Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Department of Computer Science

Master Thesis

Best Practice Algorithms in 3D Perception and Modeling

Sebastian Blumenthal

A thesis submitted to the
University of Applied Sciences Bonn-Rhein-Sieg
for the degree of
Master of Science in Autonomous Systems

Referee and Tutor: Prof. Dr. Erwin Prassler[1]
Referee: Dipl.-Inf. Jan Fischer[2]
external Referee: Dipl.-Inform. Walter Nowak[3]

Submitted: March 2010

[1]University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin, Germany
[2]Fraunhofer-Institut für Produktionstechnik und Automation (IPA), Stuttgart, Germany
[3]GPS Gesellschaft für Produktionssysteme GmbH (GPS), Stuttgart, Germany

I, the undersigned below, declare that this work has not previously been submitted to this or any other university, and that unless otherwise stated, it is entirely my own work.

DATE

Sebastian Blumenthal

ABSTRACT

A robot needs a representation of its environment to reason about and to interact with it. Different 3D perception and modeling approaches exist to create such a representation, but they are not yet easily comparable. This work tries to identify *best practice* algorithms in the domain of 3D perception and modeling for robotic applications. The goal is to have a collection of refactored algorithms that are easily measurable and comparable. To achieve this goal, software engineering techniques are applied to decompose existing algorithms into atomic elements.

A state-of-the-art survey identifies common data-types and algorithms for this domain. This work proposes harmonized data-types and harmonized interfaces for software components. The components encapsulate the atomic parts of the common algorithms. Existing implementations in public available software libraries are refactored to implement the proposed components. The software is integrated into one common framework, called BRICS_3D library. Benchmarks of the components are performed with existing data sets. These benchmarks are the base for impartial deduction of *best practice* for a given task.

ACKNOWLEDGMENTS

I would like to thank Erwin Prassler for supervising my master thesis and for giving me the great opportunity to participate in the BRICS project. I would like to thank Jan Fischer for giving me uncomplicated support with my thesis. My thanks to Walter Nowak for fruitful discussions and critical feedback.

I particular thank all the colleagues from the GPS for having a warm and inspiring working environment. Especially I would like to thank Alexey Zakharov, Thilo Zimmermann, Corinna Noltenius and Knut Drachsler.

My thanks to Davide Brugali, Herman Bruyninckx and Gerhard Kraetzschmar for giving me valuable feedback. I would like to thank all the other members of BRICS, for gathering valuable and new experiences.

I would like to thank Thorsten Linder for poof reading my this. My thanks to all my former colleagues Christoph Brauers, Viatcheslav Tretyakov, Peter Molitor, Dirk Holz, David Dröschel, Hartmut Surmann and Paul Plöger at IAIS for having a great time in the robot pavilion.

I appreciate my highest thanks to my parents Irmhild Blumenthal and Werner Blumenthal for their love, emotional and financial support during my whole study period. Many thanks to my sister Martina Blumenthal for encouraging me and for proof reading my work. Thank you very much, Ina Overath for backing me up, especially in the final stages of my thesis.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

Autonomous robot systems are complex systems with different kinds of hardware and software components. A developer, which has to design a robot for a specific task, faces many problems: What is the right choice of sensors, actuators or the robot platform? Which algorithms like navigation, task planning, manipulation or machine learning strategies are the most suitable for the application? Which of them need to be integrated and which need to be reimplemented? Nowadays, robots are typically build from scratch because a well established *robot engineering* or *robot development process* is missing completely. A *robot development process* would significantly help the developer and speed up the development time. One important aspect is that the process should help to get access to *best practice* choices for the application, and thus suitable algorithms should be measurable and comparable.

1.2 Context of work

This work is done in the context of the European Best Practice In Robotics (BRICS) project. The project's focus lies on identifying a *robot development process*. A *Model Driven Engineering (MDE)* approach for a *robot development process* is the desired goal. This comprises several subcomponents: hardware, middleware, drivers, algorithms, models and tools that support the *MDE* approach. BRICS tries to identify *best practice* solutions in each of these categories, so a robot developer might later choose the components in his design software, which are most suitable for a specific application. This master thesis is about the algorithmic component, to be more precise: *best practice* algorithms in 3D perception and modeling[4].

1.3 Problem statement

This work addresses the crucial component of 3D perception and modeling for robotic applications. A robot needs a representation of its environment to reason about what to do next to accomplish a given task. This representation is a *model* of the environment. To get access to a model the robot needs perception that means the ability to sense the environment. The more advanced the task of the robot, the more sophisticated its perception must be. For example a robot that should grasp three dimensional (3D) objects like a cup on a table needs a three dimensional

[4]The term "modeling" is referred here as a 3D representation of an object, rather than a part of *Model Driven Engineering (MDE)*

world model. Different approaches exist, but they are not yet easily comparable or exchangeable. The *problem* is to identify *best practice* algorithms for 3D perception and modeling. That means to make algorithms interchangeable, comparable and thus measurable. The goal of this work is to provide a framework of refactored *best practice* algorithms for 3D perception and modeling for robotic applications.

1.4 Thesis Outline

This master thesis is structured as follows:

- Chapter 2 clarifies definitions of important terms occurring in this work, followed by a general overview in which the context of the 3D perception and modeling domain is embedded. The Chapter closes with algorithmic details of the 3D perception and modeling domain.
- Chapter 3 depicts current efforts in benchmarking of robots and why it is so difficult to compare different robotic systems. Related work of algorithms for 3D perception and modeling is presented, followed by a list of public available libraries that at least partially implement these algorithms. The libraries are a starting point to develop comparable and thus benchmarkable algorithms that can lead to statements about *best practice* algorithms for 3D perception and modeling.
- Chapter 4 explains the process of identifying *best practice* algorithms. First the general approach is depicted, then it is applied to the domain of 3D perception and modeling. Requirements for harmonized data-types, software components and harmonized interfaces for the components are presented.
- Chapter 5 shows the implementation of the requirements for harmonized data-types and the realization of the components for common algorithms. The software is integrated into the *BRICS_3D* library.
- Chapter 6 presents benchmarks of the atomic components to judge, which algorithms are *best practice* for 3D perception and modeling. The used test-bed and the measured performance metrics are discussed.
- Chapter 7 summarized the results of this work and enumerates open issues.

Chapter 2

BACKGROUND

The Background Chapter will start with definitions of important terms occurring in this work, followed by a general overview in which the context of the 3D perception and modeling domain is embedded. The last Section in this Chapter dives a step deeper into the algorithmic details of this domain.

2.1 Terminology

The terminology Section starts with clarification of the terms *best practice* algorithms and the closely related procedure of *benchmarking*. Then definitions in the field of *3D perception* and *modeling* are provided. The terminology for *software engineering* aspects are presented, because these techniques will be applied to the algorithms in the 3D perception and modeling domain, to be able to deduce *best practice*.

Best practice algorithms

Best practice or sometimes also called *good practice*, *best in class* or *leading practice* is a term that has its origin in the business domain. A suitable definition for *best practice* is:

> *"Methods and techniques that have consistently shown results **superior** than those achieved with other means, and which are used as **benchmarks** to strive for. There is, however, **no** practice that is best for everyone or **in every situation**, and no best practice remains best for very long as people keep on finding better ways of doing things.*" [1]

Applied to 3D perception and modeling algorithms for robotic applications, *best practice* can be regarded as an algorithm that performs betters than other algorithms, for a specific task. Depending on the task for a robotic system, the superior or best algorithm might not be the same. For example a scenario requires a fast generation of a 3D model, neglecting high accuracy of that model, while another scenario depends on a very detailed model, but has more relaxed timing constraints. *Best practice* algorithms might even get outdated, when new algorithms out perform older approaches. The quoted definition identifies the principle of a *benchmark* to compare different algorithms.

