also frequently called RGB-D cameras.

Generally, unlike 2D image frames, point clouds are innately disordered [4]. However, a single point cloud frame acquired by a depth camera is typically ordered, similar to a standard 2D image. Ordered point clouds can also be represented as a depth map, an image where each pixel's color encodes the depth information. The possibility of representation through a 2D image implies that the data are, in fact, 2.5D and not entirely 3D. An example of a depth map, RGB image, and a point cloud of the same captured scene using an RGB-D camera can be seen in Figure 2.3. The 2.5D nature of the data means that any object in a single captured frame is represented only by a directly seen portion of it, often referred to as the partial view.

Apart from obtaining point clouds via a depth sensor or camera, it is also possible to simulate these sensors on an artificial scene using appropriate software. Another alternative would be to sample the surfaces of the 3D model with points. However, this does not produce partial view point clouds without special point-discarding techniques.

2.3 3D object recognition

As stated in Section 2.1, the first step towards a solution is recognizing the object based on depth data. Therefore, this section presents an overview of the approaches to 3D object recognition. The main source of information for this section is the systematic review of the literature on 3D object recognition and classification conducted by Carvalho and Wangenheim [6].

Carvalho and Wangenheim analyzed works published between 2006 and 2016, including only articles that describe techniques in their abstracts and compressed them into an overview of 3D object recognition approaches. They subdivided the approaches by the type of representation used for the recognition itself. The three most commonly used types of representation were feature-based, model-based, and view-based. In their work, 68 % of the reviewed works used a feature-based representation, compared to 8 % and 7 %, which used a model-based representation and a view-based representation. Therefore, methods that employ feature-based representation are possibly the most appropriate choice for the solution in this thesis.

The subdivision of approaches by type of representation does not differ between hand-made and learning-based methods. Learning-based methods are gaining more popularity, especially for more complex scenarios. Such scenarios are usually heavily cluttered, and thus mutual occlusion of objects occurs. In such scenarios, learning-based methods generally perform better than hand-made methods [7]. However, learning-based methods require considerable high-quality training data, and acquiring and labeling such data can be quite time-consuming. Therefore, hand-made methods should still be considered, especially for simpler scenarios where mutual occlusion of the objects is not expected.

2.3.1 General pipeline

Regardless of the type of representation, the authors of [6] described a general pipeline for 3D object detection based on their literature review. This pipeline can be seen in Figure 2.4, and its steps are briefly described below.

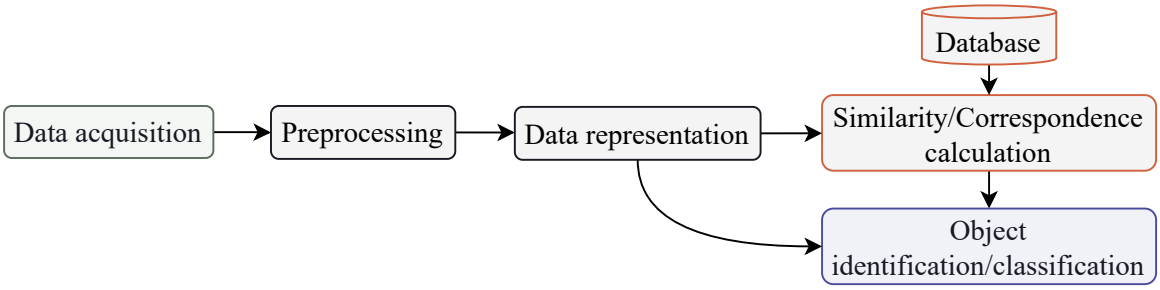

Figure 2.4: General pipeline for 3D object recognition and classification. Redrawn from [6].

The first step in the pipeline is data acquisition. In addition to self-obtaining actual data, freely available datasets or synthetic data can also be utilized. The preprocessing step usually contains multiple operations. These include filtering for noise removal and better sampling, selecting the region of interest, segmenting to separate representations of possible objects, and possibly normalizing the data. The data representation step then utilizes a specified data representation to describe the input data. Next, the similarity/correspondence calculation step compares the chosen data representation with object representations from a database. Thus, the database must be generated in advance. Alternatively, a trained classifier (using a neural network, deep learning model, or fuzzy model) can replace the database and the similarity/correspondence calculation steps. Finally, the object identification/classification step chooses the best candidate based on a specified metric.

2.3.2 Main feature-based representations

The feature-based representations for 3D object recognition can be mainly subdivided into local and global features [6]. Some works combine both but are not considered in this thesis to preserve a relevant level of simplicity. Every 3D object (or point cloud) has features that can be utilized for its description, for example, shape, edges, or distribution of normal vectors. Therefore, these features can form a description of an individual object, called a descriptor. The descriptor is then used for comparison and search of the corresponding object.

Local features are the most widely used feature-based representation [6]. The term local feature means that it describes only part of the object. Thus, local descriptors are computed in the neighborhood of a specified point of an object. Object recognition methods using local descriptors can better differentiate between objects with similar overall shapes. However, the computation of such descriptors for all points representing the entire object tends to be slow. Therefore, keypoint detectors are commonly used [8]. Then, every object can be described by a set of local descriptors. Due to the structure of this description, sophisticated matching methods, such as a voting scheme, need to be employed.

Global features are the second most widely used feature-based representation [6]. The term global feature implies that one description describes the entire object. Thus, object recognition methods that use global descriptors are usually computed more efficiently than local ones. However, global descriptors are not as discriminative and can pose problems distinguishing two objects with only minor differences. The computation of global descriptors on real data should be preceded by a segmentation of the scene, which can be particularly difficult in more complex scenarios [8]. Because of a single description of the whole object, the matching stage tends to be simpler and faster than in the case of local descriptors.

2.3.3 Relevant datasets for the considered scenario

Multiple datasets are available for download online, especially for algorithm testing. The type of dataset content ranges from 2D images over 3D models to point clouds acquired by RGB-D cameras. Many of these datasets are mentioned in [6] and [7][2]. However, most of them are unsuitable for the considered scenario (see Section 2.1) since they are often aimed at complex objects, occluded scenes, and textured objects. In contrast, industrial objects (and especially their 3D models) commonly lack color/texture. Therefore, the most relevant datasets contain relatively simple 3D models, which are usable in simulations or could be 3D printed. Suitable datasets are the ABC dataset [9] and the DeepCAD dataset [10]. An example of models from the DeepCAD dataset can be seen in Figure 2.5. Both collections allow export to various file formats, including STEP and STL (see Subsection 2.2.1).

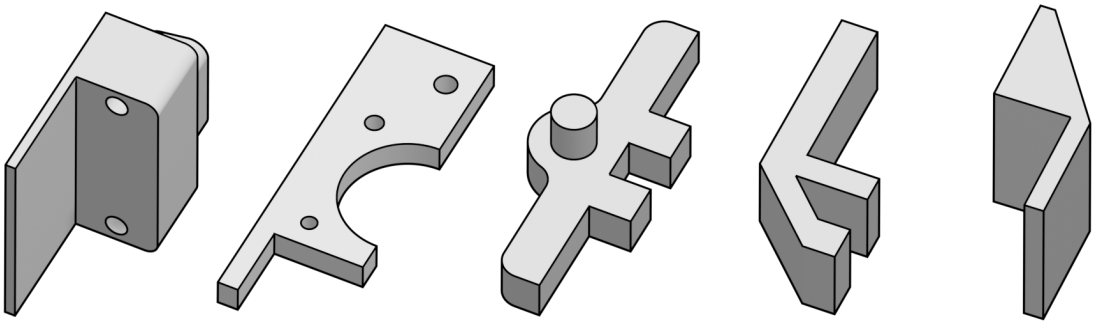

Figure 2.5: Example of objects from the DeepCAD dataset [10].

2.4 Pose estimation

As stated in Section 2.1, the second step towards a solution is the pose estimation of the recognized object. Therefore, this section introduces the task, the general approach for depth data, the problem of ambiguous views, and a method for evaluating pose estimates.

An object in a 3D space has six degrees of freedom, three translational and three rotational. If the object has its local Coordinate System (CS) constructed, its relationship to a reference CS can be described by a transformation matrix $\mathbf{T}$ (4×4) [11]. Usually, the camera CS is the reference CS. However, any known point and orientation in the space can be chosen as the reference CS instead.

[2]Or can be found on related website `https://bop.felk.cvut.cz/datasets/`.

The homogeneous transformation matrix is defined as

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}, \tag{2.1}$$

where $\mathbf{R}$ (3×3) is a rotation matrix, $\mathbf{t}$ (3×1) is a translation vector and $\mathbf{0}$ (1×3) is a vector of zeros. Such a matrix converts data from one CS to another [12]. Thus, it fully describes the pose of an object relative to the camera/reference CS. Furthermore, multiple matrices can be chained together (and be inverse) to create additional CSs or to manipulate data.

Pose estimation based on point clouds is called the 3D registration problem. In the 3D registration problem, one point cloud is labeled as the source and the other as the target. Then, the solution deals with searching for a spatial transformation that aligns the source data with the target data. An illustration of this problem can be seen in Figure 2.6.

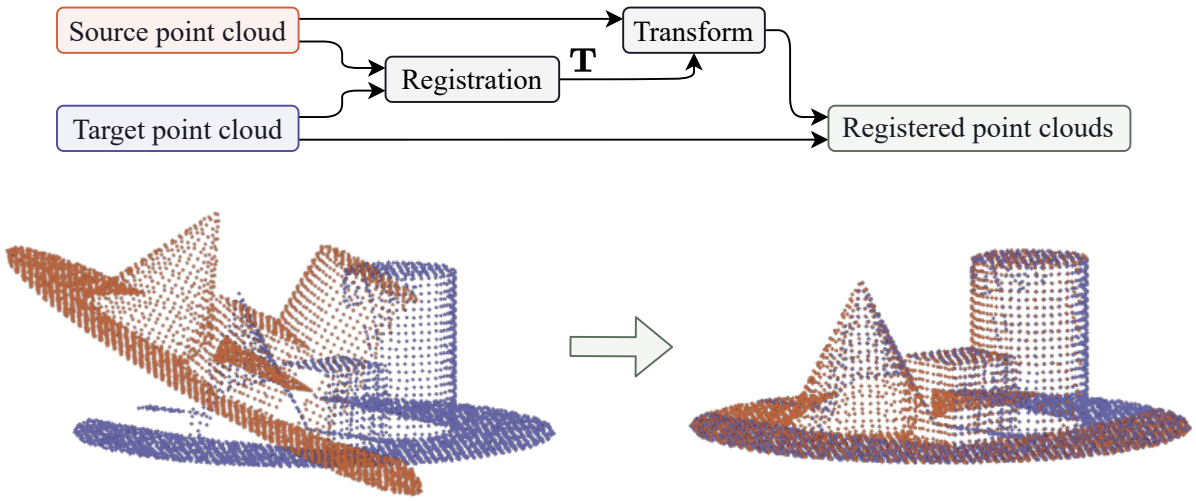

Figure 2.6: Illustration of 3D registration of two point clouds.

The most used methods for solving the 3D registration problem are iterative [4]. They are based on the principle of minimization of the distances between points (or surfaces). However, these methods can fall into local minima, resulting in a wrong alignment when a poor initial guess is provided. Furthermore, iterative methods tend to be computationally heavy. Because of these facts, a coarse-to-fine alignment approach is often employed. The idea of this approach is to provide a decent initial guess (coarse alignment) for an iterative method (fine alignment) to ideally reach global minima (accurate pose of an object).

2.4.1 Ambiguous views and pose estimate evaluation

Symmetries characterize many, even non-industrial, objects. Unfortunately, this property poses a problem within the estimation of poses due to the existence of ambiguous views [13]. Moreover, a potentially considerable number of these views can be present when a single fixed depth sensor (or standard camera) is used because of self-occlusion. Illustrations of self-occlusion and symmetric parts can be seen in Figure 2.7. The most un-

derstandable example of self-occlusion is possibly a cup, where from a considerable range of viewpoints, the exact pose of the cup cannot be identified, as from these viewpoints, the handle, which determines the pose of the cup, cannot be seen. The descriptions computed on partial views from these viewpoints would be identical, thus possibly resulting in a significant pose error in rotation. This error should always be considered, especially in the end applications which involve gripping the object.

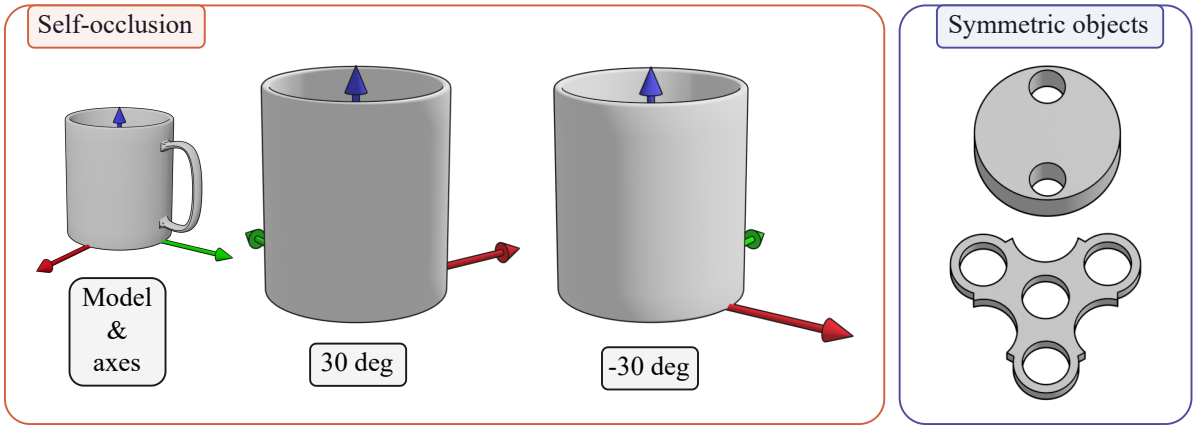

Figure 2.7: Illustrations of self-occlusion on a cup model and symmetric parts.

The pose estimate can only be evaluated when the ground truth pose is known. This condition is met only in the case of algorithm testing, either in a real-measured scene or in a simulation. One of the simplest and most common pose error functions is the translation and rotational error [13]. Let $\mathbf{P}_{\text{est}} = (\mathbf{R}_{\text{est}}, \mathbf{t}_{\text{est}})$ be the estimated pose, and let $\mathbf{P}_{\text{gt}} = (\mathbf{R}_{\text{gt}}, \mathbf{t}_{\text{gt}})$ be the ground truth. Then the translation error is

$$\varepsilon_{\text{trans}}(\mathbf{t}_{\text{est}}, \mathbf{t}_{\text{gt}}) = \|\mathbf{t}_{\text{gt}} - \mathbf{t}_{\text{est}}\|_2, \tag{2.2}$$

and rotation error is

$$\varepsilon_{\text{rot}}(\mathbf{R}_{\text{est}}, \mathbf{R}_{\text{gt}}) = \arccos\left(\frac{\text{Tr}(\mathbf{R}_{\text{gt}}\mathbf{R}_{\text{est}}^{\intercal}) - 1}{2}\right), \tag{2.3}$$

where $\mathbf{t}$ is the translation vector, $\mathbf{R}$ is the rotation matrix, and Tr is the sum of elements on the main diagonal (trace) of a matrix. The translation error is a simple distance (L2 norm) in the 3D space. The rotation error expresses an angle specified by an axis-angle representation of rotation. This angle is the smallest angle that aligns $\mathbf{R}_{\text{est}}$ with $\mathbf{R}_{\text{gt}}$, which ranges from zero to π rad [14]. The axis-angle rotation representation also specifies an axis that can be obtained (in the form of a vector) but is not needed for rotational accuracy evaluation. These errors do not assume ambiguous views, which may be considered when evaluating pose estimates on data that were taken from a single depth camera.

2.5 Available libraries for depth data processing

Several open-source libraries are available for 3D data processing. This section briefly explores the three most appropriate for use in the solution.

OpenCV is the most well-known library for computer vision. Its main aim is 2D computer vision, but it also includes a few modules for depth data processing. However, there are only a handful of options for working with point clouds, and several basic functions are not implemented. This could be changed in the future due to the increasing popularity of 3D computer vision. Although its primary programming language is C++, wrappers to other languages like Python and Java are also available. [15]

Open3D is a newer (2018) library aiming at 3D computer vision. Although it has excellent documentation and implements all basic and numerous registration functions, it does not feature many functions relevant to object recognition. This library supports C++ and Python programming languages. [16]

Point Cloud Library (PCL) is a large-scale library for point cloud and 3D model processing. It has many modules, including filtering, feature estimation, key point search, registration, segmentation, recognition, and more. Many state-of-the-art methods are implemented in these modules, making PCL the best choice for developing applications based on depth data, especially point clouds. Furthermore, its *Tools* module enables the user to perform file conversions, virtual scanning, visualizations, and many more, directly from the command line. This library officially supports only the C++ programming language, as there are currently no official wrappers for other programming languages. [17]

2.6 Main conclusions of the chapter

At the end of this chapter, it is appropriate to draw a few conclusions to narrow the objectives for the subsequent chapters. The conclusions are as follows.

- Section 2.3 provided an overview of 3D object recognition approaches and suggested that the most suitable approach is to use methods using feature-based representations. Moreover, because of the simplicity of the scene and the solution, learning-based approaches can be omitted in favor of hand-made ones.
- Section 2.4 introduced the 3D registration problem related to pose estimation and suggested the coarse-to-fine alignment approach to estimate the pose of an object efficiently. It also defined metrics to evaluate the accuracy of the pose estimation, revealing its possible deficiencies due to ambiguous views caused by symmetries and self-occlusions.
- Section 2.5 specified the PCL as the most suitable choice of the depth data processing library. This choice is made due to the alternative libraries not providing such a wide selection of state-of-the-art methods, especially for 3D object recognition. This also implies that the solution will be written in C++ language.

3 Algorithm research

This chapter builds on the previous one and, among others, provides a detailed view of the whole algorithm. First, Section 3.1 introduces and assesses related work. Subsection 3.1.4 is especially important, as it shows a specific pipeline, which serves as a template for the entire solution and even the rest of this chapter. The following sections describe the individual parts of this pipeline, which are divided into offline and online stages. Section 3.2 introduces common steps in both stages. Section 3.3 describes specifically selected feature descriptors. Sections 3.4 and 3.5 provide an explanation of the steps in the offline and online stages, respectively. Finally, Section 3.6 mentions PCL dependencies that are significant for the implementation of the solution.

3.1 Related work

The papers mentioned in the following sections are the most relevant to the specific scenario (see Section 2.1) while keeping in mind the main conclusions from the previous chapter (see Section 2.6). Thus, the focus is on feature-based 3D object recognition methods and the coarse-to-fine approach to pose estimation while omitting learning-based methods to avoid manual data labeling. This section is structured as follows. First, the PCL object recognition and pose estimation pipelines are briefly described to understand other related work. Then, related comparison works are summarized. Subsequently, other relevant related works are also mentioned. Finally, the current state is assessed, and the pipeline is extracted based on the mentioned related works.

3.1.1 Point Cloud Library recognition pipelines

This subsection describes the pipelines presented in [8] because they are a great example of a 3D object recognition and pose estimation pipelines that use hand-made feature-based representations. Moreover, they also utilize a coarse-to-fine approach to pose estimation, and the whole work considers implementation using the PCL. Therefore, it is the ideal template for the reader to create a picture of the possible solution, which is then modified by related work in the following subsections.

Aldoma et al. [8] presented two recognition pipelines and implied their implementation using the PCL. The pipelines deal with 3D object recognition and pose estimation while assuming CAD models or 3D meshes as object references. One pipeline is aimed at local descriptors and the other at global descriptors (see Subsection 2.3.2 for basic information on descriptors). These pipelines follow similar steps as shown in Subsection 2.4. Thus, both consist of offline and online stages. First, an object description database is created based on its reference models in the offline stage. Then, in the online stage, the description of the scene is matched with the description from the database. However, specific steps vary between the two pipelines, as shown in Figure 3.1.

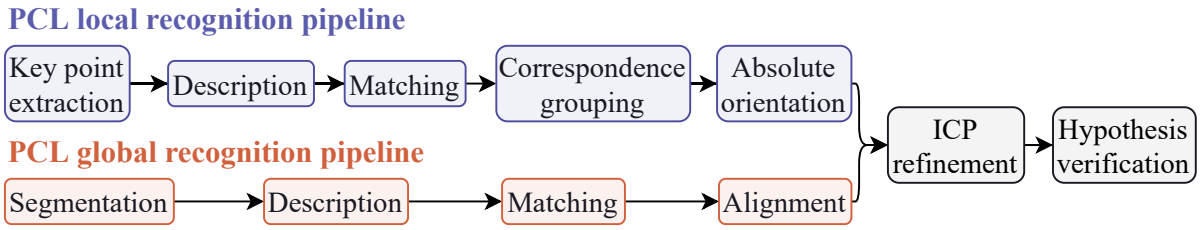

Figure 3.1: Point Cloud Library recognition pipelines. Redrawn from [8].

The creation of a database (offline stage) described in [8] involves using a virtual camera placed at 80 positions around the mesh model to generate partial view point clouds of the objects. Such positions are defined using a subdivided icosahedron, which leads to uniformly placed positions on a sphere of defined radius. Then, on the generated partial views, chosen descriptors are computed, and the results are saved into the database together with transformations between the virtual camera CS and the object CS. Generating partial views is necessary for global descriptors, as they are computed based on all provided points. On the other hand, creating a database based on the full sampled model is possible when using local descriptors. Such a point cloud can also be created by joining partial views and resampling overlapping points [18].

The first step in the local recognition pipeline is key point extraction, where each key point must be both repeatable and distinctive. Then, at each key point, the chosen local descriptor is computed. Next, in the matching step, each scene descriptor is matched against all descriptors in the database to find correspondences. Thereafter, the correspondences per object are grouped based on geometric consistency. Finally, the clusters are reduced by applying the RAndom SAmple Consensus (RANSAC) algorithm, discarding those not of the same pose.

The first step in the global recognition pipeline is to segment the scene point cloud to retain only points related to the object. This step could also enhance the local recognition pipeline, although it is not mandatory for simple scenes. Then a global descriptor is computed, which usually results in a histogram (or, in some cases, multiple histograms). The histogram is then compared with the database using Nearest-Neighbors Search (NNS) in the matching stage. Due to the global descriptors being often invariant to camera roll angle, an additional descriptor named the Camera Roll Histogram (CRH) can also be used to get a more accurate initial pose estimate.

The estimated pose from both pipelines could be categorized as a coarse alignment. Thus, two more steps are employed to improve the results. First, the pose refinement (fine alignment) can be done using the Iterative Closest Point (ICP) algorithm, as the coarse alignment should provide a reasonable initial guess for it to converge correctly. Then, in the second additional step, the hypothesis verification algorithm evaluates the finely aligned result based on inlier and outlier points based on reference. This step addresses the need for discarding false positives and possibly results in a correctly recognized object and its accurate pose. Hypothesis verification also enables the evaluation of more than one result candidate, although the tradeoff is a longer computation time.

3.1.2 Relevant comparison works

Han et al. [19] reviewed 3D point cloud descriptors, including global, local, and hybrid variants. First, the authors described various available descriptors and summarized their main characteristics in a table. Then, they conducted several experiments using 13 selected descriptors, eight of which were local and five global. A dataset was used as input data, where each frame contained only one object and no clutter. Four global descriptors outperformed all selected local descriptors based on the recognition accuracy results. These four global descriptors were (from best to worst): Ensemble of Shape Functions (ESF), Viewpoint Feature Histogram (VFH), Oriented Unique and Repeatable Clustered Viewpoint Feature Histogram (OUR-CVFH), and Clustered Viewpoint Feature Histogram (CVFH). Moreover, the authors suggested that these descriptors are suitable for real-time object recognition applications.