The term **benchmark** can be defined as ”a standard measurement that forms the basis for comparison”[2] or ”a measure ([of] a rival’s product) according to specified standards in order to compare it with and improve one’s own product” [3]. Applied to the world of robotics this means with the help of a benchmark it is possible to compare the performance of robots or algorithms. Although the perspective of a robot as a ”product” might be uncommon, the second definition gives a hint to an intrinsic motivation behind benchmarking: improve a system. Another motivation is to assess an objective measure that allows to choose the *best* or *best practice* system for a certain application. The measurable quantities for robotic benchmarks are currently a subject of discussion of its own, see also Section 3.1.

3D Perception

Perception is the process of acquiring knowledge about the environment. Perception can be subdivided into two parts: *sensing* and *information extraction* [4]. Sensing is the task of producing measurements with different sensors like cameras or laser range finders. The outcome of the sensing process is typical raw-data, which can be further processed to extract meaningful information. Cognitive reasoning on an abstract level is beyond the scope of perception.

3D perception means that knowledge in the three dimensional domain is gathered. The goal is to generate a 3D representation of the robots environment. It is crucial to reason in a 3D environment, as the robot lives in the real world and interacts with 3D objects and 3D obstacles [5]. To acquire 3D measurements, technology is applied that can achieve *depth perception*. Different sensors and approaches for depth perception exist. Some of them are listed in Section 2.3.1.

3D Modeling

3D modeling is the process of creating a 3D model of the environment. A **3D model** is a three dimensional digital representation of a three dimensional entity. As autonomous robots have to reason about their environment, they also need an representation of this environment. The research field of 3D modeling has its origin in 3D computer graphics and 3D video games. Different ways to represent a model have been developed. They can be categorized into three main categories: *dense depth*, *surface based* or *volumetric* representation [6].

Dense depth representations can be *depth images* that are typically delivered by *depth perception* devices, like Time-of-Flight or stereo cameras (cf. Section 2.3.1). A depth image stores the distance information to the front of the captured scene in the pixels of the image. For example displayed as gray-scale image, bright regions appear nearer to the sensor (depending on the coding of the depth information). As a depth image only captures a portion of an 3D environment that depends on the view point, this is sometimes referred as 2.5D representation, being more than 2D but not fully 3D (cf. Chapter 22 of [7]). Depth images can be aggregated to *layered depth images*

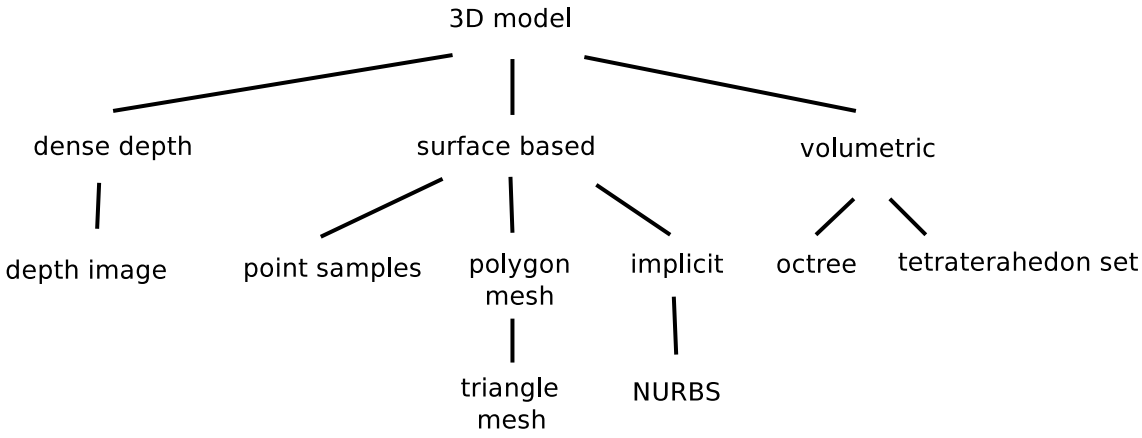


Figure 2.1: Taxonomy of 3D model representations.

(LDI) that contain multiple snapshots of the same object, taken from different positions, to form a full 3D model [6].

Surface based models are the most common 3D representations. An object is represented by its boundary: the surface. The content of the object is neglected. Surface based models fall into three subcategories: *point samples*, *polygon meshes* and *implicit* representations. **Point samples** approximate a surface by picking some points of the surface. Grouped together, the samples form a **point cloud**. Point clouds are possibly the most common representation in robotics, as many sensors like laser scanners create surface samples, for example when a laser beam hits the surface.

Polygon meshes model 3D surfaces with a set of 2D polygons. Each area, also called *facet* that is enclosed by the *edges* of the polygon, is a part of the surface. The points of a polygon are called *vertices* (singular *vertex*). A popular polygon used for meshes is the **triangle**. A triangle is a polygon with the least number of points to create a 2D polygon. Figure 2.2 depicts the terminology of the elements of a triangles. The three vertices are connected via three edges that enclose a facet. Polygon meshes are a very common in manufacturing, architectural and entertainment industries. They are the basic primitives for graphic cards.

The **implicit surface representation** tries to describe the surface with mathematical functions: curves for the 2D case and shapes for the 3D case. *Non-uniform rational basis spline (NURBS)* uses surface patches which are composed of *splines*. A spline is a piecewise constructed smooth function. NURBS are commonly used in *Computer Aided Design (CAD)*.

Volumetric representations of models use *voxels*. A voxel (volume pixel) is the smallest representable amount of space. A simple volumetric representation is a 3D binary grid that indicates if a cell is occupied or not. This approach is extremely memory inefficient. A better representation is the *Octree*. The Octree is a recursive subdivision into eight cubes. Empty cubes are neglected and the model results in a tree like structure of cubes, representing the volume of

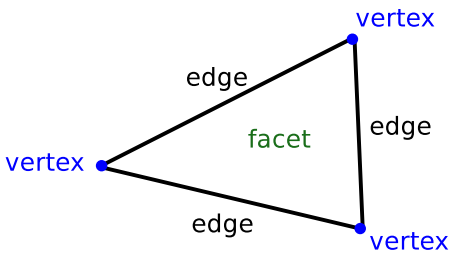


Figure 2.2: Terminology of a triangle. The three vertices are connected via three edges that enclose a facet.

the 3D model. Furthermore, an often used representation is a *set of tetrahedrons*. A volume is composed of many tetrahedrons.

Mathematically, tetrahedrons and triangles are related, as both are *simplices*. A k-simplex is the *convex hull* of exactly $k + 1$ points. The convex hull can be understood as the boundary of a point set. A vertex is a 0-simplex, an edge is a 1-simplex, a triangle is a 2-simplex and the tetrahedron is a 3-simplex. When many simplices describe on object, than it is also called *simplical complex*. The triangle set is a simplical complex for 2D surfaces and the set of tetrahedrons is the simplical complex for 3D volume representations [8].

The robotic domain is typically only interested in surface meshes. For grasping or collision checking the content is irrelevant as only the boundary of the object, defined by its surface, is needed. For the remaining parts of this work meshing algorithms will refer to surface meshes, rather than volumetric approaches.

Software Engineering

Software Engineering is a structured and systematic way to develop software. One important goal is to create maintainable and reusable high quality software. Some techniques in software engineering are the appliance of *software patterns*, *software refactoring*, design of *software components* or *model-driven engineering*.

Software patterns are standardized, abstracted solutions to often recurring problems [9]. To some extent the patterns are *best practice* solutions for software development problems. The description of a pattern captures the core of the solution and includes the main consequences. Although, providing abstract solutions, some patterns are only applicable to object-oriented programming languages, as for example class inheritance might be required.

Software Refactoring means changing the internals of a code without modifying the external behavior [10]. This term fits well to this work, as the algorithms should be made easily comparable without changing the behavior of the algorithm.

Software components are entities that structure software into modules. Each module should contain elements that somehow belong together. The motivation for software components is to have elements the can be reused in other applications. The components communicate via interfaces. The internal structure is unspecified, this is completely left to the programmer that means the implementation could be object-oriented or consist of further subcomponents. Ideally a component should be reusable and replaceable by other implementations (even with 3rd party components) [11].

In **Model-driven engineering (MDE)**, the software developer uses *domain-specific* models. A model could for example describe architectural or behavioral elements and consist of software components. These models are (partially) transformed into source code by special generator tools [12].

Unified Modeling Language (UML) is a standardized way to specify models that describe certain parts of a software. UML can be regarded as the *domain-specific* model for the domain of software development. UML has a well known way to visually display certain aspects of software: *UML diagrams*. There are different UML diagram types. This work uses *class* diagrams which can represent classes of an object-oriented language, and *component* diagrams that depict software components. Software patterns are typically accompanied by UML diagrams.

Unfortunately software engineering principles are not yet widely used for robotic applications [13]. One reason is the difficulty in creating reusable software, because of heterogeneous software and hardware requirements. Another reason is that the development focus is often on efficient implementation of algorithms, as (near) real-time capability are a major design criterion and thus neglecting the reusability of software parts. This work will apply some of the techniques mentioned above to existing algorithms for 3D perception and modeling to make them easier to benchmark and to get easier access to *best practice*.

2.2 Overview of 3D perception and modeling domain

This section will briefly look into the context of 3D perception and modeling that means which neighboring research domains are there, and how are they related. 3D perception and modeling, for robotic applications, has an overlap with at least four other major domains: *depth perception*, *Simultaneous Localization and Mapping (SLAM)*, *object recognition* and *3D modeling*. Figure 2.3 depicts the context of the 3D perception and modeling domain.

- **Depth perception**: The depth perception domain tries to measure depth information that is important to generate a 3D model of an environment. Different kinds of sensor technology is applied, like stereo cameras, Time-of-Flight cameras, laser scanners or structured light.

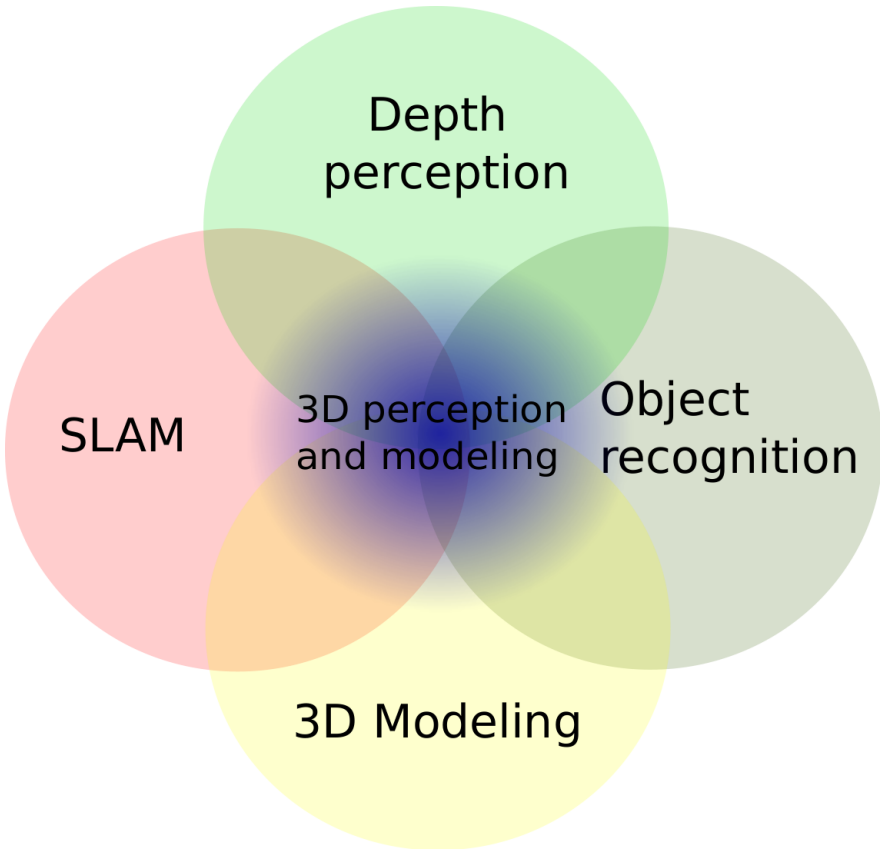


Figure 2.3: Context of 3D perception and modeling domain.

This domain contributes algorithms that can recover depth information, for example depth reconstruction of two images from a stereo camera system. Other algorithms are able to reduce the noise of the measurements or provide low level signal processing capabilities in general that might be already embedded into the sensor devices.

- **Simultaneous Localization and Mapping (SLAM)**: *SLAM* is a robotic research field. Here the problem is that the robot needs to know where it is, therefore it needs a representation of the world. This representation, also called *map*, should be generated autonomously by the robot while it explores the environment. Thus the robot is able to navigate in unknown terrains. The next step is to localize it in the map (*mapping*). If mapping is performed correctly the position on the map corresponds to the position in the real world. The problem of the simultaneous creation of a map and the localization on it has formed the term *SLAM* [14, 15].

  Although early solutions for SLAM operated with 2D *map* representations, recent developments have lifted this representation to 3D *maps*. Thus the SLAM domain contributes algorithms needed for 3D perception and modeling tasks. As sensors are needed to create the *maps*, the SLAM domain has a natural overlap with the field of *depth perception*.

- **Object recognition**: The problem in *object recognition* is to find objects in a scene that are known in a database. For example the task is to find a bottle in an image that is previously stored in a database. Some algorithms use dedicated features in an image like SIFT [16] or SURF [17], to uniquely represent objects. These kind of features are also used in some visual SLAM approaches [18]. Other algorithms exploit spatial properties, for example in point clouds they detect doors, door handles, tables or dashboards [19]. *Object recognition* approaches need sensors and therefore they are also related to the *depth perception* domain, although not all approaches need three dimensional data. Even some *depth perception* algorithms rely on dedicated features, like SIFT or SURF, to estimate the depth from motion [18].

- **3D modeling**: 3D modeling is the field of representing, generating or visualization of 3D models as presented in Section 2.1. The main interest groups are typically 3D computer games, computational geometry, 3D computer animations, digital archives (cultural heritage) like the digital *Michelangelo Project* [20], virtual reality, or even 3D-TV which is a recent emerging branch. Algorithms in this domain are able to transform, refine and visualize 3D models.