Himri et al. [20] surveyed global descriptors for 3D object recognition in an underwater environment. The authors chose global descriptors over local descriptors because of their (generally lower) computational cost. They conducted real and simulated experiments using seven different global descriptors on a simple scene with one object in it and without clutter. The descriptor database was created based on only 12 views of each object. From the results, the best-performing descriptors were CVFH and OUR-CVFH, while the Global Orthographic Object Descriptor (GOOD) performed best on noisy data. The authors note that the database and the scene should have the same resolution for the best recognition rate.

Li et al. [21] evaluated two main variants of ICP: point-to-point and point-to-plane. The authors performed experiments to assess the validity, robustness, precision, and efficiency. Point-to-point ICP was more valid and robust in most cases based on the results of the experiments. However, when Gaussian noise was present, the point-to-plane ICP was more robust and accurate, while also significantly fewer iterations were needed to converge.

The considered scenario is similar to a more known bin-picking scenario. However, there is no conveyor belt or considerable gap between parts in such a scenario. Instead, all parts are randomly stacked in a bin, cluttering the scene. This type of scene then requires more complex processing due to the amount of clutter and occlusion. Recent works dealing with the bin-picking scenario in a non-learning-based manner [22, 23, 24, 25] utilize the Point Pair Feature (PPF) method [26] or its enhanced versions. Although the PPF method is arguably the best hand-made method for bin-picking and 3D object recognition in complex scenes [27, 7], it may be unnecessarily complex for a simple scenario as the one considered in this thesis.

3.1.3 Other relevant works

Li et al. [28] investigated 3D object recognition and pose estimation for a random bin-picking scenario. The authors dealt with this task using global descriptors. They used the VFH descriptor for recognition and coarse pose estimate, refined the pose estimate using ICP, and finally verified the hypothesis. The authors also sufficiently described the entire procedure, from database creation to post-processing.

Liang and Cheng [29] dealt with RGB-D camera-based 3D pose estimation of parts and the grasping of these parts. They utilized a coarse-to-fine approach using the VFH descriptor for the initial pose estimate and the ICP algorithm for pose refinement. The approach

was based on the evaluation of multiple candidates and employed an offset removal procedure to improve the use of VFH. The authors also conducted a real-life experiment on a scene with a known object sitting on a table, which according to their results, had a high success rate. Thus, based on their experiment, using the global descriptor method for such a task proved to be an effective choice.

Hu et al. [30] investigated a fast pose estimation of shell parts in a robotic assembly scenario. The challenge of estimating the pose of shell parts lies in the similarity of the inner and outer shapes of the part. The authors dealt with pose estimation using a coarse-to-fine approach. The initial (coarse) pose was estimated using a Principal Component Analysis (PCA) algorithm and an image template matching strategy for symmetric objects. Furthermore, an additional initial pose correction strategy and translational offset were also used to avoid local minima when refining the pose estimate. The refinement step itself used a weighted point-to-plane ICP algorithm. Overall, this approach resulted in an accurate and reasonably fast pose estimation of the shell parts.

3.1.4 Assessment of current state

Based on the work mentioned above in this chapter, the most appropriate approach to solving the problem, introduced in Section 2.1, is to use global descriptors for object recognition and coarse pose estimation and a point-to-plane ICP algorithm for pose refinement, mainly due to the simplicity of the considered scenario. Furthermore, the best-suited global descriptors to use are VFH, CVFH, OUR-CVFH, ESF, and GOOD. Except for GOOD, all other mentioned descriptors are implemented in the PCL itself [17]. However, the GOOD has source code available online [31].

Based on the work mentioned above, the pipeline for 3D object recognition and pose estimation in Figure 3.2 was extracted. It assumes 3D models as the only source reference for possible objects and the use of global descriptors. All individual steps are described in detail in the remainder of this chapter.

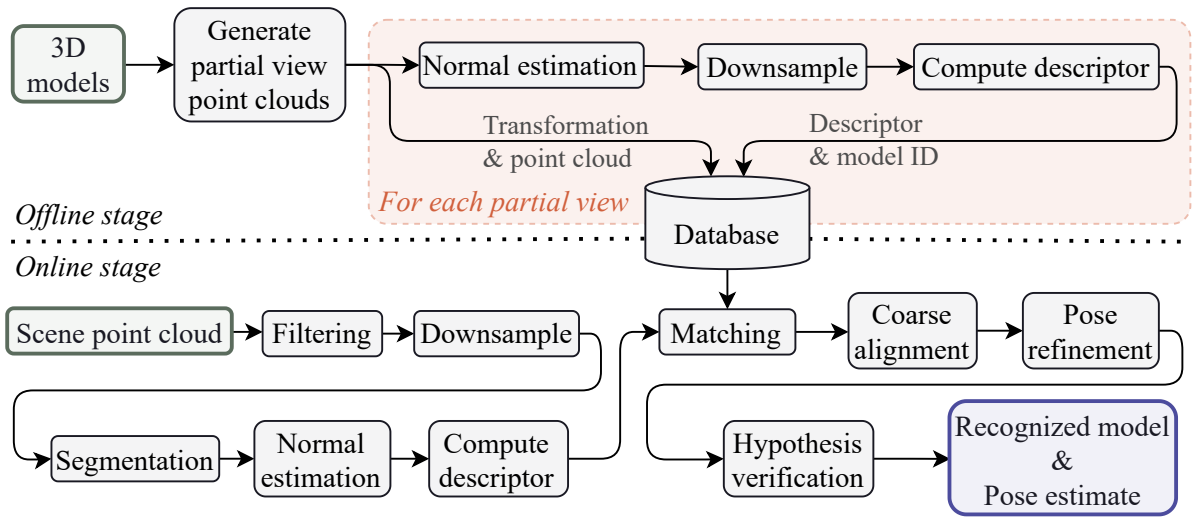

Figure 3.2: Detailed global pipeline based on related work.

3.2 Common steps for both stages

The offline and online steps of the pipeline (see Figure 3.2) share common steps. These are the downsample and the normal estimation. The first mentioned lowers the density of a given point cloud, making subsequent operations more efficient. The second estimates the normal vector of each point in the given point cloud, which many descriptors and alignment methods require. These operations are described in more detail below.

3.2.1 Downsampling

Downsampling primarily reduces the point count in a given point cloud. Reducing the number of points is essential when dealing with point clouds acquired using depth cameras/sensors. Such point clouds usually have hundreds of thousands of points per capture frame, which can lead to computationally heavy processing when considering all the captured data. Moreover, downsampling simultaneously suppresses details and thus can serve as a noise filter.

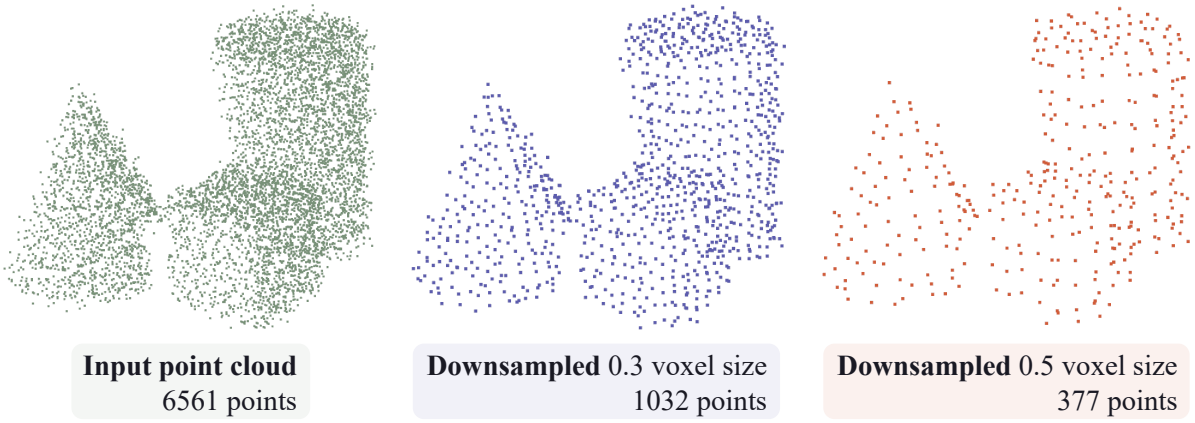

Figure 3.3: Example of point cloud downsampling using a voxel grid. The units of the voxel size values are relative to the point cloud CS units.

The most widely used method for downsampling is using a voxel grid [4]. A voxel can be defined as a cube of fixed dimensions in a 3D space. Thus, a voxel grid is a grid of 3D cubes of defined size. Individual voxels are sometimes also referred to as cells. Downsampling represents all points in a cell by (usually) a single point. There are three main approaches to specifying the coordinates of the point representing the cell: random selection, selecting the center of the cell, and computing the centroid of all points inside the cell. The basic implementation of the voxel grid in the PCL uses the centroid approach because of its more accurate surface representation [17]. However, it is slightly slower than the other two methods. When using a voxel grid as a noise filter, the voxel size should be appropriately selected to suppress artifacts, such as multiple layers representing the surface. For an illustration of point cloud downsampling using a voxel grid, see Figure 3.3.

3.2.2 Normal estimation

Many global descriptors and some variants of the ICP algorithm utilize point clouds with normals. However, such information is usually not present in raw point clouds acquired by a depth camera/sensor and thus must be estimated. This estimation is commonly used

for each point in the input point cloud by fitting a plane to the neighborhood points [4]. The neighbor points are defined by a fixed radius or k nearest points around the primary point. In the PCL, the basic implementation is based on the least-squares plane fitting and viewpoint location [17]. Thus, incorrectly oriented normals are flipped toward the viewpoint. Moreover, a parallelized version of normal estimation is available in the PCL, significantly reducing computation time. Generally, a normal estimation based on dense point clouds tends to be more accurate but at the price of a longer computation time. Thus, it can make more sense to downsample the point cloud in advance and set the radius for normal estimation accordingly. For an illustration of a normal estimation, see Figure 3.4.

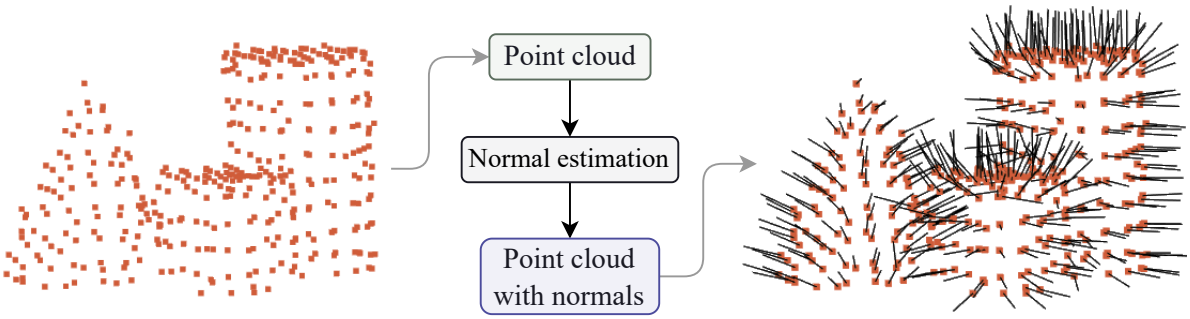

Figure 3.4: Example of input and output of point cloud normal estimation. Black lines represent the normal vectors of each point.

3.3 Selected global descriptors

Point cloud descriptors and their matching form the core of the 3D object recognition and pose estimation pipeline. The choice of a descriptor determines the robustness and speed of object recognition itself. In this section, the principles of the five specific global descriptors, which were selected based on related work (see Section 3.1), are briefly described. A point cloud is sometimes referred to as a cluster, especially where it represents a partial view of an object.

3.3.1 Viewpoint Feature Histogram and improved variants

Before diving into the Viewpoint Feature Histogram (VFH) description, the local descriptor, Fast Point Feature Histogram (FPFH), which the VFH internally utilizes, must first be introduced. FPFH describes three relative angles between the normal vectors of a pair of points. These pairs of points are defined between a selected point and its k nearest neighbors [8]. The results for all possible point pairs are then binned into a histogram, which forms the FPFH.

Furthermore, an additional extension is part of the PCL implementation of the VFH and dependent variants, which is named the Shape Distribution Component (SDC) [32]. The SDC encodes the distribution of points in the cluster related to its centroid. This should help to differentiate between objects with similar characteristics.

Viewpoint Feature Histogram (VFH) proposed in [33] consists of two components, one representing the viewpoint direction and the other representing the surface shape. The viewpoint component is computed as follows. First, the centroid of the cluster is

computed. Subsequently, a vector originating from the point of view and terminating at the centroid point is constructed. Then, this vector is normalized and translated to each cluster point. Next, the relative angle between the said vector and the normal vector of each point is computed. The results are then binned into a histogram. The surface shape component computes the FPFH for the centroid point, while the points considered for point pairs are extended to the whole cluster. These results are also binned into a histogram. Implementation of this descriptor in the PCL defaults to 308 bins. Where 128 bins belong to the viewpoint direction component, 135 to the surface shape component, and the remaining 45 bins are used for the SDC as mentioned at the beginning of this section [32]. As mentioned in [19], the VFH is computationally efficient. However, it is sensitive to noise and occlusion and is invariant to camera roll.

Clustered Viewpoint Feature Histogram (CVFH) proposed in [34], is based on the VFH, while the main objective was to make the descriptor more robust to occlusion. It adds region-growing segmentation to subdivide the cluster into stable and smooth regions. The VFH is then computed on each sub-cluster. Thus, using this descriptor results in multiple histograms per object. Capabilities under occlusion are improved, but at the cost of longer computation, while the camera roll invariance remains.

Oriented, Unique, and Repeatable CVFH (OUR-CVFH) proposed in [35], enhances the CVFH descriptor by adding a unique reference frame. After the segmentation of CVFH, another filter is applied to form better-shaped regions using the difference between the sub-cluster normals orientation and their average. After such filtering, the Semi-Global Unique Reference Frame (SGURF) is computed, which constructs the reference frame. This repeatable reference frame should solve the camera roll invariance, making the descriptor more robust.

3.3.2 Ensemble of Shape Functions

The Ensemble of Shape Functions (ESF), proposed in [36], describes the cluster based on shapes and does not need information about normal vectors. Only segmentation is needed to compute the ESF, as it constructs its voxel grid approximation internally. Then, it randomly selects three points and computes three different shape functions. This process is repeated until all points in the cluster are iterated over. The results are then binned into a histogram of 640 bins. Due to the specific description, it is robust to incomplete surfaces.

3.3.3 Global Orthographic Object Descriptor

The Global Orthographic Object Descriptor (GOOD), proposed in [37, 38] also does not require information about normal vectors. It uses PCA to determine the principal axes of the cluster. Then the cluster points are projected onto three orthographic planes based on the principal axes. The description then represents the distribution of points on planes subdivided into several bins. The number of binds in the resulting histogram is only 75 bins, making it the most memory-efficient selected global descriptor. It should be robust to noise and density variance between cluster and database.

3.4 Offline stage

First, a database must be created for 3D object recognition using global descriptors. Moreover, due to the nature of global descriptors, the database must be based on partial view point clouds of objects. This method is referred to as a template-based method. In the considered scenario, the partial views are meant to be acquired exclusively using a virtual camera/scanner and a 3D model of the objects. Then, for each partial view, the chosen global descriptor is computed. Due to the virtual scanning approach, the transformation from virtual camera CS to object CS can be easily obtained, and the partial view point cloud is accurate. Thus, the computed description, transformation matrix, and model identifier are saved to the database.

A common way to create the database using a virtual camera is to place it on the vertices or centers of the faces of a subdivided icosahedron. This 3D shape ensures uniformly distributed points in the 3D space around its defined center point. Part of the PCL *Tools* module is a command line tool *Virtual Scanner* does exactly that [39]. The virtual scanning in this tool takes a mesh model as an input and, via ray casting, samples points on the mesh surface. Furthermore, the PCL has its own file format called Point Cloud Data (PCD), which allows the saving of both point clouds and computed global descriptors (histograms) [40]. Therefore, files of this file format can be utilized as part of the database.

3.5 Online stage

The main steps of the online stage include matching, coarse pose estimation, pose refinement, and hypothesis verification. However, preparing the depth camera/sensor data accordingly (preprocessing them) is essential to make the whole pipeline work efficiently and correctly. This typically includes filtering, downsampling, and segmenting the input scene point cloud.

3.5.1 Filtering

Filtering in this subsection depicts operations that discard unwanted points of a point cloud, leading to faster processing and improved data representation accuracy. It can be divided into two types: discarding points based on coordinates and discarding points based on their neighborhood. The most basic filters are also the fastest and, thus, are the most relevant for the considered scenario.

The two most basic filters for discarding points based on their coordinates are the passthrough filter and conditional removal [41]. The passthrough filter points out a specified range along the Z-axis, the same axis used for depth value by depth cameras. Conditional removal does the exact thing but for all three axes. These filters are especially useful for processing point clouds acquired by depth cameras because the Region Of Interest (ROI) is usually not identical to the whole captured data.

The most basic filter to discard points based on neighborhood is outlier removal [41]. Outliers are points that do not truly represent any surface or relevant information. They are commonly produced by noise, insufficient segmentation, insufficient filtering, or artifacts (for example, caused by reflective surfaces). Outliers may induce errors in many operations, including description and pose estimation. Radial outlier removal discards points based on the minimal number of neighboring points in a defined radius. Statistical outlier removal has a similar effect but removes the outliers based on statistical data

about each point rather than just the minimal number of points in a specified radius.

3.5.2 Segmentation

Due to the simplicity of the considered scenario and the possible use of the conditional removal filter to define the ROI, the only additional segmentation needed to process the captured scene is the removal of the dominant plane (supporting surface). Using PCL, this could be done using the RANSAC algorithm with a plane model [42]. RANSAC is a fitting algorithm that fits the specified parametric model to the given data. Moreover, a distance threshold is specified to identify inlier and outlier points based on the fitted model. After successfully fitting the plane model to the scene, the inlier points can be discarded, removing the points representing the supporting surface.

3.5.3 Matching

The matching step is essential for descriptor-based object recognition. It consists of searching for similar computed descriptors in the database created in the offline stage. All selected global descriptors (see Section 3.3) are represented by a histogram of a specific number of bins. The most common algorithm for searching similar histograms is the Nearest-Neighbors Search (NNS), specifically the k-NNS variant.

The k-NNS is used to find the k number of the most similar histograms while using selected metrics. The metric choice should depend on the descriptor's specifics and the specific application characteristics. The implementation of k-NNS in the PCL uses Fast Library for Approximate Nearest Neighbors (FLANN), which is highly computationally efficient [40]. Therefore, it is suitable for use in real-time applications. The output of this search is the k number of the most similar descriptions, their index, and the resulting distances of the chosen metric. The k number of recognition candidates is then evaluated in the following steps.

3.5.4 Coarse alignment

Due to the matching step retrieving an index of the candidates in the database, it is possible to obtain all the data from it for each candidate for it to be evaluated. These include a transformation matrix, a partial view point cloud, and the object identifier. Thus, the initial coarse pose estimation can be done solely by matching and retrieving database data. However, this initial estimate is often too coarse. Therefore, additional correction methods should be employed for translation and even rotation. The purpose of coarse alignment is to provide a reasonable initial guess for the subsequent use of iterative pose refinement methods (in the next subsection).

The additional translational correction can be made by computing the centroids of the filtered and segmented scene point cloud and the point cloud of the partial view of the candidate. The candidate point cloud can be translated to the scene point cloud based on the coordinate difference of the centroids, and this transformation can be added to the candidate coarse pose estimate. An alternative to this is a computation of 3D bounding boxes for both point clouds and using their centers instead of the centroids. The first-mentioned method should be more robust because it is less susceptible to errors caused by outliers.

Additional rotational correction can be done via PCA, which fits an ellipsoid into a point cloud. The components of this ellipsoid can then be used to construct the principal axes. However, this method is not ideal because it can lead to a wrong (flipped by

180 degrees) initial guess, possibly leading to local minima in the pose refinement step. Alternatively, the Camera Roll Histogram (CRH) descriptor proposed in [34] can be used to at least correct rotation along the Z-axis. After computing this descriptor for both point clouds, it can provide a possible relative angle between the two representations based on the phase shift of the histograms.

3.5.5 Pose refinement

The pose refinement step ensures the accuracy of the estimated pose and allows for the hypothesis verification step. The algorithm most widely used for this task is the ICP algorithm and its variations [4]. It consists of finding an optimal rigid transformation between two point clouds, one labeled as the source and the other as the target. However, because it is an optimization-based algorithm, it can fall into local minima, leading to a wrong pose estimate. Therefore, a reasonable initial guess must be made to obtain an accurate result. Even with a good initial guess provided by coarse alignment, the pose refinement step requires considerable computation time. Thus, appropriate termination criteria must be chosen for the solution to be efficient. There are mainly two variants of the ICP: point-to-point and point-to-plane. The optimization methods often used for ICP are the Least Squares or Singular Value Decomposition.

The point-to-point ICP algorithm is the most basic variant. In every iteration, two steps are performed. Firstly, point correspondences are found using NNS, and then the rigid transformation is estimated. After each iteration, the transformation is evaluated by the sum of the distances between the correspondences.

The point-to-plane ICP algorithm differs from the point-to-point by incorporating information about the normal vector of the points from one of the point clouds to compute the rigid transformation. As a result, this version converges faster while also being more accurate and robust in the presence of noise (as mentioned in Subsection 3.1.2). Additionally, the symmetric objective function proposed in [43] may be used, incorporating information on the normal vectors of both point clouds, further improving the computational efficiency.

3.5.6 Hypothesis verification

Hypothesis verification serves to select the best candidate produced by the pipeline. One of the simplest hypothesis verification methods, greedy hypothesis verification, was proposed in [35]. It accepts scene and model point clouds and computes the number of inlier and outlier points based on the distance of model points to the nearest scene point. Moreover, occlusion reasoning is included to consider only the possibly visible points based on input view point coordinates. Then, a simple metric is computed for each candidate, where a weighted number of inlier points is subtracted from the number of outlier points. Additionally, a threshold can be set for this metric to discard false positives.

3.6 Key PCL dependencies

The PCL has several mandatory and few optional dependencies [40]. Four of these are essential to know about due to the implementation of the solution later in this thesis. These are the Boost, OpenNI2, Eigen, and Visualization ToolKit (VTK).

Boost is a collection of free peer-reviewed portable C++ source libraries. It offers a wide range of functionality commonly preceding the implementation of the same functionality in the C++ standard library. [44]

OpenNI2 library, which provides support for various depth cameras and sensors. Designed for interaction without the need to utilize proprietary libraries from manufacturers directly. [45]

Eigen library for linear algebra, especially vector and matrix operations. It contains handy classes for transformations and their manipulation. It also enables conversions between rotation representations, including axis-angle representation. [46]

Visualization ToolKit (VTK) library that is extensively used in the PCL, especially for processing standard 3D data formats (like STL) and also for visualization. Provides a wide variety of functions for mesh processing, including transformations. [47]

4 Solution draft

The problem to be solved is the 3D object recognition and pose estimation, where 3D models are the only references to the possible objects. First, in Chapter 2, the problem was described, assessed, and the solution methods were narrowed, including the choice of the main library for 3D data processing. The solution methods were narrowed to use hand-made feature-based representations for 3D object recognition and a coarse-to-fine approach to pose estimation. Then, in Chapter 3, related work was assessed, further specifying the methods for the solution, mainly to the use of global descriptors. This choice was made in the Subsection 3.1.4. For the diagram of the global pipeline based on related work, see Figure 3.2. The individual parts of this pipeline were then described, providing an overview of the entire algorithm for the solution. Thus, the previous chapters already provide the algorithm to solve the problem.

As the sequence of steps appears to be reasonable, the algorithm itself is not to be improved in this thesis. However, a packaged solution that encapsulates such an algorithm and the specifics of the considered scenario and enables easy testing and diagnosis to ensure recognition and pose estimation capabilities on a given set of 3D models is currently not widely available. These ideas then form the solution proposed in this chapter. Thus, this chapter serves as a draft for implementing the solution without implementation details. First, the required features of the solution are defined. Then the architecture of the solution is proposed, together with a description of its individual parts and intended usage. Finally, an approach for validation of the solution is also proposed.

4.1 Required features

The required features come primarily from the logical reasoning of the considered scenario (described in Section 2.1). The main aim of the solution created in this thesis is to provide a tool that can be easily implemented into other applications while being extensively configurable, providing an easy way of testing its functionality on a specific set of 3D models and enabling the user to diagnose and troubleshoot the solution. Furthermore, this tool should also be expandable for adding different pipelines (e.g., local descriptors).

Thus, to itemize the ideas above, the solution should:

- be modular,
- be relatively easy to use and implement in applications,
- be extensively configurable,
- use a 3D model of the objects as the only reference,
- enable easy testing, diagnostic, and troubleshooting.

4.2 Architecture proposal

Based on the required features (see Section 4.1), it was decided that the solution should contain three main modules: the Utilities module, the Dataset preparation module, and the Global pipeline module (containing offline and online steps). The combination of these modules then forms the solution. Furthermore, to simplify the usage of the resulting solution in potential applications, the entire solution will be encapsulated by a wrapper, providing an easy interface for its use. The structure of the entire solution can be seen in Figure 4.1. The remainder of this section describes the purpose of each module in detail.

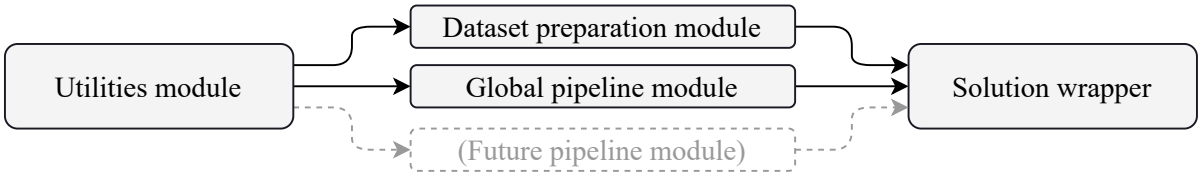

Figure 4.1: The structure of the proposed solution, arrows indicate dependency.

4.2.1 Utilities module

The Utilities module should serve as a multipurpose module that is used by the other modules. Thus, providing access to data in the database and containing other operations that may be used across modules to prevent code duplication. Moreover, it should also contain functionality to evaluate recognition and pose estimation accuracy based on given ground truth information. The metrics to evaluate pose estimation accuracy should be based on the error functions mentioned in Subsection 2.4.1. The module should also support the export and visualization of all relevant information to use for additional evaluation, diagnostic, and troubleshooting purposes.

4.2.2 Dataset preparation module

The Dataset preparation module is made independent to meet the condition of expandability for other potential pipelines. Its purpose is to prepare the partial view point clouds of the input 3D models. These point clouds can then be fed into the offline stage (database creation) of the chosen pipeline. The partial view point clouds of the 3D model should be generated using a virtual depth camera. Furthermore, the viewpoints of the virtual depth camera should be uniformly placed around the 3D model, thus employing the icosahedral technique to generate viewpoints mentioned in both Sections 3.1 and 3.4. Moreover, parameters for creating a dataset and other relevant information should be saved in a human-readable database to be easily overviewed.

Additionally, the dataset preparation module should also provide the functionality of generating datasets that will be used to test the capabilities of the whole solution. For this purpose, the virtual depth camera should be able to simulate the noise of the real depth camera. The ability to generate data for experiments enables the possibility of automated testing of the solution with a specific set of parameters and objects.

4.2.3 Global pipeline module

The Global pipeline module should contain the detailed global pipeline described in Chapter 3, which is based on related work (see Section 3.1), apart from the generation of the partial views. From the usability standpoint, this pipeline should be further subdivided. The offline and online stages should be divided because the offline stage must be executed only once per given configuration. Moreover, the online stage should be subdivided into initialization and iteration steps. This is similar to separating offline and online stages because the online stage must be initialized only before the first iteration step, which can then be used indefinitely while only providing input scene point cloud data for each iteration. Therefore, it should be designed for possible depth camera data feed usage. The module should enable easy troubleshooting and diagnosis of each step. Thus the result data should be optionally exported according to the configuration for additional evaluation or for use in an application.

4.2.4 Usage

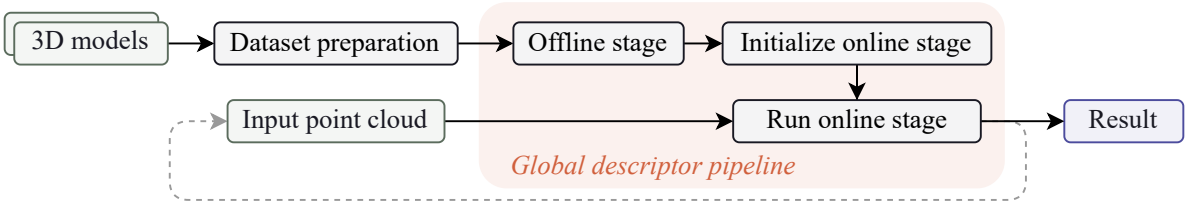

Figure 4.2: Illustration of the usage of the proposed solution.

The illustration of the intended usage of the solution is in Figure 4.2, which forms the solution pipeline. As can be seen in the diagram, each iteration of the online stage outputs a result. As a result, all relevant information about the recognized model and its pose estimate is meant, as well as additional data that can be used for diagnostic and troubleshooting. An example of such data is information about all 3D object recognition/pose estimation candidates. The amount of additional data should be an option in the configuration. All the parts of the solution pipeline should be encapsulated by the solution wrapper, providing an easy interface for using the solution.

4.3 Validation of the solution

The proposed solution should be validated in several experiments evaluating object recognition and pose estimation capabilities. The solution testing functionality should be provided by the Utilities module, as mentioned above, while having an easy interface in the solution wrapper as it is part of the requirements. The test experiments should include tests on virtual data generated by the Dataset preparation module and tests on data captured by a real depth camera.

5 Implementation

This chapter describes the implementation details of the solution proposed in Chapter 4. First, information and reasoning on the selected tools and algorithms is provided. Then, the hardware (HW) and software (SW) used are listed. Subsequently, the implementation of each module is described. Moreover, the source code, which is part of the attachments, is sufficiently commented on and understandable to make the implementation even clearer. Thereafter, this chapter, together with the attached source code (see Appendix A), serves as the documentation of the solution. Additional extended information is available in Appendix C.

5.1 Tools and algorithms

As mentioned in Section 2.6, the Point Cloud Library (PCL) [48] is used for the implementation of the solution due to alternative libraries not including functionality and state-of-the-art methods relevant to the problem. Furthermore, this choice requires the solution to be programmed using the C++ language, which requires a considerable amount of experience to work with. To ensure that the created solution would be available to a wider audience, it was decided to bind the resulting C++ solution wrapper to a Python module using Pybind11 [49]. This allows the end users to use the solution wrapper via Python, which is currently considered to be one of the most popular approachable programming languages.

It was decided not to use complex database systems to simplify the solution and instead to use JavaScript Object Notation (JSON) files and logical folder structures. Such files provide a solid compromise between parsing data for the computer and human readability. However, because the C++ standard library does not support this file format, the open-source library [50] is used instead. On the other hand, the Python standard library contains module for JSON parsing and thus does not require additional packages. Using such a universal data format gives the user high flexibility for data processing.

Other dependencies than those above include only the libraries needed for using the depth camera at hand, the Intel RealSense D415. However, the dependency list is much larger due to the PCL having various mandatory dependencies (as mentioned in Subsection 3.6).

In addition to the above, the entire solution was implemented and is intended to be used within a Docker [51] container. This follows the current trends in SW development. Using a Docker container ensures a consistent operating system environment, which (at least partially) eliminates potential problems caused by incompatibility with libraries and other SW already installed on the host machine. It creates a layer above the host operating system, which is partially sandboxed. This allows anybody with a Linux operating system to try the solution without worrying about the impact of needed additional SW. Additional instructions for trying the solution (or further developing it) are in Appendinx B.

5.2 List of used hardware and software

The PC setup was as follows:

- Intel® Core™ i7-6700HQ CPU @ 2.60GHz,
- 32 GiB RAM,
- Ubuntu 22.04 LTS.

The depth camera used for the experiment on captured data and its specifications:

- Intel® RealSense™ D415 [1],
  - active stereo vision,
  - resolution up to 1280 × 720,
  - Field Of View (FOV) for HD resolution of 50 × 40 deg,
  - minimum depth distance at max resolution approximately 45 cm,
  - depth accuracy < 2 % at 2 m.

The used libraries and their versions:

- Docker 23.0.1,
- CMake 3.22.1,
- Ninja 1.10.1,
- GCC 11.3.0,
- Point Cloud Library 1.13.0,
  - OpenNI2 2.2.0.33,
  - OpenMPI 4.1.2,
  - Visualization ToolKit 9.1.0,
  - QT 5.15.3,
  - SQLite 3.37.2
  - QHULL 2020.2,
  - Boost 1.74.0.3,
  - Flann 1.9.1,
- RealSense 2 SDK 2.50.0,
- Catch2 3.2.1,
- Nlohmann json 3.11.2,
- Pybind11 2.10.3,
- Python 3.10.6.

[1] Product website `https://www.intelrealsense.com/depth-camera-d415/`.

5.3 Implementation of the solution draft

The entire solution was programmed using the C++14 standard, as it was, at the time, the latest version supported by PCL 1.13.0. Thus, the Boost filesystem library had to provide functions regarding the file system since equivalent functions were only implemented in the C++ library above the C++17 standard. Furthermore, the Eigen library poses a memory alignment problem for arrays and matrices in specific scenarios when used below the C++17 standard. After much time was wasted on solving this problem, it was finally decided to disable all Eigen optimizations while building the PCL to avoid it completely. This results in worse computational performance but ultimately avoids segmentation faults caused by the said problem. Due to this fact, the implementation of the solution serves only as a proof of concept, and no additional optimization (including threading) was considered.

As mentioned above, the solution utilizes the detailed pipeline described in Chapter 3 while also being inspired by available tutorials [18, 32, 41, 42] that use the PCL. The whole solution pipeline is based on the following equation.

$$\mathbf{T}_{\text{object}} = \mathbf{T}_4 \cdot \mathbf{T}_3 \cdot \mathbf{T}_2 \cdot \mathbf{T}_1, \tag{5.1}$$

where $\mathbf{T}_{\text{object}}$ is the resulting pose estimate, and the numbered transformations are acquired successively through the solution pipeline. The reasoning and details of this approach are further explained in the following subsections, as it relies on the whole solution pipeline sequence.

As proposed in Chapter 4, the solution consists of three main modules and a solution wrapper. This wrapper is then bound to a Python module using Pybind11, enabling users to use the wrapper via Python. The implementation of individual modules is described below. Individual modules use namespaces for better code readability.

5.3.1 Utilities module implementation

As mentioned above and in the Subsection 4.2.1, the Utilities module provides functions that are used across the other modules. The structure of its implementation is illustrated in Figure 5.1. It is divided into two namespaces: *mtjson* (master thesis JSON) and *mtu* (master thesis utilities).

The *mtjson* namespace provides functions and objects related to the use of JSON files in the solution. This includes functions for loading and saving JSON files themselves and structures, which are used both to parse the data and as data holders for the relevant objects in the other modules. The purpose of each of these structures is described in place of their usage further down this chapter. These structures represent equally the JSON files that are used throughout the pipeline. The detailed definition of all of these structures, together with file structures that are created during the execution of the solution pipeline, is located in Appendix C.

The *mtu* namespace provides functions used in the other modules and not related to JSON files. These are relevant data input/output functions, various filtering methods, and point cloud operations. The names of these functions are self-explanatory. The generated timestamp is used as primary identification for files related to one iteration of the global pipeline. The *mtu* namespace also contains two significant classes, the *AccuracyTest* and the *Viewer*.

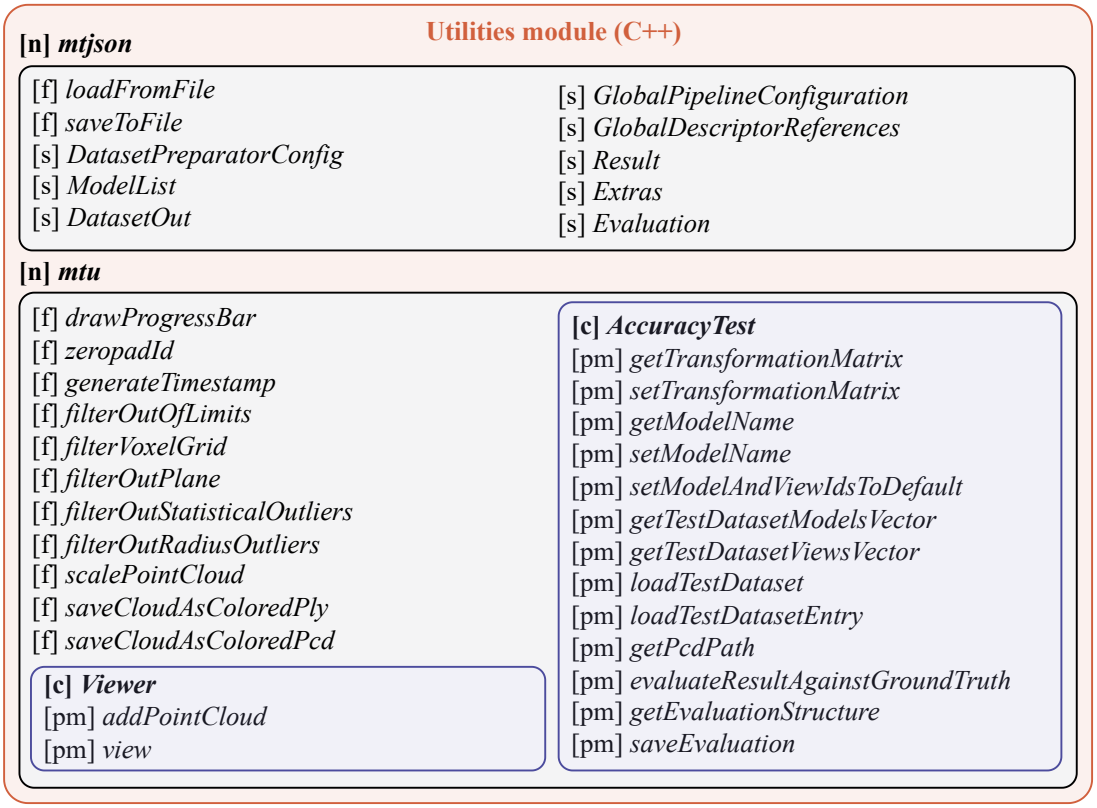

Figure 5.1: Utilities module structure including namespaces, functions, structures, classes, and public methods.

The *AccuracyTest* class supports testing both the object recognition and pose estimation accuracy. This is done by comparing the input ground-truth data with the solution pipeline result (defined in Appendix C). The accuracy test results are outputted to *Evaluation* structure object, which includes information about the correctness of recognition and the accuracy of pose estimation using errors described in the Subsection 2.4.1.

The *Viewer* class provides an easy interface to visualize point clouds. It is used mainly for diagnostic purposes in the Global pipeline module. As there is currently a bug in clearing the scene of the viewer, it is always advised to use the viewer within the subscope to ensure its one-time use.

5.3.2 Dataset preparation module implementation

As stated in the Subsection 4.2.2, the Dataset preparation module prepares the partial view point clouds of the given 3D models. This is the prerequisite for the subsequent computation of global descriptors in the offline solution pipeline stage. Moreover, it can also add Gaussian noise to the generated partial views in two modes: in three directions (3D space) or only along the view ray. The structure of the module implementation is illustrated in Figure 5.3. It includes the namespace *mtdp* (master thesis dataset preparation) consisting only of two classes, *VirtualDepthCamera* and *DatasetPreparator*.

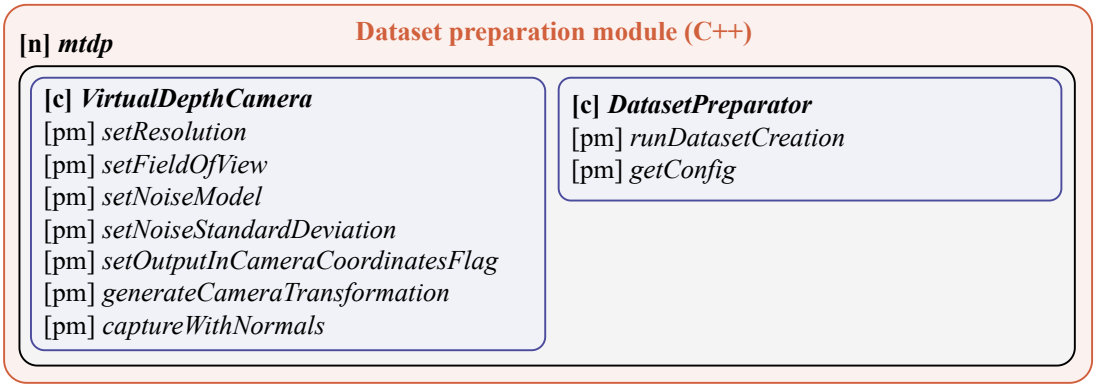

Figure 5.2: Dataset preparation module structure including namespaces, classes, and public methods.

Based on Subsection 2.2.1 and after validating supported file formats of the PCL and its dependencies, it was decided to use the *STL* (mesh model) file format for 3D models of objects. Moreover, all data is assumed to be in the same units, millimeters. This may not be the same unit that various depth cameras consider, but that is why the Utilities module provides a function to scale the input point cloud. Using millimeters instead of meters precedes problems with floating-point precision computations throughout the whole solution pipeline. 3D CAD models entering the dataset preparation are assumed to be already exported as STL models in millimeters.

The *VirtualDepthCamera* class is a modified *virtual scanner*[2] PCL tool. This tool was originally intended to be used through the command line and perform virtual scanning of a given 3D model using the ray casting method. It also allows the user to add Gaussian noise to the output partial view point cloud (only in all three directions). However, its input parameters are restricted together with the file formats it accepts. Therefore, the code was modified and incorporated into the module as a separate class. The modifications are as follows. The accepted 3D data are changed to a more general VTK polygon type, enabling it to take loaded STL models. The tool was enhanced by adding functionality for ray casted points to inherit the normal of the polygon from the STL description, which is inspired by the approach of another PCL tool: *mesh sampling*[3]. Another modification is the opinion to generate Gaussian noise only along the view ray, which makes the resulting noise more realistic than the original approach of adding random numbers to all three coordinates of each point. Moreover, the parameters were modified to resemble real depth camera parameters. Thus, it is possible to set the FOV and the resolution. The last modification was the generation of a virtual depth camera transformation matrix based on its position and view point, while ensuring that the X-Axis is in the base XY plane.

The *DatasetPreparator* class is the main class of this module. It contains and internally uses the *VirtualDepthCamera* object. It has only one important public method *runDatasetCreation*, which encapsulates the whole module and, thus, the dataset preparation step in the solution pipeline. The method takes paths to a *DatasetPreparatorConfig* and a *ModelList* JSON file as input. The *DatasetPreparatorConfig* contains parameters

[2]Link to its source code `https://github.com/PointCloudLibrary/pcl/blob/master/tools/virtual_scanner.cpp`.

[3]Link to its source code `https://github.com/PointCloudLibrary/pcl/blob/master/tools/mesh_sampling.cpp`.

for creating datasets, including parameters for the *VirtualDepthCamera*. On the other hand, the *ModelList* includes just a list of paths to STL models of objects that are contained in the created dataset. The creation of datasets results in a logical folder structure with relevant files, including partial view point clouds and also *DatasetOut* JSON, which contains all relevant information about the created dataset. For more details on the JSON files and file structure, see Appendix C.

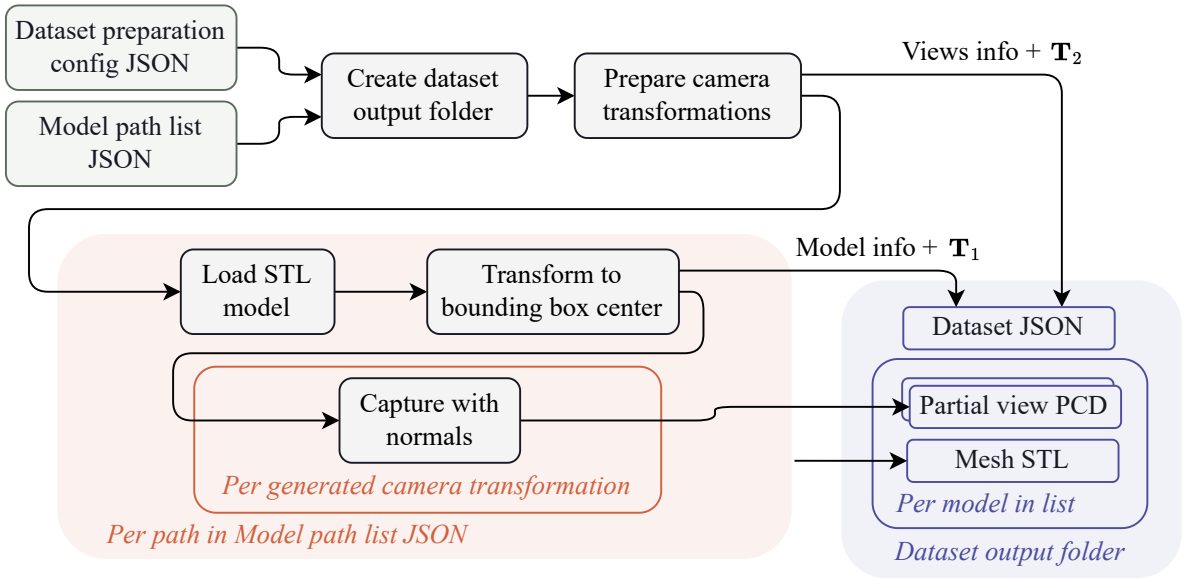

Figure 5.3: Illustration of the dataset creation implementation.

The sequence for generating partial views is largely based on Section 3.4 while also incorporating ideas from Chapter 4. The illustration of the implementation of the dataset creation is in Figure 5.3. After loading the relevant files, the output folder is created based on the path given as the argument for the *runDatasetCreation* method. Then, the camera transformations are generated using an icosahedron (see illustration in Figure 5.4) and stamped with an identification number (ID). The center of its faces or the vertices can be used for camera positions, while the center of the icosahedron is used as a viewpoint. There is also a parameter for subdivision, which results in 12, 20, 42, 80, or 162 camera transformations. The inverses of each camera transformation generated, labeled as $\mathbf{T}_2$, are then added to the *DatasetOut*. Then, the following sequence of operations is performed for each STL model of *ModelList*. The STL model is loaded as a mesh polyfile and stamped with its ID, which is added to the relevant list in *DatasetOut*. Then, its CS is translated to the center of the 3D bounding box of the model. This ensures better positioning of the model in relation to the icosahedron. This transformation is labeled $\mathbf{T_1}$ and added to the relevant list in *DatasetOut*. Subsequently, the method *captureWithNormals* of the *VirtualDepthCamera* is used for every generated camera transformation. Each generated partial view point cloud is then saved in the relevant folder as PCD file, with the name corresponding to the ID of the view. In addition to that, the source STL file is also saved in the relevant folder for convenience.