  The 3D modeling domain is related to all other previously mentioned fields. *Depth perception* is needed to capture for example cultural heritage, as mentioned before, or stereo cameras are used for 3D-TV applications. 3D SLAM approaches inherently use 3D models in the form of point clouds. Some object recognition techniques store a 3D model in a database which, in the recognition phase, might be projected into an image from a camera.

3D perception and modeling can bee seen as in between all the previous domains. It uses at least partially algorithms from these domains. Further details about the predominant algorithmic subareas are explained in the next Section 2.3.

2.3 3D perception and modeling processing stages

This section will categorize the important algorithmic subareas for 3D perception and modeling. This work is biased towards 3D perception and modeling for mobile manipulation applications. This has historical reasons, as the *best practice algorithms for mobile manipulation planning* have been already refactored and integrated into the *BRICS_MM* library, by [21]. This work on *best practice* algorithms for 3D perception and modeling tries to close the gap between the real world and 3D environment models that are needed by the mobile manipulation planning algorithms. As those planners need a triangle set representation of the environment, the processing stages for 3D perception and modeling aim to create the required a triangle mesh of the environment, with sensor data as input. This process has several stages, as indicated in Figure 2.4: *depth perception*, *filtering*, *registration*, *segmentation*, *mesh generation* and *visualization*.

Please note that Figure 2.4 suggests a *reconstruction pipeline* as it is commonly seen on recent approaches like in [22], [23] or [19]. However, a 3D perception and modeling task does not necessarily have to be organized as a *pipeline*. The reconstruction process could be a network that has multiple *depth perception* modules, multiple *mesh generation* stages or multiple models at different resolutions. For example, a global planner needs a rather rough representation to cope with delicate structures like tables or chairs while an obstacle collision avoidance algorithm needs a very detailed representation. The different 3D models might be acquired or processed with different frequencies. The processing network could even have *bidirectional* connections, where for example the *depth perception* is influenced by the model, as seen in [24]. Or the amount of noise detected by a filter has influence on the depth perception device and adjusts parameters like for example the capturing frequency or brightness correction for cameras.

The insight of possible setups within a processing *network* in terms of reusability is that this work does not impose any a *pipeline* semantics. The reconstruction *pipeline* scenario is just regarded a the simplest possible configuration. This is also why in figure 2.4 the arrows between the stages are only indicated with dashed lines. The remainder of this section describes the different processing stages in further detail.

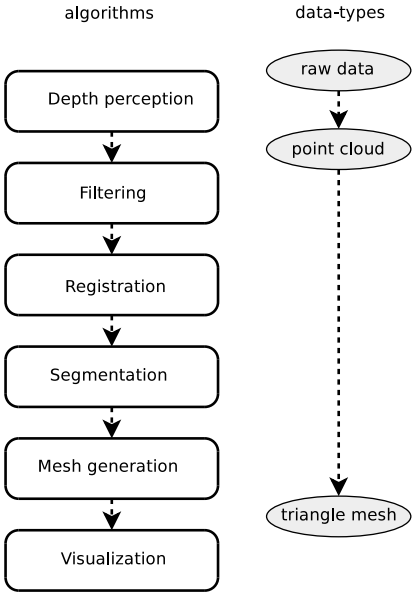


Figure 2.4: Overview of 3D perception and modeling processing stages.

2.3.1 Depth perception

For *depth perception* various kinds of sensors exist. Laser scanners emit laser beams that are reflected when the beams hit the surface. The traveling time of the light is used to deduce the distance. Time-of-Flight cameras follow the same principle, but some devices measure the phase shift of a modulated frequency rather than the traveled time. Stereo camera systems consists of

two cameras that are mounted on a fixed baseline. As the baseline is known, distances to corresponding points can be calculated via triangulation. Sensor data is often encoded into *depth* or *range* images. Although the *depth perception* is a crucial step for 3D perception, it is assumed in this work that depth images are already given. Algorithms in this domain are typically hardware dependent, and the benefit of refactoring algorithms in this sector might be low. This work will provide functionality to turn a depth image into a point cloud representation.

For further information about depth sensing technology, the reader may refer to the following literature: Chapter 22.1 in [7], Chapter 4 in [4] or Chapter 2 in [25].

2.3.2 Filtering

A filter is an algorithm that is able to process a data stream, in this case point cloud data. Three major filtering techniques are often applied to point clouds: *noise reduction*, *size reduction* and *normal estimation*.

Noise reduction filters try to remedy shortcomings of the sensors measurements. *Size reduction* filters sub-sample the input data to get an approximated but smaller amount of data. The less input data an algorithms has, the faster the processing is. *Normal estimation* filters are needed to compute a normal vector for each point in a point cloud. The normal represents the plane normal vector of an underlying patch of the surface. Many algorithms requires point clouds with normals to further process the data. The filtering stage can be regarded as optional, but it is a valuable contribution to create more robust or faster results.

2.3.3 Registration

Registration, also sometimes referred as *matching*, is the process of merging captures from different viewpoints into one global, consistent coordinate frame. This is a robotic problem, because mobile robots move in their environment and thus are able to perceive the environment from different perspectives. Some tasks require to integrate all perceived scene captures into one consistent model to reason about it (for example to plan a path). The most prominent algorithm to solve the registration problem for point clouds is the *Iterative Closest Point (ICP)* method.

As the ICP will be addressed in later Chapters a brief introduction (cf. [25]) shall be given here: The initial version of the ICP was introduced by [26] and [27]. Input data are typically two point clouds *model* and *data*. Other input like polylines, implicit/parametric curves, triangle sets or implicit/parametric surfaces are also possible, but internally the ICP works on point sets. The ICP iterates over the following three steps: establish *point-to-point correspondences*, *estimate the rigid transformation* and *apply transformation* to *data* point set (cf. Algorithm 2.1). The behavior is that the algorithm aligns both point sets better to each other with every iteration.

The creation of *point-to-point correspondences* means that two points from the two different input point clouds are regarded as corresponding if they have the closest Euclidean distance to each other. To compute the closest distance a Nearest Neighborhood search is invoked. This is the computational most expensive part of the ICP, and different approaches and optimization are available.

The second step is to *estimate the rigid transformation* that is needed to minimize a cost function which determines the overall error between the correspondences. This error function is defined as follows:

Two independent sets of points, the model point set $\hat{M}$ with size $|\hat{M}| = N_m$, and data point set $\hat{D}$ with size $|\hat{D}| = N_d$, are the input. The goal is to find a transformation (R, t) that minimizes the following error function [26, 27]:

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{N_m} \sum_{j=1}^{N_d} w_{i,j} \left\| \hat{\mathbf{m}}_\mathbf{i} - (\mathbf{R}\hat{\mathbf{d}}_\mathbf{j} + \mathbf{t}) \right\|^2 \tag{2.1}$$

The parameter $w_{i,j}$ is 1 if the i-th point of the model set $\hat{M}$ has a correspondence with the j-th point from the data point set $\hat{D}$, that means both point sets describe the same 3D point. Otherwise the parameter $w_{i,j}$ is 0. To prevent having a huge matrix for $w_{i,j}$ the equation can be further simplified to:

$$E(\mathbf{R}, \mathbf{t}) = \frac{1}{N} \sum_{N=1}^{N} \left\| \mathbf{m}_\mathbf{i} - (\mathbf{R}\mathbf{d}_\mathbf{i} + \mathbf{t}) \right\|^2 \tag{2.2}$$

Whereas $N = \sum_{i=1}^{N_m} \sum_{j=1}^{N_d} \operatorname{sign}(w_{i,j})$. The correspondence can be now expressed by a tuple $(\mathbf{m_i}, \mathbf{d_i})$ for the i-th correspondence.

In the last step of an iteration R and t are applied to the *data* point cloud. The previous steps are repeated either until the *data* converges to the *model*, for example if the error of Equation 2.2 falls below a certain threshold, or a defined amount of maximum iterations is reached. The ICP is summarized in Algorithm 2.1.

The ICP has two major disadvantages. First, the initial transformation between two point clouds relies on a good guess for correct convergence. Most robotic systems can resolve this problem by incorporation of odometric values from the wheels. Second, the ICP only converges to a local minimum, which does not need to be the correct final alignment. Especially registration of data with spacial ambiguities are problematic.

Algorithm 2.1 The ICP algorithm

1: **for** $i = 0$ to $maximumIteration$ **do**
2: Find correspondences $(\mathbf{m_i}, \mathbf{d_i})$ between point cloud $\hat{M}$ and $\hat{D}$.
3: Estimate the rigid transformation $\mathbf{T}$, which is composed of rotation $\mathbf{R}$ and translation $\mathbf{t}$ that minimizes the error function $E(\mathbf{R}, \mathbf{t})$ in Equation 2.2 between the correspondences $(\mathbf{m_i}, \mathbf{d_i})$.
4: Apply that transformation $\mathbf{T}$ to point cloud $\hat{D}$
5: **if** error $E(\mathbf{R}, \mathbf{t}) <$ some threshold ϵ **then**
6: Terminate algorithm.
7: **end if**
8: **end for**

2.3.4 Segmentation

Segmentation means a spatial partition of point clouds into subsets that belong to different objects. 3D models of special shapes, like boxes, cylinders or balls are often fitted into these regions to model the perceived objects.

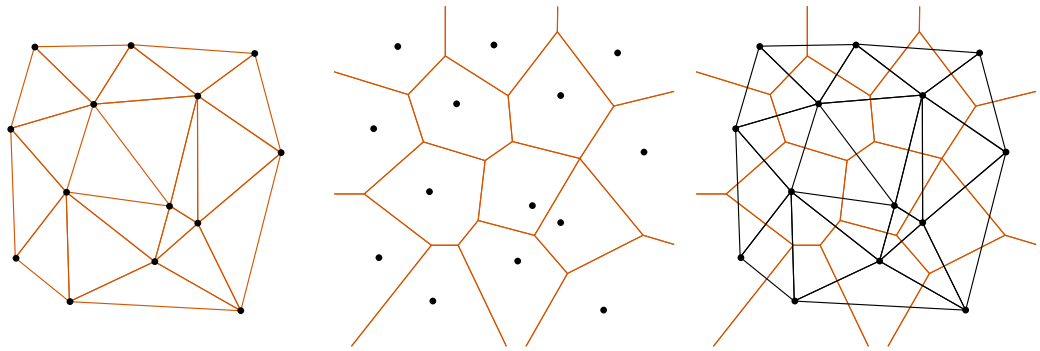
With respect to a mobile manipulation application that needs a triangle set representation of an environment this stage can be regarded as optional. However, the segmentation of data might be helpful to recognize objects that can be grasped, for example.

2.3.5 Mesh generation

The goal of the *mesh generation* step is to transform a 3D point cloud into a triangle mesh. Similar terms are *meshing, shape recovery, surface recovery, surface reconstruction, model retrieval, inverse CAD* or *geometric modeling* (in computer vision). Most of these terms are used in a broader context that already includes some filtering or registration steps. The notion *mesh generation* is used here in a more limited way restricted to the part of model transformation from point cloud to triangle mesh.

Many mesh generation algorithms use **Delaunay triangulation**. *Dalaunay triangulation* is a mesh that fulfills the *Delaunay property*. The Delaunay *empty circle property* for 2D triangulations is defined as the circumscribing circle of any triangle that does not contain any other point of the point set. For the three dimensional case the circumscribing sphere of a tetrahedron does not have any internal other points. Triangulations are possible in any dimension and always result in a partition into simplices of the *convex hull* of the vertices. As a consequence, no simplex intersects any other simplex [8].

The result of a *Delaunay* triangulation is unique, except if more points than the vertices for a simplex are *co-circular*. In this case more than one valid *Delaunay* triangulations exist. For example in the 2D case, four points might be on a circumscribing circle of a triangle, or in the 3D



(a) (b) (c)

Figure 2.5: Relation between Delaunay triangulation and Voronoi graph in a plane. (a) shows the Delaunay triangulation, (b) shows the corresponding Voronoi graph and (c) shows the superposition of both.

case, five points might be on the circumscribing sphere.

To compute a *Delaunay* triangulation for arbitrary dimensions points are typically inserted incrementally and each time the *Delaunay property* is updated. To perform this update, all circles/spheres that contain the new point need to be found. These simplices are then deleted, as they violate the *Delaunay* property. New simplices are added that include the new point as vertex. The number of deletions depends on which point of the input is inserted, thus the complexity depends on the order of the input data. To balance the input data the input points are often selected randomly, thus the name *randomized incremental* algorithm. The worst case complexity for dimension d is $O(n^{d/2})$ while the average case is $O(n \log n)$. For the special case dimension $d = 2$ other algorithms exist, like *flipping*, *plane sweep* or *divide and conquer* but without significant reduction in complexity [28].

Delaunay triangulations have an interesting dual relationship to *Voronoi* graphs. The *Voronoi* graph creates *Voronoi* cells with polygons, such that each point on the cell edges does not have a smaller distance to any other point of the input point set. The *Voronoi* diagrams result from connecting all centers of the circumscribing circles/spheres of the Delaunay triangulation. Figure 2.5[5] depicts the relationship between *Delaunay* triangulations and the corresponding *Voronoi* diagram for the 2D case [29].

2.3.6 Visualization

Visualization, or *rendering*, is the process of displaying the 3D models. This involves a transformation from the models into a 2D image, which can be displayed on a monitor. This task is often performed by specialized hardware: *graphic adapters*. The graphic adapters offer a

[5] Images taken from `http://de.wikipedia.org/wiki/Delaunay-Triangulation`

software interface to render the models that consist of primitive elements like points, lines or polygons. One standardized interface is *OpenGL*, which allows operating system independent access th the graphic adapters [30].

Robotic applications do not necessarily need to visualize the generated models, but visualization can serve as a development or debug tool to visually inspect intermediate results or the output of certain algorithms. Actually, algorithms for rendering do not need to be refactored because standardized interfaces already exist.

The next Chapter will review related work of algorithms of the presented processing stages.

Chapter 3

STATE OF THE ART

This Chapter will start with current efforts in benchmarking of robots and why it is so difficult to compare different robotic systems. As this work focuses on the 3D perception and modeling, related work in this field will be also presented, followed by a list of public available libraries that at least partially implement the presented algorithms. These libraries are a starting point to develop comparable and thus benchmarkable algorithms that can lead to statements about *best practice* algorithms for 3D perception and modeling.