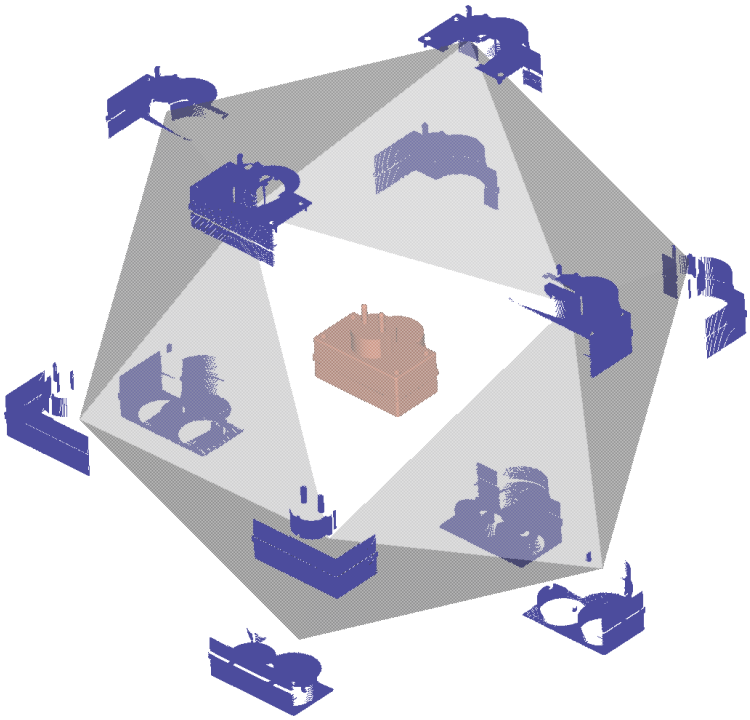

Figure 5.4: Illustration of the model, icosahedron and partial views.

5.3.3 Global pipeline module implementation

As mentioned above and in the Subsection 4.2.3, the Global pipeline module is based on the detailed pipeline described in Chapter 3. As proposed, the module is subdivided into offline and online stages, while the online stage is further subdivided into initialization and iteration parts. This module also serves as a hint for the implementation of different pipelines. The structure of the module implementation is illustrated in Figure 5.5. It includes namespace *mtgp* (master thesis global pipeline), which consists of *Candidate* structure and two main classes, the *GlobalDescriptor* and the *GlobalPipeline*.

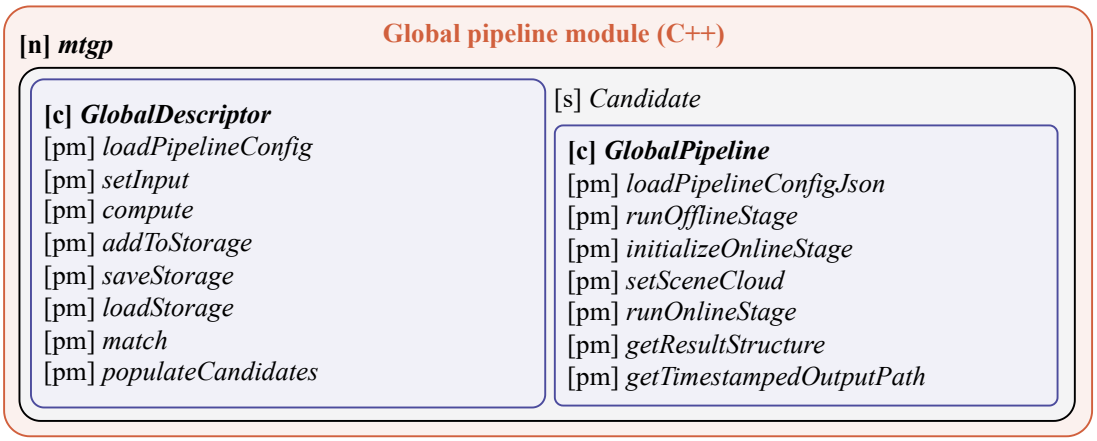

Figure 5.5: Global pipeline module structure including namespaces, structures, classes, and public methods.

The *GlobalDescriptor* class encapsulates all the selected global descriptors (see Section 3.3). Apart from the GOOD, all other global descriptors are implemented in the PCL. The source code of GOOD[4] needed to be modified to work properly with PCL 1.13.0. More specifically, the Boost shared pointers had to be replaced by standard library shared pointers. Due to the use of template methods and the need for simple incorporation into the solution, the whole implementation was moved to a single HPP file. In the case of adding another global descriptor, it is preferable to augment only this class (and the relevant part of the Utilities module).

The *GlobalPipeline* is the main class of the global pipeline. It encapsulates the entire global pipeline process. Moreover, it enables the user to define a folder path for saving output from online stage iterations in its constructor. Both the offline stage and the online stage involve the loading of two JSON files. The first is a *GlobalPipelineConfiguration* JSON, which contains all relevant parameters for each configurable step in both the offline and online stages. The second is the *DatasetOut* JSON, which is meant to be located in the folder generated while creating a dataset within the solution pipeline.

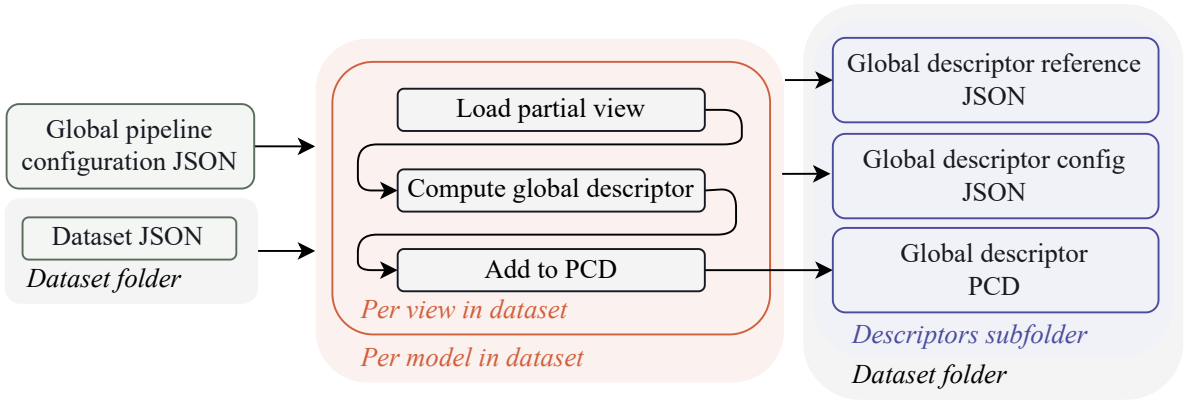

Figure 5.6: Illustration of the offline stage implementation.

Figure 5.6 illustrates the implementation of the offline stage. After loading the two mentioned essential JSONs, it loads each partial view of each model in the dataset. Then the selected global descriptor (one of the global descriptors mentioned in Section 3.3) is computed, and the result is added to the storage. The storage is an array of computed global descriptors that are saved as a PCD file at the end of the offline stage. For the exact association of the computed global descriptors with the dataset, the procedure generates a *GlobalDescriptorReferences* file. Apart from that, the implementation also saves the configuration of the used global descriptor, so in case it is equivalent for multiple runs, it does not recompute the same descriptors, thus saving computation time. The resulting files are saved in the dataset folder structure. This step must be executed only once per specific global descriptor configuration.

The illustration of the implementation of the online stage is in Figure 5.7. After loading the two already mentioned JSONs, it loads the computed global descriptors (from the offline stage) PCD file and the *GlobalDescriptorReferences* JSON containing references to the dataset. Apart from that, all partial views of an object are fused to form a point cloud of the whole object without possible inner cavities, which is used later in the hypothesis

[4]Link to its repository `https://github.com/SeyedHamidreza/GOOD_descriptor/tree/master`.

verification step. This forms the initialization of the online stage and must be executed at least once per global pipeline run before any online stage iteration.

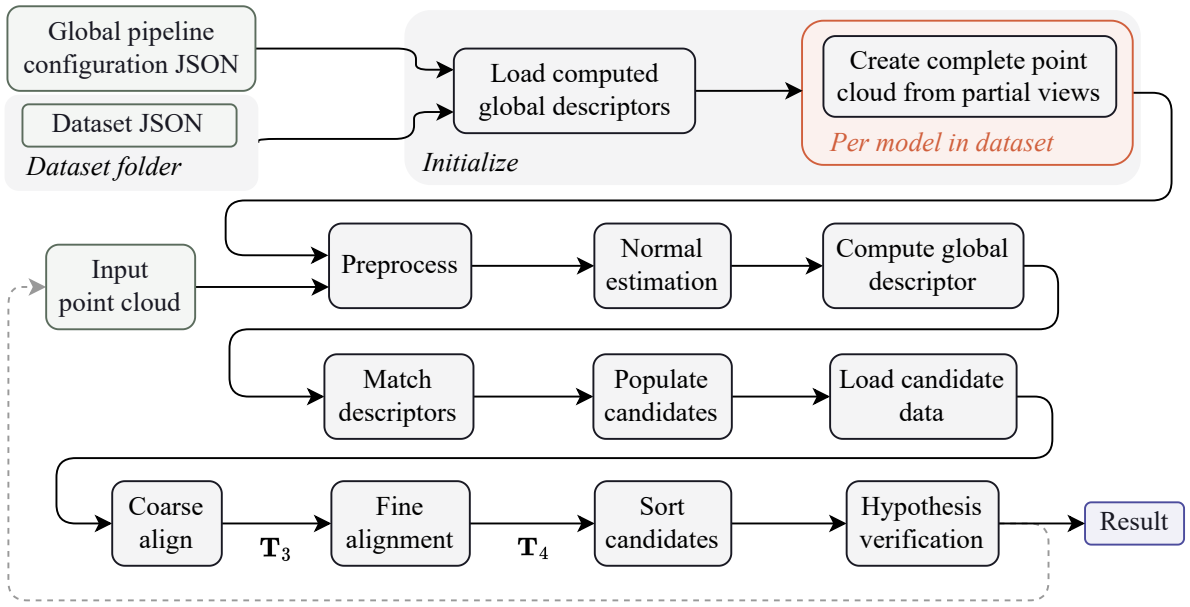

Figure 5.7: Illustration of the online stage implementation.

The iteration of the online stage uses, apart from the data already loaded during the initialization, only an input scene point cloud. Then, based on the *GlobalPipelineConfiguration*, it proceeds through all the steps of the global pipeline and retrieves a *Result* structure object. This object is then saved under the defined output folder into a newly created subfolder named, using a timestamp, that is generated for each iteration. Each step has several parameters configurable through the *GlobalPipelineConfiguration*, described in Appendix C.

The preprocessing step enables to scale, filter, and segment the input point cloud of the scene. The user can choose only the relevant operations required for a specific setup. The point cloud, after preprocessing, should include only the data related to the object of interest. This can be verified by visualization.

The normal estimation step is particularly important for the solution pipeline, as the normals are used both for recognition (besides GOOD and ESF descriptors) and final pose estimation. Its radius parameter influences the global smoothness of resulting normals. Thus setting it too large leaves the results without much geometric information about the underlying object, while setting it too small can cause distortions in case of noisy data. Thus, the setting should be compromised and consider the amount of noise.

The compute global descriptor step computes the global descriptor of the preprocessed input point cloud with estimated normals. All global descriptors mentioned in Section 3.3 are available, with their most relevant parameters configurable. This is the same operation as for all partial views in the offline stage.

The matching step matches the computed global descriptor with the global descriptors computed in the offline stage. This results in k candidates based on the distance of the match. The metric used by k-NSS for all of the descriptors is set to L1 and is not configurable to not add another complexity to the already complicated solution.

The populate candidates step populates the vector of *Candidates* objects based on the references from the offline stage. After the candidates have proper references to the dataset, the load candidates step loads relevant partial point clouds of each candidate and add all already known information, such as the $\mathbf{T}_1$ matrix, $\mathbf{T}_2$ matrix, and match distance.

The coarse alignment step coarsely aligns the loaded point cloud of each candidate to the scene point cloud. It is possible to use CRH and, in the case of OUR-CVFH or GOOD, even their alignment estimation. Moreover, there is an option to remove translation offset in coarse alignment by centroid correction. After the transformation of each candidate point cloud, its coarse alignment transformation is stored in the relevant *Candidates* object as $\mathbf{T}_3$.

The fine alignment step finely aligns the loaded point cloud of each candidate to the scene point cloud. It uses point-to-plane ICP with the option to use a symmetric objective function, which speeds up convergence and provides better accuracy. After the transformation of each candidate point cloud, its fine alignment transformation is stored in the relevant *Candidate* object as $\mathbf{T}_4$ and ICP fitness. Because this step is usually the most computationally demanding, there is an option to additionally use a voxel filter on both the scene and the candidate point cloud. This can considerably shorten the computation time but can result in lower final pose accuracy. At this point, as all the partial transformations of the solution pipeline are known, the $\mathbf{T}_{\text{object}}$ (see Equation 5.1) is computed for each candidate.

The sort candidates step then sorts the candidates based on the value of the candidate's ICP fitness, representing the average distance between correspondences. Thus, the sorting is done from the lowest ICP fitness to the highest. Candidates with lower ICP fitness are more likely to be correct. However, it could also be just a local minimum in an inaccurate pose.

The hypothesis verification step then helps skip candidates in inaccurate poses or entirely different objects. It goes sequentially through the sorted candidates and searches for the first candidate that succeeds in greedy hypothesis verification. This step uses the fused model representation created in the offline stage, transformed to $\mathbf{T}_{\text{object}}$ pose. Configuration of this stage is essential to eliminate (at least partially) false positives and incorrect poses. At the end of the hypothesis verification step, the result is embedded into *Result* structure object and saved to the iteration output folder.

Moreover, the implementation has several diagnostic options available under the *debug* key in the *GlobalPipelineConfiguration*. This includes printing results to the command line and the notion of the individual steps in progress. Then there is an option to save an *Extras* structure object, which includes all information about each of the candidates, the scene point cloud, and the computation time of individual steps. Other options enable saving point clouds from the global pipeline itself, used *DatasetOut* JSON and used *GlobalPipelineConfig* JSON. The last practical option is to visualize selected steps in the pipeline, which is especially useful during finetuning of the parameters. All additional files, if requested, are also saved to the iteration output folder, just as the *Result*.

5.3.4 Solution wrapper implementation

The solution wrapper envelops the entire solution pipeline, providing an interface for easier use in potential applications. Thus, it covers all the steps, from creating a dataset to the accuracy testing. The structure of the solution wrapper implementation is illustrated in Figure 5.8. The solution wrapper is bound to Python using the already mentioned Pybind11 library into the *mtsolution* module. The bound functions have the same names and arguments as the solution wrapper.

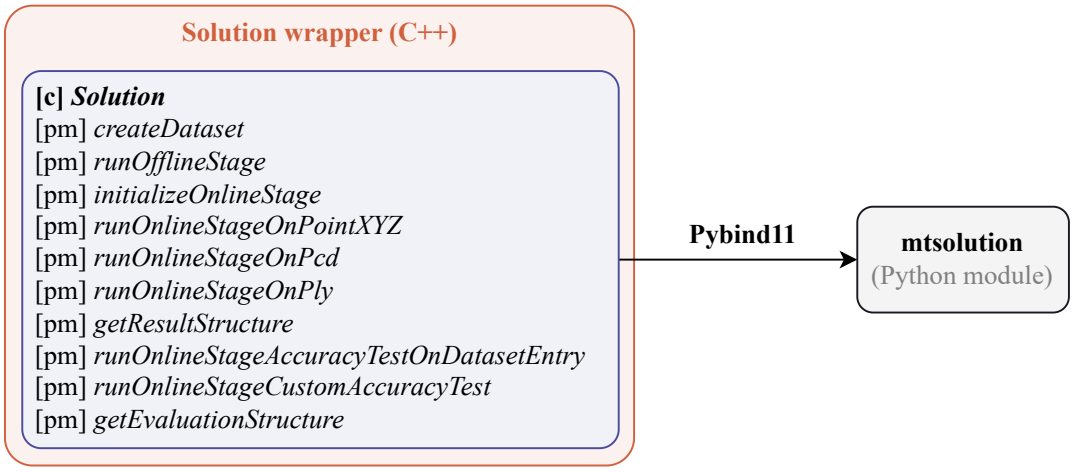

Figure 5.8: Illustration of the solution wrapper implementation, including classes and public methods.

The solution wrapper involves only one class that wraps around all the other modules described in the sections above. This class is named *Solution*. When constructing an object of this class, it can take an argument equal to the argument for the *GlobalPipeline* class. Thus, specifying the path for the output folder. The created *Evaluation* structure object is also saved to this folder when using the accuracy test functionality. The individual methods are then self-explanatory. This class can be used in Python scripts thanks to the binding to the Python module *mtsolution*. The binding also provides the *Result* and *Evaluation* structures as Python classes for convenience. Less advanced users should use the solution only through the Solution module, which makes the whole solution more accessible and flexible. Configuring the solution pipeline is as easy as editing relevant JSON files in a text editor. Illustration of the solution pipeline, which visually clarifies the Equation 5.1, can be seen in Figure 5.9. In case of interest in using the solution, getting inspired, or developing it further, see Appendix A as well as Appendix B and Appendix C.

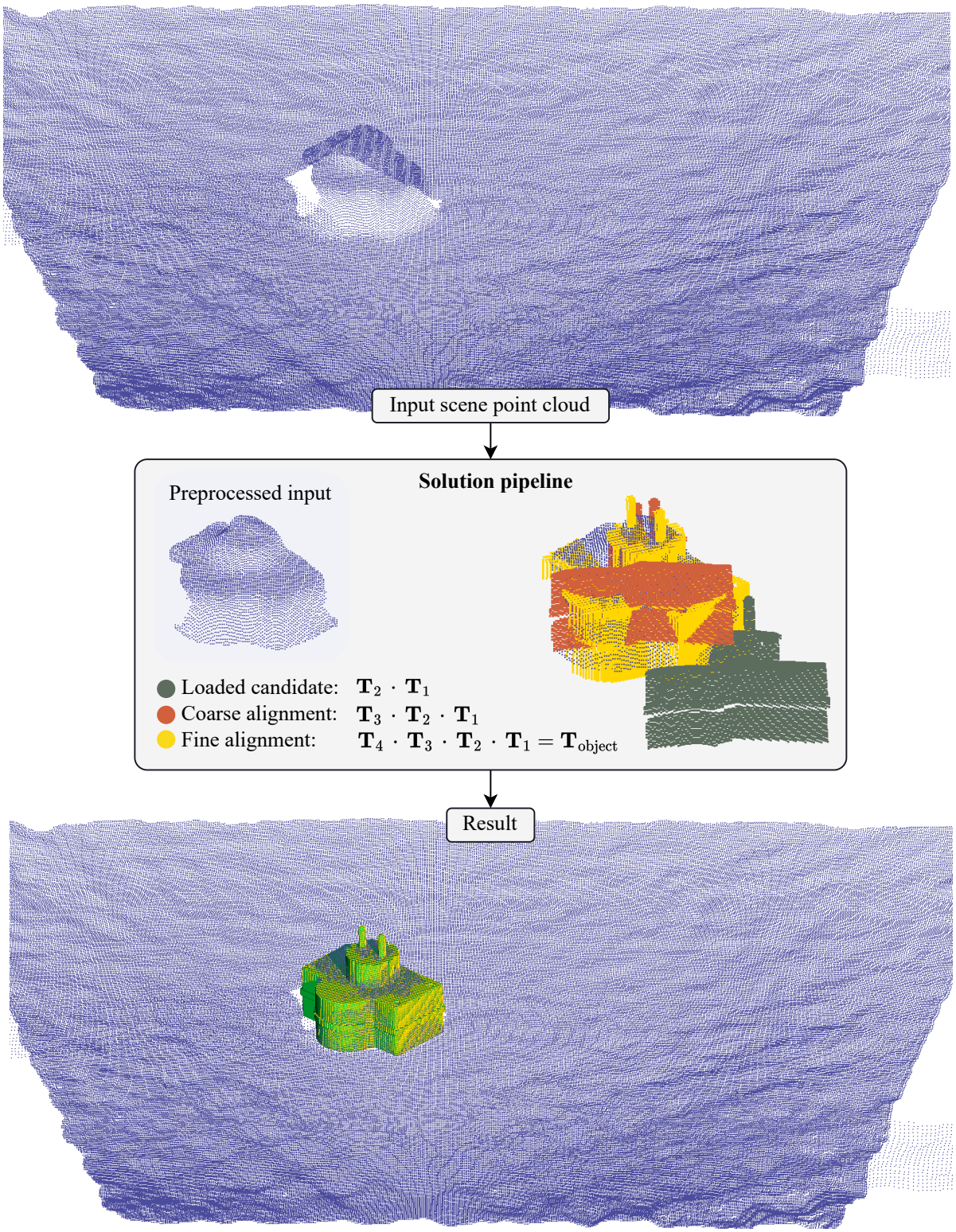

Figure 5.9: Illustration of the solution pipeline.

6 Validation of the solution

As proposed in Section 4.3, the solution was tested by conducting experiments on data of two categories: artificially generated datasets using the solution and real-world data captured with the Intel RealSense D415 depth camera on an improvised setup. Three main types of experiments were conducted: ground truth tests, virtual noise tests, and captured tests.

This chapter is structured as follows. First, the data used for the experiments are presented and described in more detail. Then, the conditions and details of the execution of the experiments are described. After that, results from each experiment are shown and commented on. Finally, the created solution is assessed, its limitations identified, and advancements proposed.

6.1 Data

For validation of the solution, data for experiments had to be generated first. In this section, first, the selected 3D models for the experiments are presented. Then, the description and visual examples of the input scene point clouds for each type of experiment are shown, including a closer description of the real-world data capture.

6.1.1 3D models and its groups

Due to the realization of the real-life data capture, three industrial-looking objects that were available at hand were selected, modeled in 3D CAD software, and exported as STL files. However, to test the solution capabilities with more than three objects, another six objects from the DeepCAD dataset (which was mentioned in Subsection 2.3.3) were also selected. All of these models and their identification names in testing datasets can be seen in Figure 6.1. It is worth noting that the origin CSs of these models are placed on various points of the objects. Thus, a rotation error of 180 degrees can also result in a significant translational error, which may look unintuitive.

For the experiments, three 3D model lists further referred to as groups, were created:

- *One object*, containing only one 3D model, which is the *real_object_0*.
- *Real objects*, containing all three 3D models of real objects.
- *Six objects*, containing all the six 3D models selected from DeepCAD dataset.

These groups were created to better evaluate the behavior under different conditions from the database size standpoint. All three groups are used for ground truth and virtual noise experiments. Only the first two groups mentioned are used for the captured tests as none of the models in the *six objects* groups were captured in the real world.

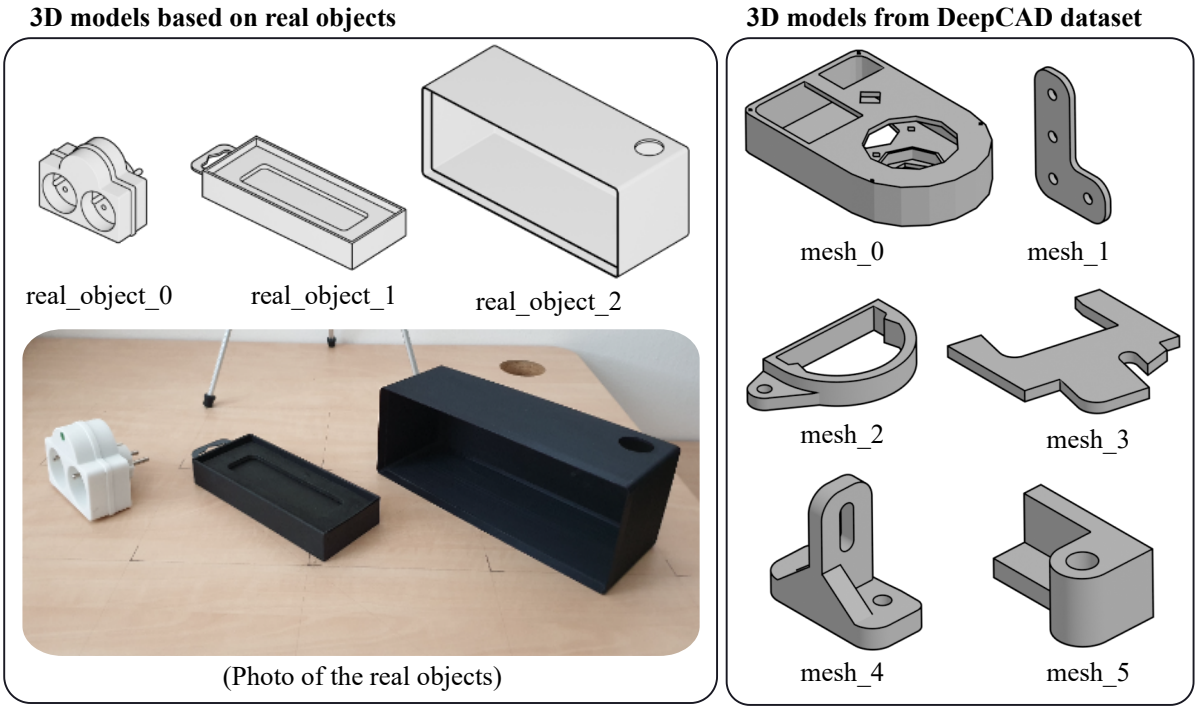

Figure 6.1: 3D models of selected real objects and selected 3D models from the DeepCAD dataset, together with identification names.