3.1 Benchmarking in robotics

Performance metrics, benchmark databases and widely accepted comparison methodologies are very important instruments, for industry as well as for science. Robotics research does not have a well elaborated methodology for benchmarks and experiments yet. This makes it hard to compare different approaches, especially in different scenarios or environments [31].

Related work is paying more and more attention to benchmarking and evaluation [32]. Benchmarking in robotics can be roughly categorized into a *top-down* and a *bottom-up* a perspective [33]. *Top-down* means the robot is investigated as a whole system or a ”black box”. This view can be further subdivided into robot **competitions** and **system benchmarks**. Competitions as benchmarks like RoboCup have the advantage that it is easy to evaluate if a robot can achieve a given task. As a disadvantage, intensive evaluation with real robots require detailed protocols of experimentations’ to allow for the experiments to be repeated. System benchmarks measure quantities like functional time, memory consumption or accuracy for example in pose estimation.

A more fine-grained benchmarking is achieved in a *bottom-up* fashion. Here each single sub-entity is individually evaluated. Sub-entities can be algorithms, devices, atomic components or complex/composite components. A special problem and open issue is the fact that no common software interfaces are yet available for these entities [31]. Thus harmonized interfaces would be a valuable contribution to enable component-based benchmarking. Most benchmarking efforts have been concentrating on evaluation of a robotic system as a whole, rather than on a component level.

Competitions and challenges

Robot *competitions and challenges* have become a very poplar way to compare robotic systems. The performance metrics for each competition can be seen as standardized as every par-

ticipating team has to obey the same set of rules. The RoboCup [6] is a well-known competition with its various subcategories like RoboCup Rescue or RoboCub@Home. The US Defense Advanced Research Projects Agency hosted the DARPA Grand challenge in 2004 and 2005 and the URBAN challenge[7] in 2007. These competitions benchmarked the capabilities of unmanned autonomous cars. Similar to this, the ELROB [8] activities have been initiated by the German FGAN institution. However, they have a less competitive focus and serve rather as a demonstration of state-of-the-art robotic technology.

Simulation platforms

Simulation platforms play an important role in benchmarking, as they allow to perform experiments of robotic systems or algorithm with the absence of a real robot. Experiments can be repeated easier without a hardware platform, and can be even automatized. The ability to be able to evaluate alternatives (e.g. different algorithms) during the design phase is another advantage of simulation tools [31]. Although simulation is a helpful tool for benchmarking it cannot address all real world problems and it cannot replace system benchmarks with hardware platforms. One of the most popular robot simulators (especially in education) is the Player/Stage[9] framework. It has its focus on 2D navigation, but in combination with Gazebo [10] 3D simulation is possible. The USARSim [11] is a 3D simulation developed for the RoboCup rescue competition. Even Microsoft has developed a commercial tool for robotic simulation: Microsoft Robotic Studio [12]. Recent research efforts have started to integrate robot specific functionality into the Blender simulation toolkit [13]. More specific robotic subareas like simulations for a visual servoing tasks have been addressed with the Java-based Visual Servo Simulator (JaViSS) [14]

Public shared data sets

The robotic community has started to provide *public shared data sets*. The Radish (Robotic Data Set Repository) [15] and the Rawseeds project [16] provide data sets (developed by the Politecnico of Milano) for SLAM applications. Recent efforts have also addressed data sets for visual SLAM[17]. Other robotic subareas established data sets for machine learning like the UCI Machine

[6] http://www.robocup.org/
[7] http://www.darpa.mil/grandchallenge/index.asp
[8] http://www.elrob2006.org/
[9] http://playerstage.sourceforge.net/
[10] http://playerstage.sourceforge.net/gazebo/gazebo.html
[11] http://sourceforge.net/projects/usarsim/
[12] http://msdn.microsoft.com/en-us/robotics/default.aspx
[13] http://wiki.blender.org/index.php/Robotics:Index
[14] http://www.robot.uji.es/research/projects/javiss
[15] http://radish.sourceforge.net
[16] http://rawseeds.elet.polimi.it/home
[17] http://babel.isa.uma.es/mrpt/index.php/Paper:Malaga_Dataset_2009

Learning repository [18] or the PASCAL Collection [19] for visual object recognition. In the field of motion planning the Parasol Lab at A&M University of Texas (famous for the alpha puzzle) [20] or the MOVIE Project (motion planning in virtual environments) [21] made public benchmarks available. Many publications in the field of mesh generation utilize datasets from the Stanford 3D Scanning Repository [22] This repository also includes the famous ”Stanford Bunny”.

Specialized data sets for surface reconstruction are available [23], as well as data sets [24] for 3D segmentation with human labeled reference data.

Performance metrics

Definition of *performance metrics* is a crucial task to be able to perform benchmarking, as it forms the basis of comparison. Metrics are very difficult to define (as they have a multi dimensional nature), but can be categorized into *cost*, *utility* and *reliability*. *Cost* measures the efficiency or *intrinsic quantity*, as entitled in [34], of a system of or an algorithm. Common quantities are computational time, memory consumption, profiling in the sense of how much time has been spend in which functions, amount of communication or energy consumption. *Utility* measures the quality of the outcome of a task or an algorithm. For example the accuracy of a pose estimation algorithm can be compared against manual measurements. The *reliability* gives information about the failure-success rate of a given task. How often a robot can successfully plan a path to a given goal could be such a scenario. Some examples for performance metrics for SLAM applications can be found in [31]. Here the measured quantities are the time required to reach a goal pose and the accuracy between final and desired goal pose. [35] defines performance metrics for evaluation navigation of mobile robot. The NIST (National Institute of Standards and Technology) is developing performance metrics [25] for mobile robots in realistic environments, but these are not yet commonly accepted methodologies by the robotic comminity. The University of Zaragoza focuses on metrics for obstacle avoidance algorithms in their *Automatic Evaluation Framework* [33].

Conference tracks and research coordination

Dedicated *conference tracks* have been established like the *Performance Metrics for Intelligent Systems (PerMIS)* workshop [26] which was held for the first time in 2000, or workshops on benchmarks on other well-known international events (*IROS'06, IROS'07, RSS'08, IROS'08*).

[18] http://mlearn.ics.uci.edu/MLRepository.html
[19] http://www.pascal-network.org/challenges/VOC/databases.html
[20] http://parasol-www.cs.tamu.edu/groups/amatogroup/benchmarks/mp/
[21] http://www.cs.uu.nl/centers/give/movie/description.php
[22] http://graphics.stanford.edu/data/3Dscanrep/
[23] http://www-sop.inria.fr/prisme/manifestations/ECG02/SurfReconsTestbed.html
[24] http://www-rech.telecom-lille1.eu/3dsegbenchmark/
[25] http://www.isd.mel.nist.gov/projects/USAR/
[26] http://www.isd.mel.nist.gov/research_areas/research_engineering/Performance_Metrics/past_

A broader view on benchmarking in robotics are provided by *research coordination* initiatives that have been established to push standardized benchmarks for robotics, like the *RoSta* [27] project, the *Autonomy Levels For Unmanned Systems (ALFUS)* project [28], or the *EURON* Network [29] with its subgroups, specialized in manipulation and grasping, motion planning, networked robotics, and visual servoing. The *EURON GEM (Good Experimental Methodology) Special Interest Group* has published a document entitled "General Guidelines for Robotics Papers Involving Experiments" [36] with benchmark recommendations for the following domains: SLAM, Mobile Robots Motion Control, Obstacle Avoidance, Grasping, Visual Servoing and Autonomy/Cognitive Robotics.

Trends

Current robotic papers with a benchmarking focus, follow certain tends. Typically, the purpose of the experimental evaluation can been categorized into four motivations [37]:

- The first motivation has just the intention to demonstrate that something works, regardless how many attempts have been necessary to achieve a certain task.
- The second wants to show that an algorithm or a system performs better than some other algorithm or system (*horse race paper*).
- The third tries to get insight on a certain behavior or limits.
- The last motivation is a mixed version of the previous ones.

Especially papers that are dominated by the first motivation often do not follow well elaborated methodologies, which makes it hard to compare different approaches.

Difficulties

Some common *difficulties* in comparison between different robots or algorithms arise from the substantial differences in the used software or hardware [38]. This makes it hard or impossible to reproduce the presented results. The pure publication of results in papers does not make an algorithm comparable. This lack of comparability slows down research progress and prevents easy evaluation of the *best practice* algorithms.

[34] remark that it is difficult to perform time-consuming experiments and denotes a weak awareness of the important role of experiments in the robotic development process. In many papers *tuning* parameter of algorithms are not presented in enough detail to reproduce an experiment. Furthermore algorithms often exploit certain assumptions that are hidden and not made explicit to

[27] `http://www.robot-standards.eu/`
[28] `http://www.isd.mel.nist.gov/projects/autonomy_levels/`
[29] `http://www.euron.org/activities/benchmarks/index.html`

the reader. An issue often neglected is the problem of ground truth which is only partially solved (for example in simulation) [38].

One major difficulty is to ensure **repeatability** of an experiment. If this is given, the experiment can be verified independently by other research groups. Beside a detailed documentation including all parameters and assumptions, all occurred anomalies should be addressed. This does not only foster the honesty of presented results, it also potentially reveals new issues with the proposed approach/algorithm/system which are worth to further study. A solid documentation is supposed to contain the number of trials needed to correctly perform a certain task or to reach a certain goal. Otherwise it remains unclear if the result was achieved by chance.

Benchmarking a complex system like robot and ensuring repeatability is a difficult task, but an interesting parallel can be drawn to the scientific field of *biology*. Here the subject of study (e.g. human body) is also a highly complex system, thus clinical protocols and strict guidelines have been developed to produce reliable and comparable results [32]. Another way to deal with the complexity of robots is to benchmark on different levels. These levels can be either functional for example cognition, perception or control. Or these levels can be on different stages of complexity from a single algorithms to a complete system that means benchmarking either in a *bottom-up* or a *top-down* fashion. To relief the difficulties in robotic benchmarking a number of recommendations has been suggested.

Recommendations

Some *recommendations* emphasize stronger focus on elaborated methodologies for experiments, like the proposed ones in the GEM guidelines [36]. These guidelines started being adopted to subfields for example in the SLAM domain like [39] or [40]. To produce comparable results, the experiments must be well documented. Some ways to compare different algorithms are proposed by [34] and [37]:

1. Use the same source-code. This is not always possible as authors might not be willing or are not allowed to distribute the code. Even if the authors are willing to share their code they might not have it any more or they cannot reproduce the version that was used for the experiments. A solid backup and versioning strategy for every software development alleviates the latter issues.

2. Re-implement code by descriptions in papers. This is difficult for various reasons: first this might be a very time consuming process and sometimes it is even not possible because of hidden assumptions or undocumented parameter settings. Thus the description of all parameters and assumption should be as complete as possible.

3. Compare the results with those listed in previous publications. This is the easiest, but certainly the weakest approach as the comparability for example in processing time on different

hardware is questionable. If the used systems would be benchmarked with a performance benchmark (e.g. drystone/whetstone [30], LINPACK[31], etc), this would give information about the used test-bed. To make the published result more comparable, they could be normalized with these performance benchmarks. One example of using normalized results is shown in the DIAMCS Traveling Salesman Problem Challenge [32], where all new produced results are measured relatively against existing ones, which allows for normalization.

Beside the pure comparison it is important for a benchmarking paper to explain and justify the presented results. That means it should always be tried to explain why something performs better than something else.

To conclude this section, benchmarking in robotics is vital to get access to *best practice* components in robotics, because it forms the basis of comparison. Systematic comparison of robotic systems is an emerging research field, as indicated by various kinds of current research efforts like competitions and challenges, simulation platforms, public shared data sets, definition of common performance metrics, dedicated conference tracks, research coordination initiatives. Establishment of good methodologies and easy comparable result is a difficult task in robotics, because of the substantial difference in hardware or software. Especially benchmarking on a component based level has no standardized way of comparison, as harmonized software interfaces are non existing. This also holds true for algorithms in the domain of 3D perception and modeling.

3.2 Algorithms for 3D perception and modeling

This Section reviews related work in the subfields of the 3D perception and modeling domain, presented in Section 2.3. The focus of the survey is on the registration and the mesh generation process, but filtering and segmentation methods are also briefly discussed.

Filtering

A filter is an algorithm that is able to process a data stream, in this case point cloud data. For 3D perception and modeling, there are three major filtering techniques for point clouds: *noise reduction*, *size reduction* and *normal estimation*.

A **noise reduction** filter tries to cope with noisy measurements from a depth perception device. To reduce ”salt and pepper noise”, [41] uses a *median filter* that takes seven index neighbors in a laser scan into account. A source for this noise is if a laser beam partially hits an edge and is reflected by two surfaces. [42] uses smoothing techniques for point clouds based on estimated

[30] http://www.netlib.org/benchmark/
[31] http://www.netlib.org/linpack/
[32] http://www.research.att.com/~dsj/chtsp

normals and a robust hyperplane projection.

The **size reduction** filter is a possibility to improve the computational time for an algorithms. The less input an algorithms has, the faster it is. The *Octree* decomposition is the de-facto standard to reduce point cloud data [25]. It uses a recursive subdivision of cubes, until a discretization threshold is reached. The input data is approximated by the center of the cells. [41] use a *standard reduction* filter to replace dense grouped points by their mean value. This filter incorporates laser scanner characteristics that have a more dense representation the nearer an object is to the source of a leaser beams. The *PICKY ICP* by [43] takes a subset of points in the point cloud for matching process. The selection strategy uses representative points that are taken from a hierarchically clustering.

A filter for **normal estimation** calculates normals of for the points in a point cloud, such that the normals represent the plane normal of a underlying patch of the surface. It is needed for various algorithms like for registration methods [44], for segmentation [19] and for mesh generation algorithms like [45], [46] or [47]. The approach of [48] approximates the normal by a surface that is formed by the k-Nearest Neighborhood of a query point. After a Principal Component Analysis (PCA), the vector with smallest eigenvalue serves as the estimated normal. [49] uses k-Nearest Neighborhood in combination with plane fitting. More robust fitting methods have been proposed by [19], which use a MLESAC method, inspired by the approach of [50], to estimate the normals. [51] use a least square fitting approach applied to the k-Nearest Neighborhood. The authors conclude that the accuracy of the normal estimation using a total least square approach depends on the noise in the point cloud, the curvature of the underlying surface, the density and distribution of the points and the neighborhood size k.

Registration

In the registration process, scene captures from different viewpoints are merged into one common and consistent coordinate frame. This survey concentrates on registration of static environments, that means deformable or articulated objects are not taken into account. Registration of point clouds can be performed either in a *global* or a *local* fashion.