6.1.2 Generated datasets

All the artificial data used as input for the experiments were created using the dataset creation functionality of the solution. It was decided to use configuration, which results in 42 views per object for both datasets for the database and for the accuracy test input. The exact configuration parameters used to create all the datasets can be checked in Appendix D. The virtual depth camera parameters are set to resemble the Intel RealSense D415 depth camera parameters in the VGA setting.

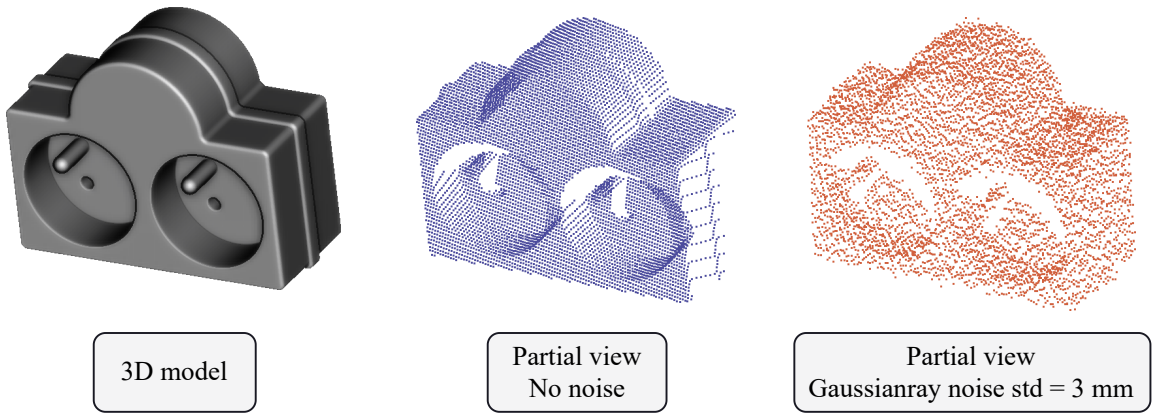

Figure 6.2: Example of generated partial views for accuracy tests.

The dataset for ground truth tests is characterized by no added noise to the generated partial views, an icosahedral radius of 650 mm, and a global rotation of 5 degrees. In

contrast, the dataset for virtual noise tests has added noise with a standard deviation of 3 mm using the Gaussianray noise model, an icosahedral radius of 700 mm, and a global rotation of 15 degrees. An example of generated partial view point clouds both with and without noise can be seen in Figure 6.2. Data generated using the solution needs no preprocessing, as it is already segmented. Thus, all preprocessing flags for experiments on artificially generated datasets are set to false.

6.1.3 Captured data

The real dataset was captured using the Intel RealSense D415 depth camera on an improvised setup. Both the setup and part of an example captured point cloud can be seen in Figure 6.3. As can be seen, the captured data needs preprocessing, resulting in a segmented point cloud of only the object. Moreover, because the output of the depth camera is in meters, even the scaling preprocessing option must be employed. Additionally, it is appropriate to draw attention to the wave-like character of the noise, which smoothens out the whole captured point cloud, degrading many details in the scene.

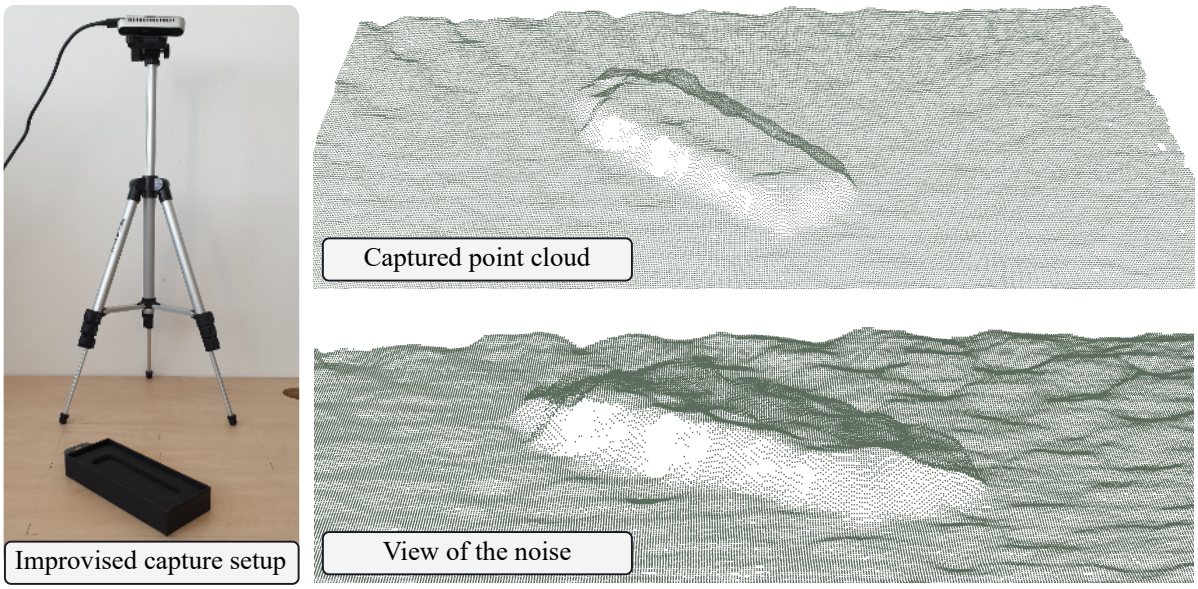

Figure 6.3: Improvised capture setup and part of an example captured data.

The depth camera was calibrated in advance using the automatic calibration algorithm provided by the Intel RealSense SW, and an artificial light source secured proper lighting conditions. Each object was placed in 24 predefined poses relative to their origin CS and captured in all of them. The ground truth transformation was estimated by capturing an RGB point cloud using the RealSense Viewer SW, from which a pattern drawn on the surface was measured and based on which object transformations were reconstructed. However, due to the noisy character of the captured point cloud, these transformations are just coarse estimates with a possible error of approximately 20 mm.

The data capture process faced numerous challenges, particularly with regard to the RealSense depth camera and its connection with the PCL. Although PCL is compatible with both ReaSense and OpenNI2, both methods of point cloud acquisition demonstrated inconsistency and slowness in the specific tried implementations. Thus, more investigation into the data acquisition problematics is needed to use the depth camera in con-

junction with the solution properly. Nevertheless, eventually, it managed to capture the already-mentioned scenes successfully. However, the successful data capture did not offer adjustable depth camera output. Thus every object position was captured both through the direct application of the RealSense SDK and via OpenNI2, each with distinct default resolutions. The RealSense SDK defaulted to 1280 × 720 (HD) resolution, whereas OpenNI2 defaulted to 640 × 480 (VGA).

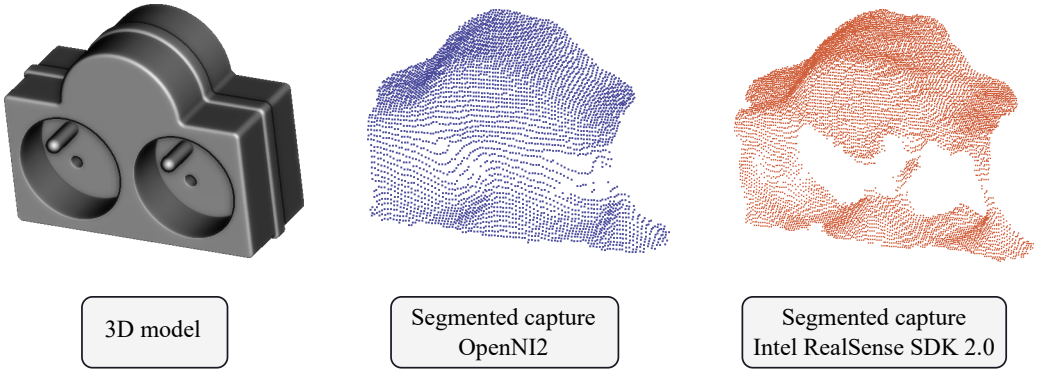

Figure 6.4: Example of segmented captured point clouds from real dataset.

Figure 6.4 shows segmented point clouds captured by both configurations. The higher-resolution configuration preserves more geometric features than the lower-resolution configuration. However, the geometric features of the captured data are very subtle due to the character of the output of the specific depth camera, regardless of the configurations.

6.2 Experiments

As already mentioned above, three types of experiments were conducted: ground truth tests, virtual noise tests, and captured tests. This section first provides more details about each type of experiment considered. Then, the conditions and execution are described in more detail. Subsequently, the tools and procedures used for data processing of the outputs are mentioned.

6.2.1 Types

Ground truth tests are characterized by the use of the same data for accuracy testing and for database creation. This means that each partial view in the database is also used for global pipeline iteration. This represents the ideal scenario, as both data are exactly the same. However, the results may not be ideal due to the inaccuracies during the solution pipeline and the ambiguous views. This type of experiment is designed to assess the capabilities of the solution core because the influence of the input data quality can be, in this case, neglected.

The virtual noise tests use the same dataset for the database creation as ground truth tests (no noise) but use partial views generated with Gaussianray noise as input for accuracy tests. Moreover, the parameters for generating virtual depth camera viewpoints differ from the database dataset to introduce more deviation between the two. This type of experiment is designed to test the capabilities of the solution on more realistic data, which will always be nonideal.

The captured tests also use the same database dataset as the two previous types of

experiments. However, the input data for accuracy tests were captured using the Intel RealSense D415 camera on an improvised capture setup (see Subsection 6.1.3). The data were captured using two configurations, which differ in capture resolution. The ground truth poses were estimated using RGB point cloud and grid pattern drawn on the table. Due to the wave-like character of the captured data, the ground truth pose estimation is quite coarse, especially in the translation factor. This type of experiment is designed to test the capabilities of the solution on real data. However, it is appropriate to note that the data from this particular model of the depth camera highly suppresses the geometrical details of captured objects (as can be seen in Figure 6.4).

6.2.2 Conditions

The parameters in the configuration files used for the experiment were fine-tuned manually by assessing the visual feedback (debug option) in a few iterations of mentioned experiment types. The aim was to achieve an acceptable balance between recognition capabilities and false positive results. It is crucial to point out that these parameters may not be optimal, and employing parameter optimization techniques could improve the results. However, this thesis did not pursue such optimization due to time constraints. Thus, the results within this chapter serve more as an indicative measure. The parameters were set to such values that were appropriate for all the experiments, only differing in a few details between the experiment types due to the various amount of noise in input data. For the exact configuration parameters used in this chapter, see Appendix D.

6.2.3 Execution

All the experiments were executed solely using the solution wrapper via the *mtsolution* module in Python using the HW and SW listed in Section 5.2. Each experiment was conducted for each object in each relevant model group and for each supported global descriptor. The main portion of the evaluation was done via the *AccuracyTest* class under hood. This means that the metrics are the correctness of recognition, based on the name of the 3D model in the database and the name provided to the *AccuracyTest*. If the recognition result is correct, the pose estimation accuracy is then specified by rotational and translational error mentioned in the Subsection 2.4.1. It is worth noting that the pose accuracy errors do not consider ambiguous views or symmetries. Moreover, the configuration was also set to save the *Extras* for additional information about each iteration.

6.2.4 Data processing

The results were processed using Python, the Pandas library 2.0.1, and the Python standard library. All the presented data were extracted from *Result* JSON, *Extras* JSON, and *Evaluation* JSON of individual online stage iterations. First, the data from individual iterations were packed together and organized by models, then by the global descriptors used, and finally, by the model group. These organized data groups were subsequently processed to compare the performance of different global descriptors and to provide more detailed results for the top-performing descriptor within each model group.

Two kinds of results are presented in this chapter. First, the recognition results present tables illustrating the recognition capabilities of each supported descriptor, sorted by the number of correct results. If it is appropriate, confusion maps for the top-performing descriptors are also illustrated. Then, the pose accuracy results present the top-performing

descriptors in each group based on the number of pose estimations with errors under a predefined threshold. This is followed by a boxplot representation of both rotational and translational errors (while excluding outliers) for the relevant groups. This comprehensive evaluation can then establish a broader understanding of the capabilities and limitations of the solution.

6.3 Results

In this section, individual results are first shown while being organized by the experiment type. This includes results related to both 3D object recognition and pose estimation. Subsequently, the results are commented on.

6.3.1 Ground truth tests

Table 6.1: Recognition results for ground truth tests on *one object* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
CVFH	42	42	100.0	0	0.0	0	0.0
ESF	42	42	100.0	0	0.0	0	0.0
VFH	42	42	100.0	0	0.0	0	0.0
OUR-CVFH	42	42	100.0	0	0.0	0	0.0
GOOD	42	42	100.0	0	0.0	0	0.0

Table 6.2: Recognition results for ground truth tests on *real objects* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
CVFH	126	125	99.21	0	0.00	1	0.79
OUR-CVFH	126	123	97.62	0	0.00	3	2.38
VFH	126	123	97.62	1	0.79	2	1.59
GOOD	126	117	92.86	3	2.38	6	4.76
ESF	126	110	87.30	11	8.73	5	3.97

Table 6.3: Recognition results for ground truth tests on *six objects* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
CVFH	252	232	92.06	6	2.38	14	5.56
OUR-CVFH	252	229	90.87	10	3.97	13	5.16
GOOD	252	206	81.75	22	8.73	24	9.52
VFH	252	205	81.35	26	10.32	21	8.33
ESF	252	167	66.27	62	24.60	23	9.13

Table 6.4: Pose errors under the specific threshold for ground truth tests of each group on best recognizing descriptor.

		Under $\varepsilon_{\text{rot}} < 15^\circ$ **and** $\varepsilon_{\text{trans}} < 15\,\text{mm}$		
Group	**Descriptor**	[-]	Of correct [%]	Of total [%]
One object	CVFH	39	92.86	92.86
Real objects	CVFH	117	93.60	92.93
Six objects	CVFH	215	92.67	85.32

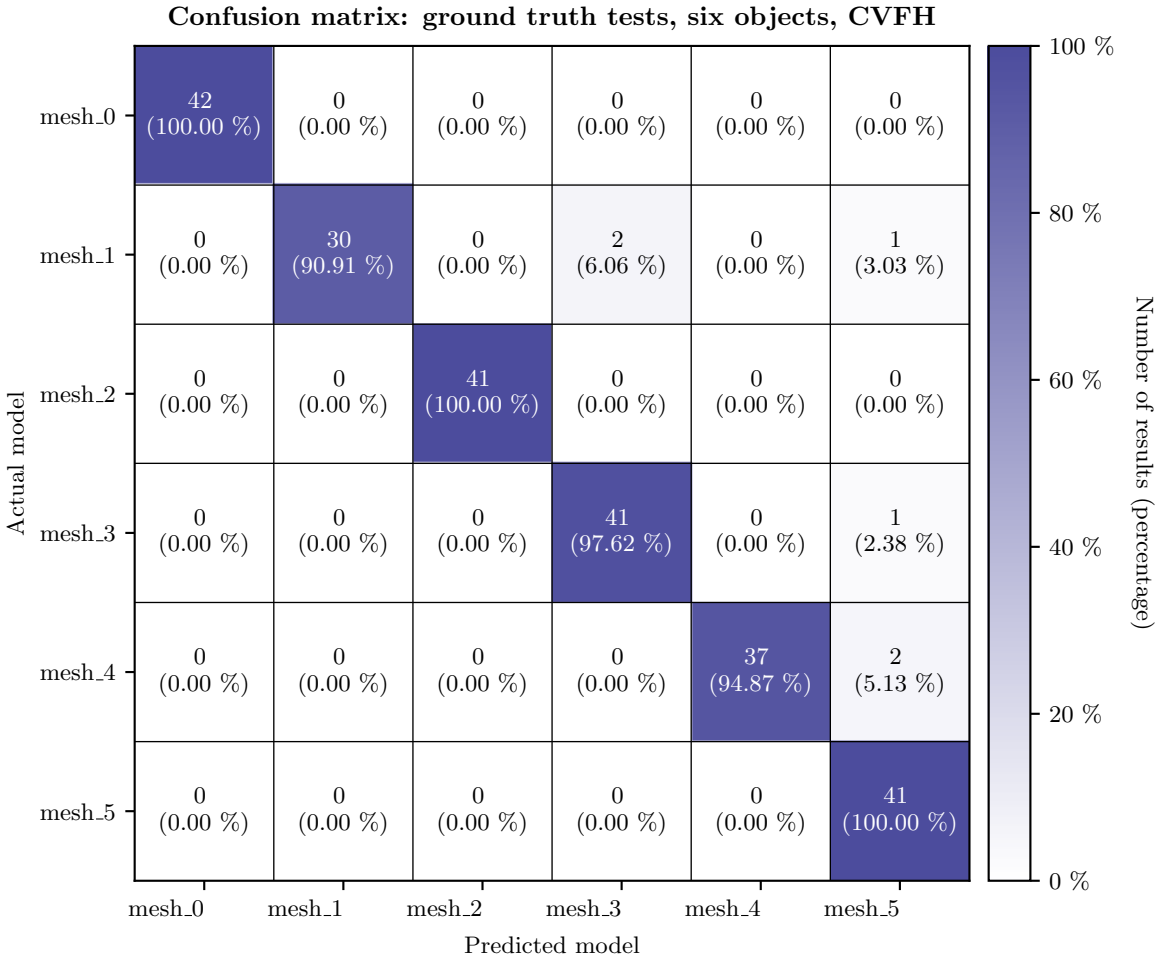

Figure 6.5: Confusion matrix of the best global descriptor in ground truth tests on *six objects* group.

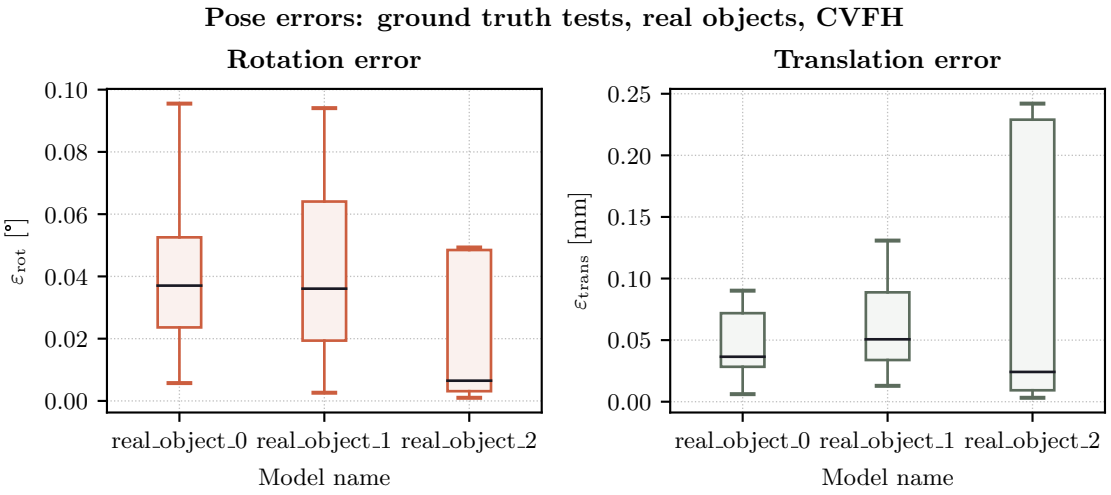

Figure 6.6: Box plot of pose errors of the best global descriptor in ground truth tests on *real objects* group.

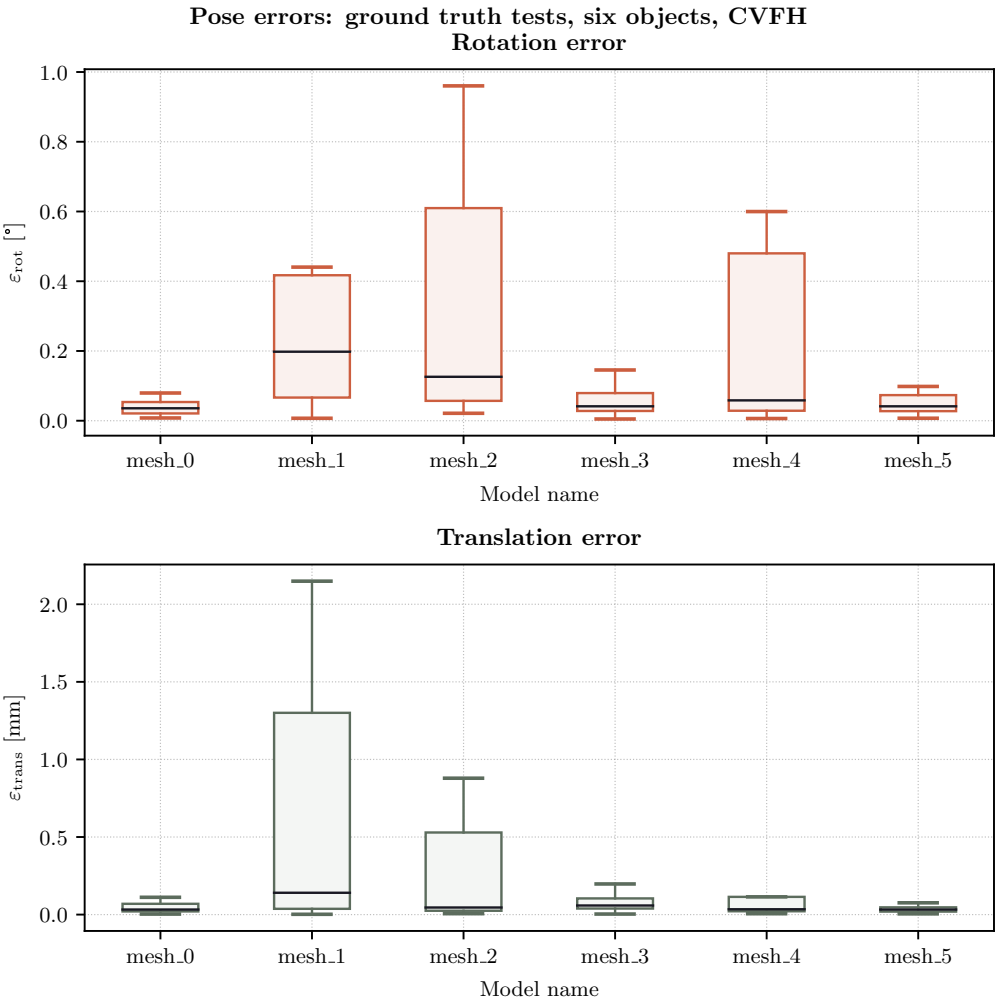

Figure 6.7: Box plot of pose errors of the best global descriptor in ground truth tests on *six objects* group.

6.3.2 Virtual noise tests

Table 6.5: Recognition results for virtual noise tests on *one object* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
CVFH	42	42	100.00	0	0.0	0	0.00
VFH	42	42	100.00	0	0.0	0	0.00
ESF	42	42	100.00	0	0.0	0	0.00
OUR-CVFH	42	41	97.62	0	0.0	1	2.38
GOOD	42	40	95.24	0	0.0	2	4.76

Table 6.6: Recognition results for virtual noise tests on *real objects* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
CVFH	126	102	80.95	12	9.52	12	9.52
GOOD	126	93	73.81	11	8.73	22	17.46
ESF	126	91	72.22	24	19.05	11	8.73
OUR-CVFH	126	87	69.05	29	23.02	10	7.94
VFH	126	76	60.32	31	24.60	19	15.08

Table 6.7: Recognition results for virtual noise tests on *six objects* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
CVFH	252	187	74.21	47	18.65	18	7.14
VFH	252	171	67.86	64	25.40	17	6.75
OUR-CVFH	252	157	62.30	75	29.76	20	7.94
GOOD	252	140	55.56	90	35.71	22	8.73
ESF	252	123	48.81	114	45.24	15	5.95

Table 6.8: Pose errors under the specific threshold for virtual noise tests of each group on best recognizing descriptor.

		Under $\varepsilon_{\text{rot}} < 15^\circ$ **and** $\varepsilon_{\text{trans}} < 15\,\text{mm}$		
Group	**Descriptor**	[-]	Of correct [%]	Of total [%]
One object	CVFH	26	61.9	61.9
Real objects	CVFH	60	58.82	47.62
Six objects	CVFH	127	67.91	50.4

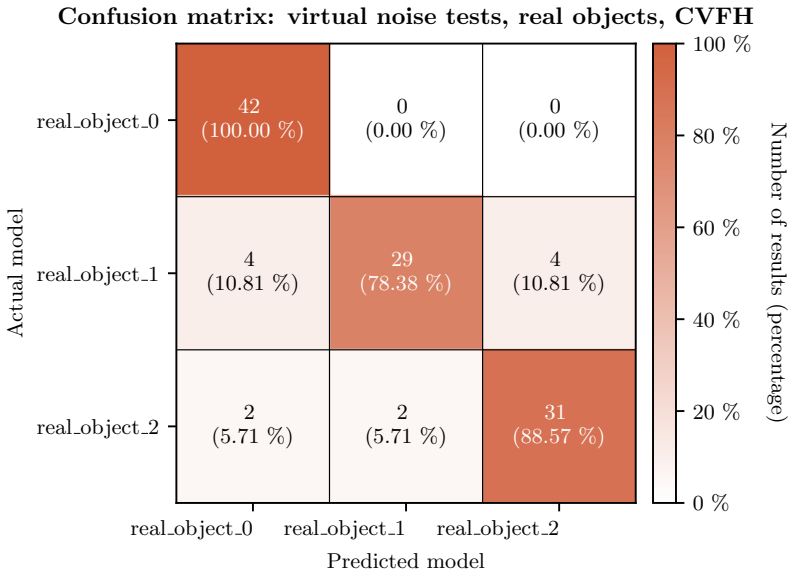

Figure 6.8: Confusion matrix of the best global descriptor in virtual noise tests on *real objects* group.

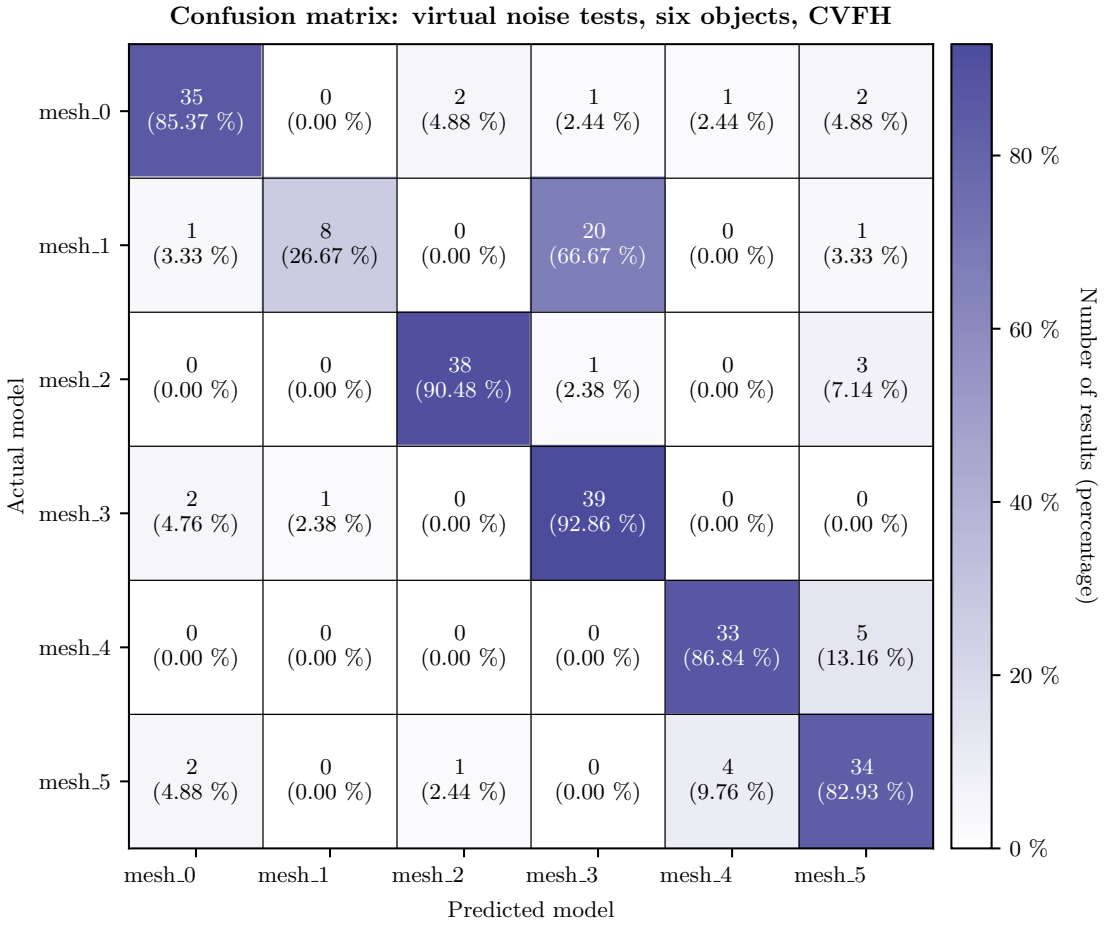

Figure 6.9: Confusion matrix of the best global descriptor in virtual noise tests on *six objects* group.

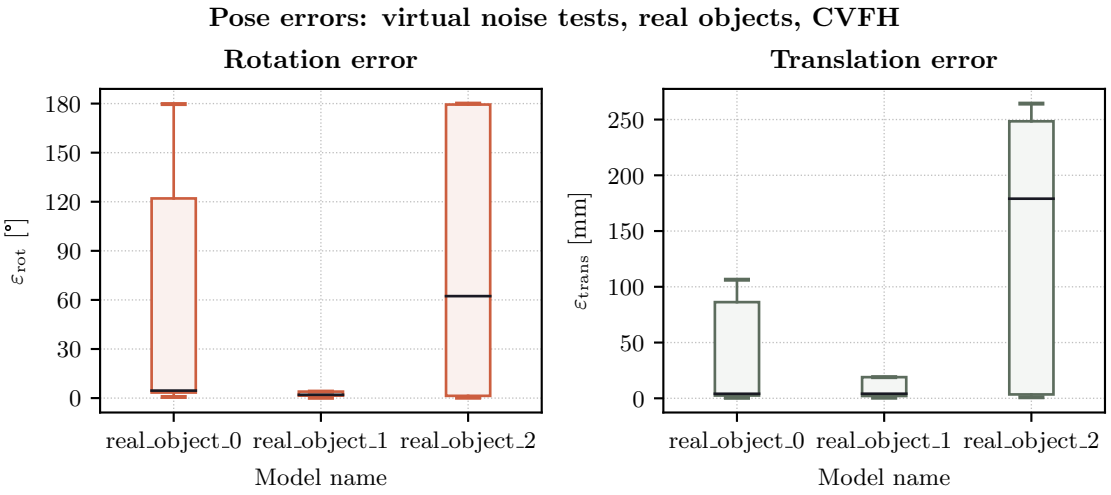

Figure 6.10: Box plot of pose errors of the best global descriptor in virtual noise tests on *real objects* group.

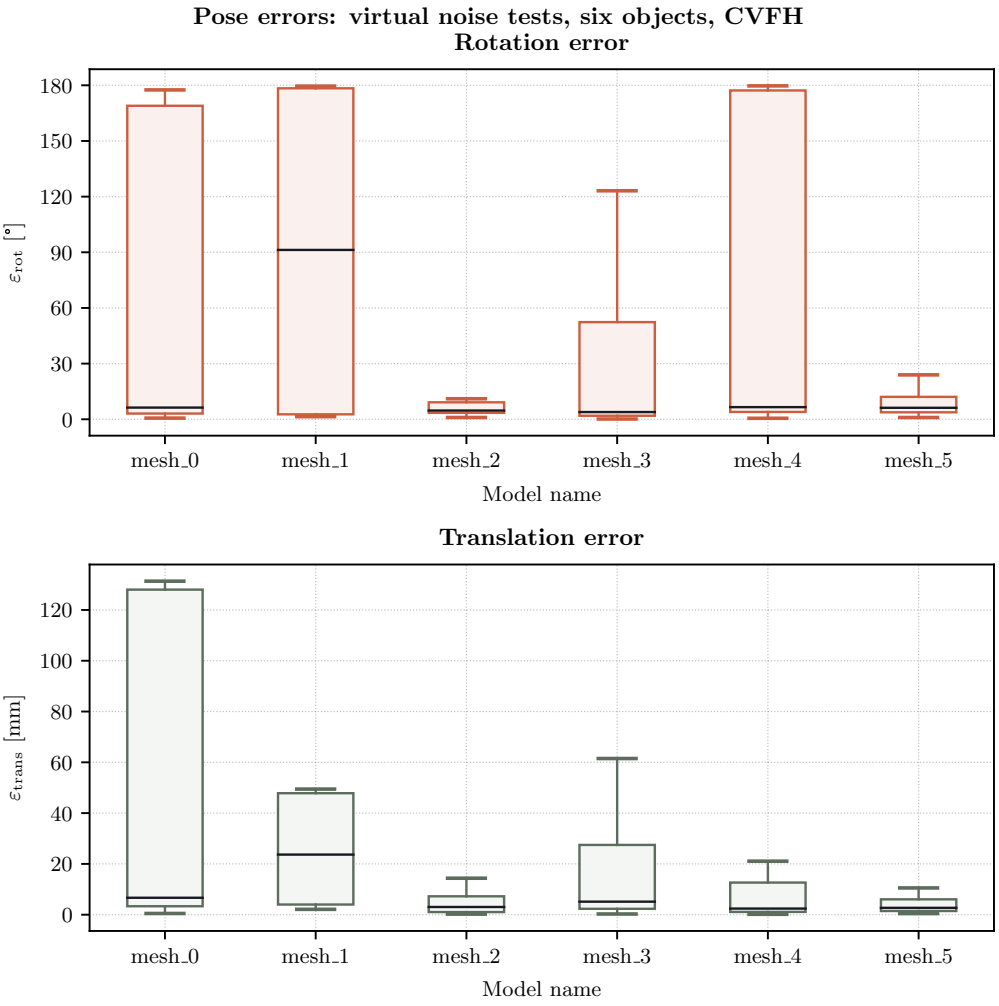

Figure 6.11: Box plot of pose errors of the best global descriptor in virtual noise tests on *six objects* group.

6.3.3 Capture tests

Table 6.9: Recognition results for capture OpennNI2 tests on *one object* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
OUR-CVFH	72	17	23.61	22	30.56	33	45.83
ESF	72	15	20.83	15	20.83	42	58.33
GOOD	72	14	19.44	11	15.28	47	65.28
CVFH	72	14	19.44	16	22.22	42	58.33
VFH	72	14	19.44	19	26.39	39	54.17

Table 6.10: Recognition results for capture RealSense SDK tests on *one object* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
VFH	72	19	26.39	21	29.17	32	44.44
OUR-CVFH	72	17	23.61	17	23.61	38	52.78
ESF	72	15	20.83	14	19.44	43	59.72
CVFH	72	15	20.83	17	23.61	40	55.56
GOOD	72	12	16.67	17	23.61	43	59.72

Table 6.11: Recognition results for capture OpennNI2 tests on *real objects* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
CVFH	72	30	41.67	22	30.56	20	27.78
VFH	72	29	40.28	14	19.44	29	40.28
OUR-CVFH	72	28	38.89	18	25.00	26	36.11
ESF	72	22	30.56	12	16.67	38	52.78
GOOD	72	19	26.39	8	11.11	45	62.50

Table 6.12: Recognition results for capture RealSense SDK tests on *real objects* group.

	Total	**Correct result**		**Incorrect result**		**No result**	
Descriptor	[-]	[-]	Of total [%]	[-]	Of total [%]	[-]	Of total [%]
VFH	72	30	41.67	10	13.89	32	44.44
OUR-CVFH	72	26	36.11	10	13.89	36	50.00
ESF	72	26	36.11	10	13.89	36	50.00
CVFH	72	26	36.11	14	19.44	32	44.44
GOOD	72	16	22.22	9	12.50	47	65.28

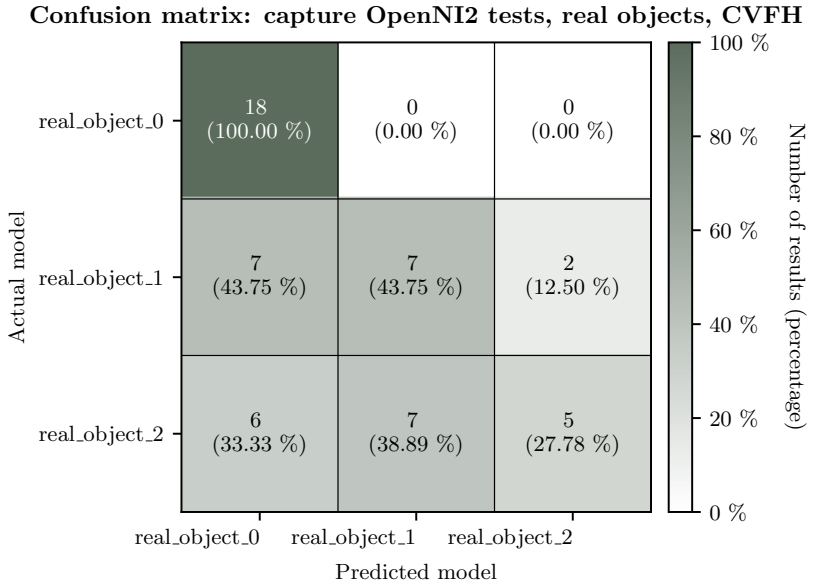

Figure 6.12: Confusion matrix of the best global descriptor in capture OpenNI2 tests on *real objects* group.

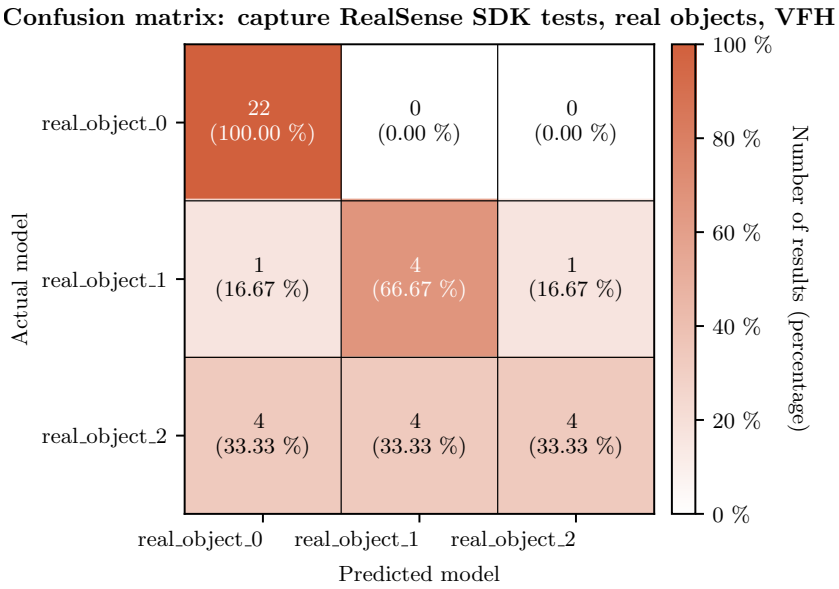

Figure 6.13: Confusion matrix of the best global descriptor in capture RealSense SDK tests on *real objects* group.

Table 6.13: Pose errors under the specific threshold for OpenNI2 capture tests of each group on best recognizing descriptor and descriptor with most poses under the threshold. Ground truth poses were just coarsely estimated.

		Under $\varepsilon_{\text{rot}} < 15^\circ$ **and** $\varepsilon_{\text{trans}} < 30\,\text{mm}$		
Group	**Descriptor**	[-]	Of correct [%]	Of total [%]
One object	OUR-CVFH	0	0	0
Real objects	CVFH	7	23.33	9.72
One object	VFH	4	28.57	5.56
Real objects	VFH	8	27.59	11.11

Table 6.14: Pose errors under the specific threshold for RealSense SDK capture tests of each group on best recognizing descriptor and descriptor with most poses under the threshold. Ground truth poses were just coarsely estimated.

		Under $\varepsilon_{\text{rot}} < 15^\circ$ and $\varepsilon_{\text{trans}} < 30\,\text{mm}$		
Group	**Descriptor**	[-]	Of correct [%]	Of total [%]
One object	VFH	2	10.53	2.78
Real objects	VFH	2	6.67	2.78
One object	GOOD	4	33.33	5.56
Real objects	ESF	7	26.92	9.72

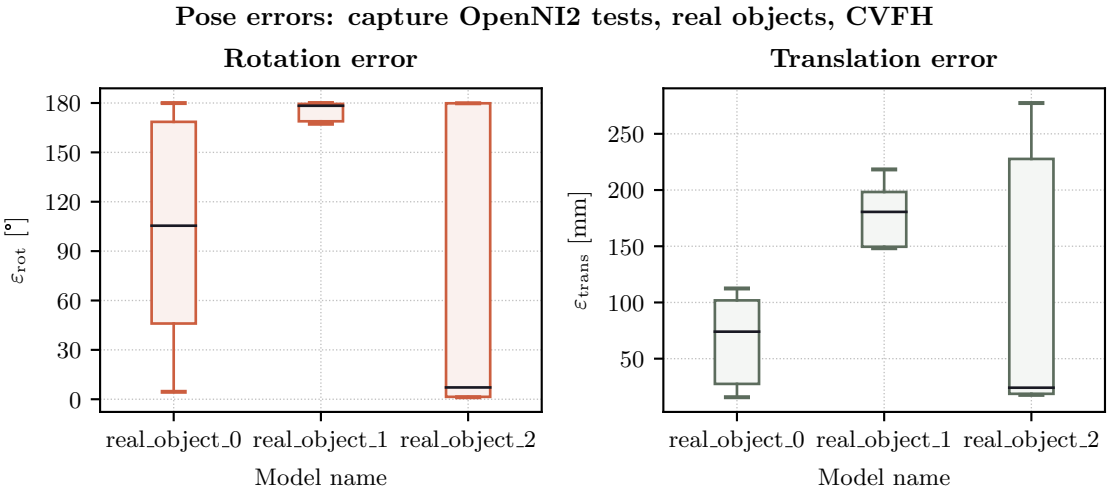

Figure 6.14: Box plot of pose errors of the best global descriptor in capture OpenNI2 tests on *real objects* group. Ground truth poses were just coarsely estimated.

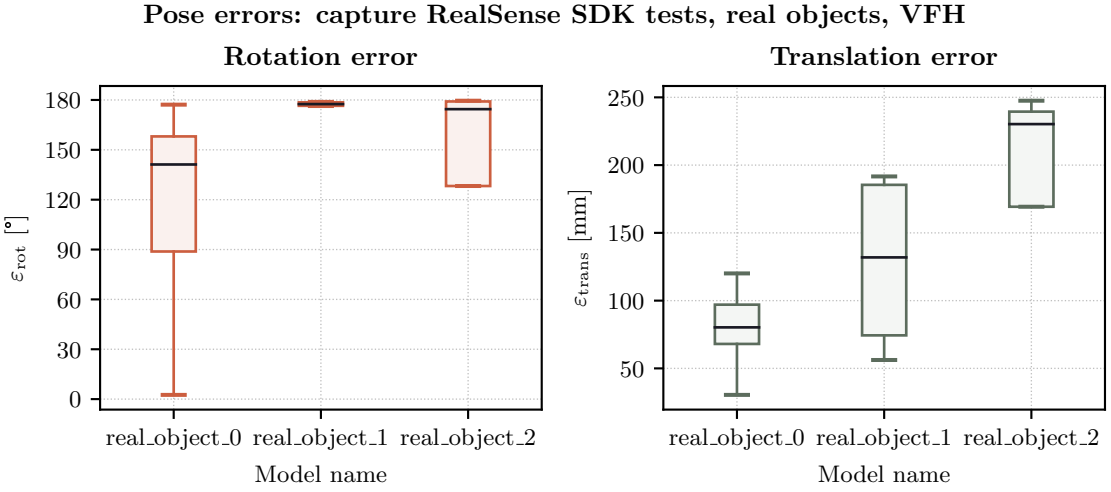

Figure 6.15: Box plot of pose errors of the best global descriptor in capture RealSense SDK tests on *real objects* group. Ground truth poses were just coarsely estimated.

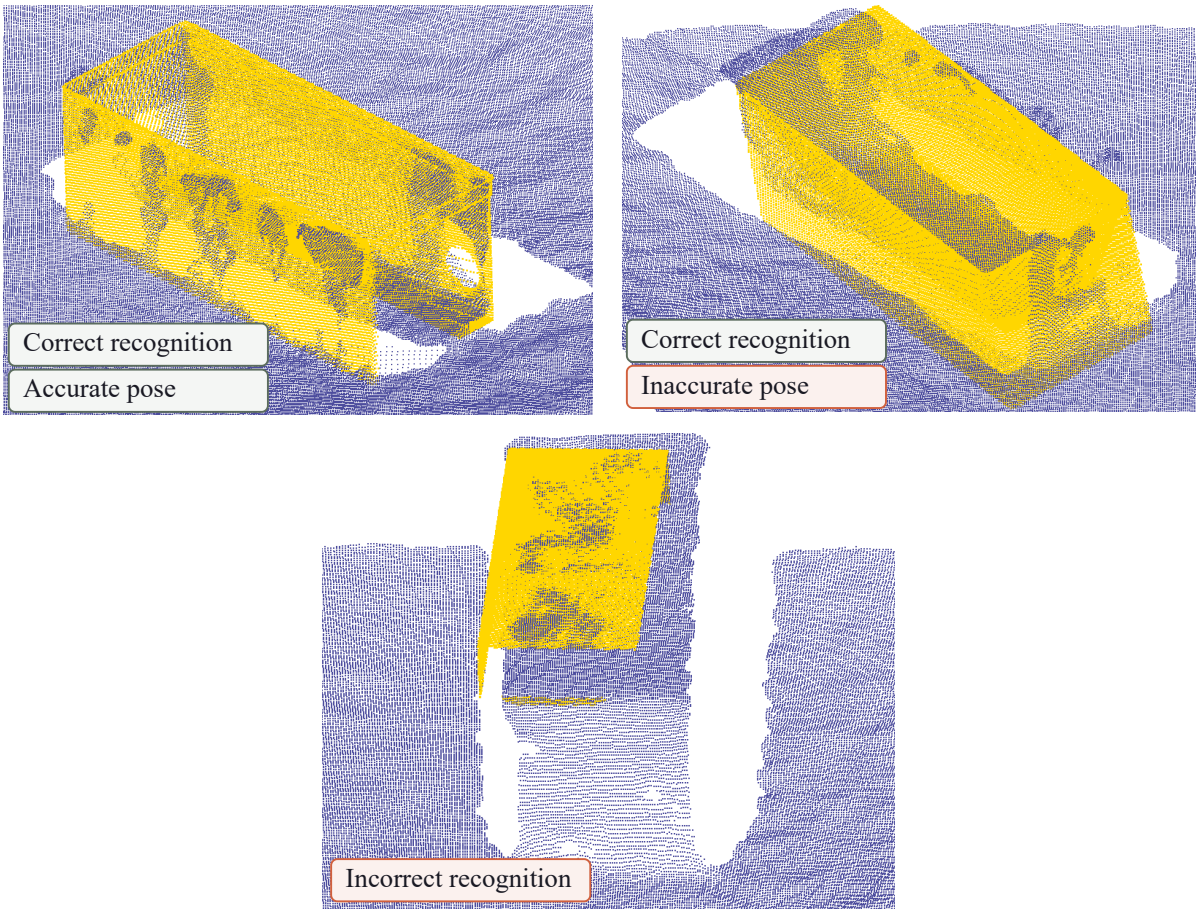

Figure 6.16: Visual examples of results in capture OpenNI2 tests on *real objects* group.

6.3.4 Commentary

As shown by the Tables 6.1, 6.2, and 6.3, which regards the ground truth tests experiment, even if the input data are ideal, with more objects in the database the recognition correctness lowers. Specifically, the percentage of correct results lowers from 100.00 % for the *one object* to 99.21 % for the *real objects*, and then to 92.06 % for the *six objects*. However, the CVFH was the most robust of supported global descriptors in such conditions. Based on the confusion matrix in Figure 6.5, it can be said that some objects are harder to recognize than others. Moreover, they are even more prone to false positives. An example of such properties is the *mesh_1* model, which has only 30 correct recognitions (out of 42 possible) and three incorrect ones. Due to the ideal input, the fine pose alignment has minimal errors, apart from a few outliers, as shown in Table 6.4. More specifically, around 93 of the correct results have pose errors under thresholds of 15 deg and 15 mm. However, some models are more prone to these pose errors than others, as shown in Figures 6.6 and 6.7.

As can be seen from trends in Tables 6.5, 6.6, and 6.7, which regards the virtual noise tests experiment, the correct recognition capabilities lowers faster, when the database and scene data differs. Specifically, the percentage of correct results lowers from 100.00 % for the *one object* to 80.95 % for the *real objects*, and then to 74.21 % for the *six objects*. Moreover, the percentage of incorrect results raised significantly between the *real objects*

with 9.52 % and the *six objects* with 18.65 %. However, the CVFH was (again) the most capable global descriptor for all model groups in this experiment. The rise of incorrect recognitions is also apparent from the confusion matrix in Figure 6.9. The *mesh_1* is not suited well for the specific tested configuration parameters or the given set of objects, as it has more incorrect results than correct ones. Both *mesh_1* and *mesh_3* have similar features, mainly their thickness, which can cause confusion between models. The other confusion matrix in Figure 6.8 shows that the model *real_object_0* has exceptional recognizability against the rest of its set. However, the remainder of the models in *real objects* group had at least four incorrect results. The pose accuracy results are also worse than in the previous experiment, which is apparent from Table 6.8. More specifically, only around 52 % of the correct results have pose errors under thresholds of 15 deg and 15 mm. This suggests that the pose refinement step falls into a bad pose, which can be either caused by a bad coarse pose or by the character of the deformed data. As shown in Figures 6.10 and 6.11, the median pose error of some object are low, while another object, like, for example, the *real_object_2*, tends to be more susceptible to it. This can be caused by ambiguous views, as there is only a fine detail (hole on the side) describing the unique pose of the object.

The results in Tables 6.9 and 6.14 should be reviewed differently than in the previous experiments, as the tests for the *one object* group involves all the captured scenes and not just the ones containing the object. The capabilities of best-performing global descriptors in such cases are similar, with around 30 % of incorrect results and 45 % of no results. However, those best-performing global descriptors are not the same. For the lower resolution (OpenNI2) capture, the best performing global descriptor was OUR-CVFH, while for the higher resolution (RealSense SDK) capture, it was the VFH. A similar occurrence is apparent from Table 6.11 and 6.12, where the best performing global descriptor for lower resolution capture is CVFH, with 41.67 % of correct results and 30.56 % of incorrect results, meanwhile for the higher resolution capture it is again the VFH, with 41.67 % of correct results and only 13.89 % of incorrect results. This may be related to the differences in the character of the captured data between the two, which can be seen in Figure 6.4. Based on both confusion matrices in Figures 6.12 and 6.13, it is apparent that the used configuration parameters do not filter the results enough, as there are many incorrect recognitions. More specifically, apart from the *real_object_0* model, all models have unconvincing results. As the ground truth poses for this experiment were only coarsely estimated (see Subsection 6.1.3), the following results should be taken with a grain of salt. However, for this reason, the translational error threshold was raised from 15 mm to 30 mm. The pose accuracy results for the best-performing global descriptors for the recognition part have, in the case of the lower resolution capture, 23.33 % of correct results under pose thresholds of 15 deg and 30 mm, and in the case of the higher resolution capture, only 6.67 % of correct results under the same threshold. As shown in Figures 6.14 and 6.15, the vast majority of the poses have significant errors, the best object in this regard is the *real_object_2* in case of the lower resolution capture and CVFH descriptor. Moreover, in this type of experiment, the best-performing global descriptor in the recognition part is not the best-performing regarding the accuracy of the resulting pose. The most successful results on *real objects* group in the capture tests experiment overall were obtained using the VFH global descriptor used on the lower resolution (OpenNI2) capture. More specifically, 40.28 % of correct results and 19.44 % of incorrect results out of 72 test scenes regarding the recognition, while 27.59 % of the correctly recognized

results are under the pose error thresholds of 15 deg and 30 mm. These results are still respectable, considering the lack of geometrical details in the captured data. A visual example of results from each main category of results can be seen in Figure 6.16.

6.4 Assesment of the solution

This section assesses the solution following its successful validation and the commentary of the results in Section 6.3. First, additional information about computation time and storage requirements is provided. Then, the key observations are presented, followed by identifying the limitations. Finally, suggestions for improving the solution and future work are proposed.

By disabling architecture optimizations and not implementing additional threading in the solution, the assessment of the computation performance is highly distorted. Moreover, it heavily depends on the used HW and solution configuration. However, to provide at least a few indicative values, the capture OpenNI2 tests experiment using VFH was selected as the source of the following data, as it is closest to the real usage scenario. One iteration of the online stage took an average of 1.83 seconds. Around 65 % of that time was taken by fine alignment of the candidates. Another 12 % was taken up by scene preprocessing, including scaling of the point cloud. This is followed by 7 % on coarse alignment, 6 % on loading data of the candidates, and 6 % on scene normal estimation.

As for the storage requirements, the created database in the case of the *real objects* model group, with the specified configuration, takes up 165.3 MiB of storage space, thus 55.1 MiB per object. Space taken by iteration result varies based on the *debug* options. In the most compact form, only the *Result* structure is saved and takes up only 4 KiB of storage space per iteration. The optional *Extras* structure takes up an additional 24 KiB. The storage space taken up by the computed descriptors varies based on the used global descriptor.

A key observation based on the conducted experiments is that the success of the solution pipeline is tied to the specific set of objects within the database and the parameters of the configuration files. Moreover, some objects may demonstrate lower success rates and more incorrect results than others. Furthermore, shell objects resembling boxes may lose some geometric features due to plane removal segmentation, which affects global descriptors that do not create sub-clusters. Another observation is that smaller objects can be falsely recognized in larger objects with the same local features, as seen in Figure 6.16.

Moving on to the limitations. The coarse pose estimation often produces a suboptimal initial pose for fine alignment. Ambiguate views are not identified in the database and cause part of the suboptimal coarse alignment cases. With a larger radius for normal estimation, the minimal possible difference between database partial view normals and the normals estimated on the input scene enlarges, resulting in considerably different descriptions even though both point clouds may be identical. It is clear from the results that the configuration parameters used were suboptimal. Thus, ideally, the parameters should be tuned to the specific character of the input. Moreover, it would be insightful to conduct an experiment with a different depth camera, which possibly preserves more geometrical details, than the one used in this thesis.

Building upon these observations, several potential improvements can be made. Modifying the pose alignment and hypothesis verification stages to utilize more robust algorithms could significantly benefit the robustness of the solution. Ambiguous views should

ideally be identified and treated in a specialized manner, including different behavior in accuracy tests. The issue of different normals information due to the normal estimation parameters could be mitigated by not retrieving the normals from the mesh model when creating the dataset but by estimating them the same way as in the online stage. Although this might enhance both the recognition and pose estimation accuracy, it could simultaneously degrade the level of detail for all models in the database, possibly making them too similar. Enhancing the build configuration and implementation details could lead to enabling architecture-based optimizations, which can significantly boost the computational efficiency of the whole solution. Concurrently, employing threading for candidate evaluation would offer further performance enhancements.

As for future work, several promising directions are evident. Firstly, implementing a local descriptor pipeline would provide a relevant comparison of the two competing handmade methods. It could also be beneficial to compare this solution with learning-based methods or explore a hybrid approach integrating both approaches. Further integration of the solution with planning and robotic control applications, particularly in pick-and-place tasks, could provide more insight into the needs of a practical solution. Last but not least, exploring the usage of a moving depth camera to help mitigate ambiguate views should also be considered.

7 Conclusion

The primary goal of this thesis was to select and validate an algorithm for the recognition and pose estimation of objects, relying solely on depth camera data and 3D models as references.

Following the initial literature review, the focus was refined to hand-made feature-based methods for 3D object recognition, a coarse-to-fine approach to pose estimation, and the utilization of the Point Cloud Library. A general pipeline using global descriptors was extracted based on a more in-depth literature review. Based on this pipeline, a solution was proposed, resulting in a C++ library composed of modules, and a wrapper, that provides an easy interface for users. This wrapper was extended to a Python module, offering even greater flexibility in its usage. Moreover, the entire solution pipeline is configured using JSON files, enabling easy modification with any text editor. The developed solution offers various functionalities, from generating partial view point clouds to testing the accuracy of object recognition and pose estimation. This solution was validated through extensive experiments involving both artificial and real data and multiple sets of models.

These experiments demonstrated that the success of the solution is dependent not only on the particular configuration of the solution pipeline but also on the specific set of objects in the database. As the number of objects in a database increased, the number of correct recognitions decreased. Moreover, some objects demonstrated lower success rates and a higher occurrence of incorrect results. Although the real captured data considerably suppressed geometric details of the objects, the solution resulted in 40.28 % of correct recognitions and 19.44 % of incorrect recognitions out of 72 test scenes for a database containing three objects. However, just 27.59 % of the correct recognitions had accurate poses. Results on the artificial data with noise were much better. For example, tests on a database containing six objects resulted in 74.21 % of correct recognitions and 18.65 % of incorrect recognitions out of 252 test scenes, while 67.91 % of the correct recognitions had accurate poses. The relatively high number of inaccurate poses is caused by the limited robustness of the used pose estimation algorithms and the existence of ambiguous views. Another limitation is the need for fine-tuning the configuration parameters based on the specifics of the input data.

Regarding future work. It would be insightful to test the solution on a better depth camera, which does not suppress geometrical features as much as the one used for experiments in this thesis. However, there are also many possible advancements regarding the current pipeline. These include improving coarse and fine alignment steps using more robust algorithms, or optimizing the implementation, especially regarding candidate evaluation. Another interesting insight could potentially arise from implementing a pipeline with local descriptors, learning-based methods, or a hybrid approach and comparing results with the already implemented global descriptor pipeline.

Bibliography

[1] Szeliski, R. *Computer Vision*. 2nd ed. Cham: Springer International Publishing, 2022 [cit. 2022-05-10]. ISBN 978-3-030-34371-2. Available at: DOI: 10.1007/978-3-030-34372-9.

[2] Kelly, R. and Chakravorty, D. *The Most Common 3D File Formats in 2022: So Many File Formats, So Little Time!* [Munich]: All3DP, 2022 [cit. 2022-05-01]. Available at: https://all3dp.com/2/most-common-3d-file-formats-model/.

[3] *The STL File Format – Simply Explained: Standard Tessellation Language*. [Munich]: All3DP, 2021 [cit. 2022-05-01]. Available at: https://all3dp.com/1/stl-file-format-3d-printing/.

[4] Liu, S., Zhang, M., Kadam, P. and Kuo, C.-C. J. *3D Point Cloud Analysis: Traditional, Deep Learning, and Explainable Machine Learning Methods*. 1st ed. Cham: Springer International Publishing, 2021 [cit. 2022-05-01]. ISBN 978-3-030-89179-4. Available at: DOI:10.1007/978-3-030-89180-0.

[5] *Depth Sensing Technologies Overview: Choose the Best one for Your Application*. [Taufkirchen]: FRAMOS, c2022 [cit. 2022-05-03]. Available at: https://www.framos.com/en/products-solutions/3d-depth-sensing/depth-sensing-technologies.

[6] Carvalho, L. E. and Wangenheim, A. von. 3D object recognition and classification: a systematic literature review. *Pattern Analysis and Applications*. 2019, vol. 22, no. 4, p. 1243–1292, [cit. 2022-05-05]. DOI: 10.1007/s10044-019-00804-4. ISSN 1433-7541. Available at: http://link.springer.com/10.1007/s10044-019-00804-4.

[7] Hodaň, T., Sundermeyer, M., Drost, B., Labbé, Y., Brachmann, E. et al. BOP Challenge 2020 on 6D Object Localization. Cham: Springer International Publishing. 2020, vol. 12536, p. 577–594, [cit. 2022-05-08]. DOI: 10.1007/978-3-030-66096-3_39. Available at: https://link.springer.com/10.1007/978-3-030-66096-3_39.

[8] Aldoma, A., Marton, Z.-C., Tombari, F., Wohlkinger, W., Potthast, C. et al. Tutorial: Point Cloud Library. *IEEE Robotics & Automation Magazine*. 2012, vol. 19, no. 3, p. 80–91, [cit. 2022-05-07]. DOI: 10.1109/MRA.2012.2206675. ISSN 1070-9932. Available at: https://ieeexplore.ieee.org/document/6299166/.

[9] Koch, S., Matveev, A., Jiang, Z., Williams, F., Artemov, A. et al. ABC: A Big CAD Model Dataset for Geometric Deep Learning. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2019, p. 9593–9603 [cit. 2022-05-08]. DOI: 10.1109/CVPR.2019.00983. ISBN 978-1-7281-3293-8. Available at: https://ieeexplore.ieee.org/document/8954378/.

[10] Wu, R., Xiao, C. and Zheng, C. DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, 2021, p. 6752–6762 [cit. 2022-05-08]. DOI: 10.1109/ICCV48922.2021.00670. ISBN 978-1-6654-2812-5. Available at: https://ieeexplore.ieee.org/document/9710909/.

[11] Marchand, E., Uchiyama, H. and Spindler, F. Pose Estimation for Augmented Reality: A Hands-On Survey. *IEEE Transactions on Visualization and Computer Graphics*. 2016-12-1, vol. 22, no. 12, p. 2633–2651, [cit. 2022-05-08]. DOI: 10.1109/TVCG.2015.2513408. ISSN 1077-2626. Available at: http://ieeexplore.ieee.org/document/7368948/.

[12] Mortenson, M. E. *Geometric transformations for 3D modeling*. 2nd ed. New York: Industrial Press, 2007. ISBN 978-0-8311-3338-2.

[13] Hodaň, T., Matas, J. and Obdržálek Štěpán. On Evaluation of 6D Object Pose Estimation. In: *Computer Vision – ECCV 2016 Workshops*. Cham: Springer International Publishing, 2016, p. 606–619 [cit. 2022-05-09]. DOI: 10.1007/978-3-319-49409-8_52. ISBN 978-3-319-49408-1. Available at: http://link.springer.com/10.1007/978-3-319-49409-8_52.

[14] Huynh, D. Q. Metrics for 3D Rotations: Comparison and Analysis. *Journal of Mathematical Imaging and Vision*. 2009, vol. 35, no. 2, p. 155–164, [cit. 2022-05-09]. DOI: 10.1007/s10851-009-0161-2. ISSN 0924-9907. Available at: http://link.springer.com/10.1007/s10851-009-0161-2.

[15] *OpenCV: OpenCV Modules*. 2022 [cit. 2022-05-12]. Available at: https://docs.opencv.org/4.x/.

[16] *Open3D: A Modern Library for 3D Data Processing*. C2018-2021 [cit. 2022-05-12]. Available at: http://www.open3d.org/docs/release/.

[17] *Point Cloud Library (PCL): PCL API Documentation*. 2022 [cit. 2022-05-12]. Available at: https://pointclouds.org/documentation/index.html.

[18] *PCL/OpenNI tutorial 5: 3D object recognition (pipeline)*. [León]: [Robotics Group of the University of León], 2015 [cit. 2022-05-13]. Available at: https://robotica.unileon.es/index.php?title=PCL/OpenNI_tutorial_5:_3D_object_recognition_(pipeline).

[19] Han, X.-F., Sun, S.-J., Song, X.-Y. and Xiao, G.-Q. *3D Point Cloud Descriptors in Hand-crafted and Deep Learning Age: State-of-the-Art*. arXiv, 2018 [cit. 2022-05-13]. DOI: 10.48550/ARXIV.1802.02297. Available at: https://arxiv.org/abs/1802.02297.

[20] Himri, Ridao and Gracias. 3D Object Recognition Based on Point Clouds in Underwater Environment with Global Descriptors: A Survey. *Sensors*. 2019, vol. 19, no. 20, [cit. 2022-05-13]. DOI: 10.3390/s19204451. ISSN 1424-8220. Available at: https://www.mdpi.com/1424-8220/19/20/4451.

[21] Li, P., Wang, R., Wang, Y. and Tao, W. Evaluation of the ICP Algorithm in 3D Point Cloud Registration. *IEEE Access*. 2020, vol. 8, p. 68030–68048, [cit. 2022-05-14]. DOI: 10.1109/ACCESS.2020.2986470. ISSN 2169-3536. Available at: https://ieeexplore.ieee.org/document/9060927/.

[22] Li, M. and Hashimoto, K. Fast and Robust Pose Estimation Algorithm for Bin Picking Using Point Pair Feature. In: *2018 24th International Conference on Pattern Recognition (ICPR)*. Beijing: IEEE, 2018, p. 1604–1609 [cit. 2022-05-14]. DOI: 10.1109/ICPR.2018.8545432. ISBN 978-1-5386-3788-3. Available at: https://ieeexplore.ieee.org/document/8545432/.

[23] Li, D., Wang, H., Liu, N., Wang, X. and Xu, J. 3D Object Recognition and Pose Estimation From Point Cloud Using Stably Observed Point Pair Feature. *IEEE Access*. 2020, vol. 8, p. 44335–44345, [cit. 2022-05-14]. DOI: 10.1109/ACCESS.2020.2978255. ISSN 2169-3536. Available at: https://ieeexplore.ieee.org/document/9024052/.

[24] Hanh, L. D. and Hieu, K. T. G. 3D matching by combining CAD model and computer vision for autonomous bin picking. *International Journal on Interactive Design and Manufacturing (IJIDeM)*. 2021, vol. 15, 2-3, p. 239–247, [cit. 2022-05-14]. DOI: 10.1007/s12008-021-00762-4. ISSN 1955-2513. Available at: https://link.springer.com/10.1007/s12008-021-00762-4.

[25] Wang, N., Lin, J., Zhang, X. and Zheng, X. Fast and Robust Object Pose Estimation Based on Point Pair Feature for Bin Picking. In: *2021 27th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*. Shanghai: IEEE, 2021-11-26, p. 528–533 [cit. 2022-05-14]. DOI: 10.1109/M2VIP49856.2021.9664997. ISBN 978-1-6654-3153-8. Available at: https://ieeexplore.ieee.org/document/9664997/.

[26] Drost, B., Ulrich, M., Navab, N. and Ilic, S. Model globally, match locally: Efficient and robust 3D object recognition. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. San Francisco: IEEE, 2010, p. 998–1005 [cit. 2022-05-14]. DOI: 10.1109/CVPR.2010.5540108. ISBN 978-1-4244-6984-0. Available at: http://ieeexplore.ieee.org/document/5540108/.

[27] Hodaň, T., Michel, F., Brachmann, E., Kehl, W., Glent Buch, A. et al. BOP: Benchmark for 6D Object Pose Estimation. *European Conference on Computer Vision (ECCV)*. 2018, [cit. 2022-05-13].

[28] Li, D., Liu, N., Guo, Y., Wang, X. and Xu, J. 3D object recognition and pose estimation for random bin-picking using Partition Viewpoint Feature Histograms. *Pattern Recognition Letters*. 2019, vol. 128, p. 148–154, [cit. 2022-05-14]. DOI: 10.1016/j.patrec.2019.08.016. ISSN 01678655. Available at: https://linkinghub.elsevier.com/retrieve/pii/S0167865518304501.

[29] Liang, X. and Cheng, H. RGB-D Camera based 3D Object Pose Estimation and Grasping. In: *2019 IEEE 9th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*. IEEE, 2019,

p. 1279–1284 [cit. 2022-05-14]. DOI: 10.1109/CYBER46603.2019.9066550. ISBN 978-1-7281-0770-7. Available at: https://ieeexplore.ieee.org/document/9066550/.

[30] Hu, H., Gu, W., Yang, X., Zhang, N. and Lou, Y. Fast 6D object pose estimation of shell parts for robotic assembly. *The International Journal of Advanced Manufacturing Technology*. 2022, vol. 118, 5-6, p. 1383–1396, [cit. 2022-05-14]. DOI: 10.1007/s00170-021-07960-0. ISSN 0268-3768. Available at: https://link.springer.com/10.1007/s00170-021-07960-0.

[31] Kasaei, S. H., Lopes, L. S., Tome, A. M. and Oliveira, M. *GOOD: A Global Orthographic Object Descriptor for 3D Object Recognition and Manipulation*. GitHub, 2017. Available at: https://github.com/SeyedHamidreza/GOOD_descriptor.

[32] *PCL/OpenNI tutorial 4: 3D object recognition (descriptors)*. [León]: [Robotics Group of the University of León], 2015 [cit. 2022-05-13]. Available at: https://robotica.unileon.es/index.php?title=PCL/OpenNI_tutorial_4:_3D_object_recognition_(descriptors).

[33] Rusu, R. B., Bradski, G., Thibaux, R. and Hsu, J. Fast 3D recognition and pose using the Viewpoint Feature Histogram. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Taipei: IEEE, 2010, p. 2155–2162 [cit. 2022-05-16]. DOI: 10.1109/IROS.2010.5651280. ISBN 978-1-4244-6674-0. Available at: http://ieeexplore.ieee.org/document/5651280/.

[34] Aldoma, A., Vincze, M., Blodow, N., Gossow, D., Gedikli, S. et al. CAD-model recognition and 6DOF pose estimation using 3D cues. In: *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. Barcelona: IEEE, 2011, p. 585–592 [cit. 2022-05-16]. DOI: 10.1109/ICCVW.2011.6130296. ISBN 978-1-4673-0063-6. Available at: http://ieeexplore.ieee.org/document/6130296/.

[35] Aldoma, A., Tombari, F., Rusu, R. B. and Vincze, M. OUR-CVFH – Oriented, Unique and Repeatable Clustered Viewpoint Feature Histogram for Object Recognition and 6DOF Pose Estimation. In: *Pattern Recognition*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 113–122 [cit. 2022-05-15]. DOI: 10.1007/978-3-642-32717-9_12. ISBN 978-3-642-32716-2. Available at: http://link.springer.com/10.1007/978-3-642-32717-9_12.

[36] Wohlkinger, W. and Vincze, M. Ensemble of shape functions for 3D object classification. In: *2011 IEEE International Conference on Robotics and Biomimetics*. Karon Beach: IEEE, 2011, p. 2987–2992 [cit. 2022-05-16]. DOI: 10.1109/ROBIO.2011.6181760. ISBN 978-1-4577-2138-0. Available at: http://ieeexplore.ieee.org/document/6181760/.

[37] Kasaei, S. H., Tomé, A. M., Lopes, L. S. and Oliveira, M. GOOD: A global orthographic object descriptor for 3D object recognition and manipulation. *Pattern Recognition Letters*. 2016, vol. 83, p. 312–320, [cit. 2022-05-16]. DOI: 10.1016/j.patrec.2016.07.006. ISSN 01678655. Available at: https://linkinghub.elsevier.com/retrieve/pii/S0167865516301684.

[38] Kasaei, S. H., Lopes, L. S., Tome, A. M. and Oliveira, M. An orthographic descriptor for 3D object learning and recognition. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Daejeon: IEEE, 2016, p. 4158–4163 [cit. 2022-05-16]. DOI: 10.1109/IROS.2016.7759612. ISBN 978-1-5090-3762-9. Available at: http://ieeexplore.ieee.org/document/7759612/.

[39] *Point Cloud Library*. GitHub, 2013. Available at: https://github.com/PointCloudLibrary/pcl#point-cloud-library.

[40] *Point Cloud Library: The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing.* [cit. 2022-05-19]. Available at: https://pointclouds.org/.

[41] *PCL/OpenNI tutorial 2: Cloud processing (basic)*. [León]: [Robotics Group of the University of León], 2015 [cit. 2022-05-13]. Available at: https://robotica.unileon.es/index.php?title=PCL/OpenNI_tutorial_2:_Cloud_processing_(basic).

[42] *PCL/OpenNI tutorial 3: Cloud processing (advanced)*. [León]: [Robotics Group of the University of León], 2015 [cit. 2022-05-13]. Available at: https://robotica.unileon.es/index.php?title=PCL/OpenNI_tutorial_3:_Cloud_processing_(advanced).

[43] Rusinkiewicz, S. A symmetric objective function for ICP. *ACM Transactions on Graphics*. 2019, vol. 38, no. 4, p. 1–7, [cit. 2022-05-19]. DOI: 10.1145/3306346.3323037. ISSN 0730-0301. Available at: https://dl.acm.org/doi/10.1145/3306346.3323037.

[44] *Boost C++ Libraries*. Boost, 2022 [cit. 2023-05-19]. Available at: https://www.boost.org/.

[45] *OpenNi 2 Downloads and Documentation*. Occipital, c2022 [cit. 2022-05-19]. Available at: https://structure.io/openni.

[46] *Eigen*. 2022 [cit. 2022-05-19]. Available at: https://eigen.tuxfamily.org/index.php?title=Main_Page.

[47] *VTK: The Visualization Toolkit*. [Kitware] [cit. 2022-05-19]. Available at: https://vtk.org/.

[48] Rusu, R. B. and Cousins, S. 3D is here: Point Cloud Library (PCL). In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, p. 1–4. ISBN 9781612843865.

[49] Jakob, W., Rhinelander, J. and Moldovan, D. *Pybind11 - Seamless operability between C++11 and Python*. GitHub, 2017 [cit. 2023-05-13]. Available at: https://github.com/pybind/pybind11.

[50] Lohmann, N. *JSON for Modern C++*. GitHub, 2020 [cit. 2023-05-13]. Available at: https://github.com/nlohmann.

[51] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*. 2014, vol. 2014, no. 239, p. 2.

List of abbreviations

CAD Computer-Aided Design

CRH Camera Roll Histogram

CRH Camera Roll Histogram

CS Coordinate System

CVFH Clustered Viewpoint Feature Histogram

ESF Ensemble of Shape Functions

FLANN Fast Library for Approximate Nearest Neighbors

FOV Field Of View

FPFH Fast Point Feature Histogram

GOOD Global Orthographic Object Descriptor

ICP Iterative Closest Point

ID Identification number

JSON JavaScript Object Notation

NNS Nearest-Neighbors Search

OUR-CVFH Oriented Unique and Repeatable Clustered Viewpoint Feature Histogram

PCA Principal Component Analysis

PCD Point Cloud Data

PCL Point Cloud Library

RANSAC RANdom Sample Consensus

ROI Region Of Interest

STEP Standard for the Exchange of Product Data

STL STereo Litography

VFH Viewpoint Feature Histogram

List of Figures

List of Tables

A Attached files

/
- `solution/` ... folder all source files regarding the solution
  - `configs/` ... folder including sample input JSON for the solution pipeline
  - `examples/` ... folder examples of both in Python and C++
    - `example.py` ... example usage of the solution wrapper in Python
  - `include/` ... folder containing mostly header files
    - `dataset_preparation.h` ... headers file of the Dataset preparation module
    - `global_pipeline.h` ... headers file of the Global pipeline module
    - `solution.h` ... headers file of the solution wrapper
    - `thirdparty/` ... folder containing modified third party code
      - `good.hpp` ... modified GOOD for use in the solution
    - `utilities.hpp` ... headers file of the Utilities module
  - `src/` ... folder containing cpp files
    - `dataset_preparation.cpp` ... source code of the Dataset preparation module
    - `global_pipeline.cpp` ... source code of the Global pipeline module
    - `pybind.cpp` ... source code of the Pybind of solution wrapper
    - `solution.cpp` ... source code of the solution wrapper
    - `utilities.cpp` ... source code of the Utilities module
  - `CMakeLists.txt` ... CMakeLists file for building the project
- `data/` ... folder including 3D models and data examples
- `docker/` ... folder including Docker image recepie
  - `build_image.sh` ... shell script for building the image based on the Dockerfile
  - `Dockerfile` ... Dockerfile for the creation of an image that can run the solution
- `experiments/` ... folder regarding experiments
  - `configs/` ... folder including all used JSON files for the experiments
- `build_solution.sh` ... shell script for building the solution
- `run_docker.sh` ... shell script for building the solution.

B Instructions

To run/develop the solution, a Linux OS, preferably Ubuntu or related distributions, is required. First, the Docker image must be built to create a stable environment for the solution to compile. The procedure is as follows:

1. Install the Docker.
2. Extract the contents of the attachments folder to a folder on your disk.
3. Via terminal, change directory into the *attachments/docker* folder.
4. Run the *build_image.sh* shell script, root privileges may be required. The build of the image can take approximately 40 minutes, as the PCL is built from the source.

After successfully building the image, a Docker container of this image can be run. The interactive run makes building or developing the solution within this container possible, with a workflow similar to doing so on a remote machine. The procedure for running the container and building the solution, and executing an example is as follows:

1. Change directory to the *attachments* folder.
2. Execute the *run_docker.sh*, which creates a container of the Docker image created earlier and enters this container in interactive mode.
3. Within the Docker container, execute the *build_solution.sh* shell script, which compiles the solution.
4. To test the solution, run *'python3 /masters_project/solution/examples/example.py'*.

C Solution pipeline related files

Multiple JSON files are used within the created solution pipeline. These include file path lists, configuration files, and results. Individual contents of these types of files are described below while being organized by relevant parts of the solution pipeline, where they are used or generated. For each object, its type is noted in square brackets, followed by a short description.

C.1 Dataset preparation

The dataset preparation part of the solution pipeline requires two JSON files as an input and outputs dataset folder, including the main dataset JSON file.

C.1.1 Dataset preparation config JSON

These files are used as input.

- *tag* [**str**] user note on the dataset,
- *virtual_depth_camera* [**contains objects below**],
  - *field_of_view_deg* [**1×2 vec of float**] horizontal and vertical FOV of the virtual depth camera,
  - *noise* [**contains objects below**],
    - ◇ *model* [**str**] noise model to use ("none", "gaussian", and "gaussianray"),
    - ◇ *standard_deviation_mm* [**float**] standard deviation of the noise models in mm,
  - *resolution_px* [**1×2 vec of int**] resolution of a virtual depth camera in horizontal and vertical axes,
- *organized_output* [**bool**] output point clouds are organized (this makes the resulting files bigger),
- *virtual_views_icosahedron* [**contains objects below**],
  - *capture_distance_mm* [**float**] virtual depth camera distance from the origin in mm,
  - *subdivide* [**int**] number of icosahedron subdivisions,
  - *use_vertices* [**bool**] use vertices for camera views generation,
  - *xyz_rotation_deg* [**float**] rotate the whole icosahedron in all axes.

C.1.2 Model list JSON

These files are used as input.

- *model_paths* [**1×*n* vec of str**] paths to the STL 3D model files.

C.1.3 Dataset out JSON

These files are outputted.

- *dataset_preparation_config* [**dataset preparation config**],
- *models* [**1×*n* vec of model objects**],
  - [**model**],
    - ◇ *T1* [**4×4 mat**] transformation matrix,
    - ◇ *id* [**int**] ID of the model,
    - ◇ *name* [**str**] name of the model,
- *views* [**1×6 vec of view**],
  - [**view**],
    - ◇ *T2* [**4×4 mat**] transformation matrix,
    - ◇ *id* [**int**] ID of the view.

C.1.4 Generated dataset folder

The following file structure is generated when creating a dataset.

```
specified_dataset_output_path/
├─dataset.json ... generated Dataset out JSON
├─descriptors/ ... descriptors already computed in the offline stage
│  ├─descriptor_name.pcd
│  ├─descriptor_name_config.json
│  ├─descriptor_name_references.json
│  └─...
└─models/
   ├─0000/ ... folder named by model ID
   │  ├─mesh.stl
   │  └─views
   │     ├─0000.pcd ... partial view named by view ID
   │     └─...
   └─...
```

C.2 Global pipeline

The global pipeline requires extensive configuration contained within the JSON file described below. Moreover, this part of the solution pipeline outputs various JSON files, which are also presented.

C.2.1 Global pipeline config JSON

These files are used as input.

- *preprocessing* [**contains objects below**],
  - *scale_flag* [**bool**] use scaling,
  - *scale_factor* [**flaot**] scale factor,
  - *xyz_filter_flag* [**bool**] use XYZ filter,
  - *xyz_limits* [**1×6 vec of float**] points within this limits are preserved,
  - *voxel_grid_flag* [**bool**] use voxel grid filter,
  - *voxel_size* [**float**] voxel size,
  - *plane_removal_flag* [**bool**] use plane removal,
  - *plane_removal_distance_threshold* [**float**] plane removal threshold in mm,
  - *radius_outlier_removal_flag* [**bool**] use radius outlier removal,
  - *radius_outlier_removal_radius* [**float**] radius around each point of the point cloud,
  - *radius_outlier_removal_min_points_in_radius* [**int**] minimal number of points that must be inside the radius,
  - *statistical_outlier_removal_flag* [**bool**] use statistical outlier removal,
  - *statistical_outlier_removal_num_of_k_nearest_neighbors* [**int**] number of k nearest neighbors that will be considered for computing statistics,
  - *statistical_outlier_removal_std_deviation_multiplier* [**float**] standard deviation multiplier for a threshold above which the points are removed,
- *normal_estimation* [**contains objects below**],
  - *search_radius* [**float**] search radius for normal estimation in mm,
  - *normal_to_view_angle_deviation_degrees* [**float**] allowed deviation from the camera -Z axis and estimated normal,
- *global_descriptor* [**str**] global descriptor to use ("vfh", "cvfh", "ourcvfh", "good", "esf"),
- *coarse_alignment* [**contains objects below**],
  - *use_crh* [**bool**] use crh for coarse rotation alignment,
  - *use_descriptor_if_possible* [**bool**] use a descriptor for coarse alignment if possible (in case of "ourcvfh" or "good")
  - *use_correction_by_centroids* [**bool**] use correction by translating candidate point cloud based on centroids,

- *pose_refinement* [**contains objects below**],
  - *voxelize_scene* [**bool**] voxelize scene before pose refinement,
  - *voxelize_view* [**bool**] voxelize candidates before pose refinement,
  - *voxel_size* [**float**] voxel size,
  - *enforce_same_direction_normals* [**bool**] enforce same direction of normals for scene and candidates,
  - *euclidean_fitness_epsilon* [**float**] euclidian fitness epsilon,
  - *max_correspondence_distance* [**float**] max distance between correspondences,
  - *max_iterations* [**int**] maximum number of iterations,
  - *transformation_epsilon* [**float**] transformation epsilon,
  - *use_symmetric_objective* [**bool**] use symmetric objective,
- *hypothesis_verification* [**contains objects below**],
  - *model_representation_voxel_size* [**float**] voxel size for creation of full point cloud from all partial views,
  - *inlier_threshold* [**float**] inlier threshold,
  - *occlusion_reasoning* [**bool**] use occlusion reasoning,
  - *occlusion_threshold* [**float**] occlusion threshold,
  - *regularizer* [**float**] regularizer size,
  - *verification_voxel_size* [**float**] voxel size used for hypothesis verification
- *vfh* [**contains objects below**],
  - *k_nn* [**int**] number of nearest neighbours,
  - *use_fill_size_component* [**bool**] use fill size component,
  - *use_normalized_bins* [**bool**] normalize bins after computation,
- *cvfh* [**contains objects below**],
  - *cluster_tolerance_distance* [**float**] cluster tolerance distance,
  - *eps_angle_threshold_deg* [**float**] eps angle threshold in degrees,
  - *k_nn* [**int**] number of nearest neighbours,
  - *min_points* [**int**] minimum number of points in a cluster,
  - *use_normalized_bins* [**bool**] normalize bins after computation,
- *ourcvfh* [**contains objects below**],
  - *cluster_tolerance_distance* [**float**] cluster tolerance distance,
  - *eps_angle_threshold_deg* [**float**] eps angle threshold in degrees,
  - *k_nn* [**int**] number of nearest neighbours,
  - *min_points* [**int**] minimum number of points in a cluster,
  - *use_normalized_bins* [**bool**] normalize bins after computation,
- *esf* [**contains objects below**],
  - *k_nn* [**int**] number of nearest neighbours,

- *good* [**contains objects below**],
  - *k_nn* [**int**] number of nearest neighbours,
  - *num_of_bins* [**int**] number of bins for orthogonal view,
  - *threshold* [**float**] threshold value,
- *debug* [**contains objects below**],
  - *result_to_cout* [**bool**] print result to the terminal,
  - *step_notion_to_cout* [**bool**] print information about executing individual steps to the terminal,
  - *save_extras_json* [**bool**] print result to the terminal,
  - *save_point_clouds* [**bool**] save point clouds to the output folder,
  - *save_used_dataset_json* [**bool**] save used dataset out to the output folder,
  - *save_used_pipeline_config_json* [**bool**] save used pipeline config to the output folder
  - *visualize_flag* [**bool**] visualize steps according to visualize object
  - *visualize* [**contains objects below**],
    - *preprocess* [**bool**] visualize preprocess step,
    - *normal_estimation* [**bool**] visualize normal estimation step,
    - *coarse_alignment* [**bool**] visualize coarse alignment step,
    - *pose_refinement* [**bool**] visualize pose refinement step,
    - *result* [**bool**] visualize result,
    - *result_with_candidates* [**bool**] visualize result with all candidates.

C.2.2 Result JSON

These files are outputted.

- *T_final* [**4×4 mat of float**] transformation matrix,
- *candidate_id* [**int**] result candidate id (default: -1),
- *icp_converged* [**bool**] signaling if the ICP converged for the resulting candidate (default: false),
- *icp_fitness* [**float**] ICP fitness of the resulting candidate (default: -1),
- *match_distance* [**float**] descriptor match distance of the resulting candidate (default: -1),
- *model_id* [**int**] model id of the resulting candidate,
- *model_name* [**str**] model name of the resulting candidate,
- *timestamp* [**str**] formatted timestamp,
- *view_id* [**int**] view id of the resulting candidate.

C.2.3 Extras JSON

These files are outputted.

- *candidate_extras* [**1×*n* vec of candidate objects**]
  - [**candidate**]
    - ◇ *T_1* [**4×4 mat of float**] transformation matrix,
    - ◇ *T_2* [**4×4 mat of float**] transformation matrix,
    - ◇ *T_3* [**4×4 mat of float**] transformation matrix,
    - ◇ *T_4* [**4×4 mat of float**] transformation matrix,
    - ◇ *T_final* [**4×4 mat of float**] transformation matrix,
    - ◇ *icp_converged* [**bool**] signaling if the ICP converged for the resulting candidate (false),
    - ◇ *icp_fitness* [**float**] ICP fitness of the resulting candidate (default: -1),
    - ◇ *id* [**int**] candidate ID (before ICP fitness sorting),
    - ◇ *match_distance* [**float**] descriptor match distance of the resulting candidate (default: -1),
    - ◇ *model_id* [**int**] model id of the resulting candidate,
    - ◇ *model_name* [**str**] model name of the resulting candidate,
    - ◇ *num_of_points* [**int**] number of candidate view point cloud points,
    - ◇ *num_of_points_refinement* [**int**] number of candidate point cloud points used for pose refinement,
    - ◇ *view_id* [**int**] view id of the resulting candidate
- *scene_extras* [**contains objects below**],
  - *num_of_points_input* [**int**] number of input scene point cloud points,
  - *num_of_points_normal_estimation* [**int**] number of input scene point cloud points used for normal estimation,
  - *num_of_points_refinement* [**int**] number of input scene point cloud points used for pose refinement,
- *time_profiling* [**1×*n* vec of measured operation objects**]
  - [**measured operation**]
    - ◇ *measure_ms* [**float**] measured time in ms,
    - ◇ *tag* [**str**] tag of the counter
- *timestamp* [**str**] formatted timestamp,

C.2.4 Evaluation JSON

These files are outputted.

- *ground_truth_model_id* [**int**] ground truth model id (in case of using dataset created by solution, otherwise -1),
- *ground_truth_model_name* [**str**] mesh model name
- *ground_truth_transformation_matrix* [**4×4 mat of float**] transformation matrix,
- *ground_truth_view_id* [**int**] ground truth model id (in case of using dataset created by solution), otherwise -1,
- *timestamp* [**str**] formatted timestamp,

- *recognized_correctly* [**bool**] signals if the ground truth model corresponds with the resulting model,
- *rotation_error_deg* [**float**] result to ground truth pose rotation error in degrees,
- *translation_error_mm* [**float**] result to ground truth pose translation error in mm.

Iteration output folder

```
specified_global_pipeline_out_folder/
└─formatted-timestamp/ ... the iteration output folder
  ├─result.json
  ├─evaluation.json (accuracy testing)
  ├─evaluation_dataset.json (accuracy testing)
  ├─extras.json (debug option)
  ├─used_dataset.json (debug option)
  ├─used_pipeline_config.json (debug option)
  ├─candidate_0.ply (debug option)
  ├─...
  ├─candidate_n.ply (debug option)
  ├─result_full.ply (debug option)
  ├─result_partial.pcd (debug option)
  ├─result_partial.ply (debug option)
  ├─scene_cloud_w_normals.pcd (debug option)
  ├─scene_cloud_w_normals.ply (debug option)
  └─scene_original.ply (debug option)
```

D Configuration used for testing

Apart from this chapter, the exact JSON files used for experiments are also in the attached files. The bold values are the values that differ between the experiments.

D.1 Dataset preparation config

Dataset preparation JSON **for the ground truth tests and database creation**:

- *virtual_depth_camera* :
  - *field_of_view_deg* : [50.0, 40.0],
  - *noise* :
    - *model* : **none**,
    - *standard_deviation_mm* : **0.0**,
  - *resolution_px* : [640, 480],
  - *organized_output* : false,
- *virtual_views_icosahedron* :
  - *capture_distance_mm* : **650.0**,
  - *subdivide* : 1,
  - *use_vertices* : true,
  - *xyz_rotation_deg* : **5.0**,

Dataset preparation JSON **for the virtual noise tests dataset** creation:

- *virtual_depth_camera* :
  - *field_of_view_deg* : [50.0, 40.0],
  - *noise* :
    - *model* : **gaussianray**,
    - *standard_deviation_mm* : **3.0**,
  - *resolution_px* : [640, 480],
  - *organized_output* : false,
- *virtual_views_icosahedron* :
  - *capture_distance_mm* : **700.0**,
  - *subdivide* : 1,
  - *use_vertices* : true,
  - *xyz_rotation_deg* : **15.0**,

D.2 Global pipeline config

Global pipeline config JSON **for ground truth tests and virtual noise tests** experiments:

- *preprocessing* :
  - *scale_factor* : **1000.0**,
  - *scale_flag* : **false**,
  - *xyz_filter_flag* : **false**,
  - *xyz_limits* : [**-10000.0, 10000.0, -10000.0, 10000.0, -10000.0, 10000.0**],
  - *voxel_grid_flag* : true,
  - *voxel_size* : 1.0,
  - *plane_removal_distance_threshold* : **10.0**,
  - *plane_removal_flag* : **false**,
  - *radius_outlier_removal_flag* : false,
  - *radius_outlier_removal_min_points_in_radius* : 50,
  - *radius_outlier_removal_radius* : 10.0,
  - *statistical_outlier_removal_flag* : **false**,
  - *statistical_outlier_removal_num_of_k_nearest_neighbors* : **50**,
  - *statistical_outlier_removal_std_deviation_multiplier* : **1.0**,
- *normal_estimation* :
  - *normal_to_view_angle_deviation_degrees* : 89.0,
  - *search_radius* : 9.0,
- *vfh* :
  - *k_nn* : 6,
  - *use_fill_size_component* : true,
  - *use_normalized_bins* : true,
- *cvfh* :
  - *k_nn* : 6,
  - *min_points* : 200,
  - *cluster_tolerance_distance* : 9.0,
  - *eps_angle_threshold_deg* : 20.0,
  - *radius_normals* : 9.0,
  - *use_normalized_bins* : true,
- *ourcvfh* :
  - *k_nn* : 6,
  - *min_points* : 200,
  - *cluster_tolerance_distance* : 9.0,
  - *eps_angle_threshold_deg* : 20.0,
  - *radius_normals* : 9.0,
  - *use_normalized_bins* : true,

- *esf* :
  - *k_nn* : 6,
- *good* :
  - *k_nn* : 6,
  - *num_of_bins* : 6,
  - *threshold* : 200.0,
- *coarse_alignment* :
  - *use_crh* : true,
  - *use_descriptor_if_possible* : true,
  - *use_correction_by_centroids* : true,
- *pose_refinement* :
  - *enforce_same_direction_normals* : true,
  - *euclidean_fitness_epsilon* : 0.1,
  - *max_correspondence_distance* : 1500.0,
  - *max_iterations* : 150,
  - *transformation_epsilon* : 0.1,
  - *use_symmetric_objective* : true,
  - *voxel_size* : 3.0,
  - *voxelize_scene* : true,
  - *voxelize_view* : true,
- *hypothesis_verification* :
  - *inlier_threshold* : **10.0**,
  - *regularizer* : 45.0,
  - *resolution* : 5.0,
  - *occlusion_reasoning* : true,
  - *occlusion_threshold* : **6**,
- *debug* :
  - *result_to_cout* : false,
  - *step_notion_to_cout* : true,
  - *save_extras_json* : true,
  - *save_point_clouds* : false,
  - *save_used_dataset_json* : true,
  - *save_used_pipeline_config_json* : true,
  - *visualize_flag* : false,
  - *visualize* :
    - *preprocess* : false,
    - *normal_estimation* : false,
    - *coarse_alignment* : false,
    - *pose_refinement* : false,
    - *result* : false,
    - *result_with_candidates* : false,

Differences from the configuration above **for capture experiments**:

- *preprocessing* :
  - *scale_flag* : **true**,
  - *scale_factor* : **1000**,
  - *xyz_filter_flag* : **true**,
  - *xyz_limits* : [**-262, 262, -261, 200, 200, 790**],
  - *voxel_grid_flag* : true,
  - *voxel_size* : 1.0,
  - *plane_removal_flag* : **true**,
  - *plane_removal_distance_threshold* : **11.0**,
  - *radius_outlier_removal_flag* : false,
  - *radius_outlier_removal_radius* : 10.0,
  - *radius_outlier_removal_min_points_in_radius* : 50,
  - *statistical_outlier_removal_flag* : **true**,
  - *statistical_outlier_removal_num_of_k_nearest_neighbors* : **50**,
  - *statistical_outlier_removal_std_deviation_multiplier* : **4.0**,
- *hypothesis_verification* :
  - *inlier_threshold* : **15.0**,
  - *regularizer* : 45.0,
  - *resolution* : 5.0,
  - *occlusion_reasoning* : true,
  - *occlusion_threshold* : **15**,