Global strategies for registration involve genetic algorithms like [52], or evolutionary computation approaches [53]. As these registration methods are computational expensive they are uncommon for applications in the robotics domain, as (near) real-time capabilities are important. A recent development: *HSM3D (Hough Scan Matched in 3D)* [54], uses the Hough transform for a global registration. As comprehensive experiments have not been performed yet, it remains unclear how competitive this solution is compared to other existing approaches. Therefore, most efforts have been spend on *local* registration techniques.

The most prominent **local** registration algorithm is the **Iterative Closest Point (ICP)**. The

original ICP version was invented by [26] and [27]. Since then various improvements have been made that mostly address the *correspondence problem* of finding corresponding point pairs in two point clouds and the *rigid transformation estimation* problem. Both problems are the atomic elements the ICP (cf. Algorithm 2.1 step 2 and 3 in Section 2.3.3).

The research efforts in finding the **point-to-point correspondences** have addressed either improvements in the *robustness* or the *computational* complexity.

Approaches to increase **robustness** that means that the point-to-point relations are correctly calculated, typically enhance the spacial information of a point by additional information like intensity [55], color [56] or point normals [44]. The calculation of distinctive features in point clouds to deliver an approximate initial alignment is applied by [57] and [58]. Recent efforts developed the *Point Feature Histograms (PFH)* [59] which is demonstrated in [19] The PFH is an informative pose-invariant local feature, which represents the underlying surface model properties of a point. The resulting distinctive feature descriptor bases on angular relation of the k-Nearest Neighborhood expressed with a Darboux frame in combination with the Euclidean distance. An improved version called *Fast Point Feature Histograms (FPFH)* has been presented by [60]. The authors conclude that the Euclidean distance information has only a minor influence on the expressiveness on the descriptor, and thus can be neglected. [61] improves robustness for the correspondence problem, by computing a rough initial transformation guess with an *Extended Gaussian Image (EGI)* and a rotational Fourier function.

Various improvements that address **computational complexity** for the ICP have been proposed by [62]. As computation of a point-correspondence typically involves a Nearest Neighbor search, all improvements of the more general k**-Nearest Neighbors** search problem can help to speed up the ICP algorithm. The goal is to fasten up the extensive search through the solution space to find the closest entities to a search query. A naïve solution has a complexity of $O(n^2)$, as every point is compared to every other point in a cloud.

A widely used solution to reduce complexity in k-Nearest Neighbors problems the is usage of *k-d trees*. Possibilities for optimization of k-d trees are the different splitting and merging rules. [25] describes that splits performed at the longest axis of the point clouds, which ensure compact volumes, are the most promising policy to yield an optimized k-d tree for the ICP, with a reduced complexity to $O(kn\ log\ n)$. Further improvements of the k-d tree for the ICP is the *cached* k-d tree [63]. It starts searches in following iterations at the leaves of the tree. This advantage can only be gained if a Nearest Neighbor search is invoked several times, as it is the case for the ICP. Despite the processing time improvements, the memory consumption is slightly higher. The k-d tree works best for low dimensional data, as it is the case for 3D point clouds.

A k-Nearest Neighbors search algorithm for higher dimensions has been proposed by [64]. It is an approximated search with *Balanced Box-Decomposition Trees (BD-Trees)* that require less

operations than searching a regular k-d tree. The implementation is also known as *ANN* library. [25] showed that the k-d tree slightly outperforms the BD-tree for dimension three.

The use of *multiple randomized k-d trees* has been proposed by [65] and [66]. For high search space dimensions it tends to be faster than a single k-d tree. [67] present the *spill tree* for Nearest Neighbor search, though the *randomized k-d approach* is faster and less memory intensive.

The concept of *hierarchical k-means* is used by [68] in their vocabulary tree approach. Similar to this, [69] present a novel improvement of the *k-means clustering* algorithms by exploiting priority queue based search instead of depth first search. They also propose the *FLANN* library that automatically detects the best suitable algorithm. Two variants are selected as the result of minimized cost function of the parameters: either *hierarchical k-means* or *multiple random k-d trees*. Their parameter selection is inspired by [70] and [71]. However, the presented results are only performed with high dimensional data sets.

[72] present the *STANN* library. The implemented algorithm tries to exploit parallelization techniques to get a performance speedup for k-Nearest Neighbors search on multi-core and multi-processor hardware architectures. The algorithm bases on the *z* or *Morton* ordering that means the data is sorted according to this, before searching the k-Nearest Neighbors is performed. The construction of this ordering is reduced to a bit-wise comparison to further gain performance improvements. The results where performed on different hardware architectures with slightly different setups (number of threads). The proposed algorithm seems to outperform the *ANN* library, on appropriate hardware. [73] uses the *CUDA* framework[33] to accelerate the Nearest Neighbor search on a Nvidia graphics adapter. The authors' experiments show that the GPU[34] version is 88 times faster on their test-bed than the k-d tree based sequential version. [74] also exploits GPU acceleration for construction of k-d trees.

Note that some of the approaches above are general solutions of k-Nearest Neighbors searches and can be used beyond the scope of the ICP algorithm. There are methods that address computational complexity, which are dedicated to be used for the matching processes with ICP. For example one approach is to exploit semantic information like walls, floors or ceilings, as seen in [75]. Results show that computational time can be up to 30% faster. [76] presents a concept of a parallel ICP called *pICP* where the correspondences are calculated in distributed manner. [77] combined an *pICP* implementation based on the *OpenMP*[35] technology with a *cached* k-d tree. The author was able to exploit hardware specific characteristics and strongly coupled algorithm internals to speed up the registration process.

Along with the improvements for computational complexity, the computation of the **rigid transformation estimation** is an important step for the ICP algorithm: As depicted by [25], four

[33] Compute Unified Device Architecture. Software framework for parallel computing on Nvidia graphic adapters.

[34] Graphics Processing Unit. This is the equivalent to a CPU that is embedded on a graphics adapter. It is specialized on 3D rendering.

[35] Cross platform programming interface to create parallelized applications.

closed-form variants exist. Each algorithm tries to estimate a transformation that minimizes the cost function in Equation 2.2. The first variant is a *Singular Value Decomposition (SVD)* based solution that is directly derived from the (R, t) representation of the transformation [78]. The second variant exploits orthonormal properties of the rotation matrix in combination with an *eigensytem* [79]. The third version uses a *unit quaternion* in combination with computation of *eigensystem* [80]. The forth closed form solution variant incorporates a *dual quaternion* representation [81]. The authors of [82] have compared the four solutions. They conclude that all algorithms have about the same accuracy with respect to stability and noisy data.

Beside the closed form solutions, *approximated* estimation approaches exist. One method is based on *helical motion* proposed by [83] and applied by [19]. It uses instantaneous kinematics that means it uses a point-to-surface, rather than a point-to-point metric. A possible disadvantage is that the method only works reliably for small displacements of the point clouds. [25] denotes that this algorithms will take more iterations, as it is an approximation. The author also contributes an approximated rigid transformation estimation bases on a *linearized rotation*.

Although the ICP can be regarded as the de-facto standard for registration, other approaches can solve the registration problem. [84] just use the poses, deuced from their visual localization system, to transform the scenes data into one common coordinate frame.

A recent development is the probabilistic *Normal Distributions Transform (NDT)* [85]. It uses combinations of normal distributions, rather than single points in a cloud. Each distribution describes the probability to find a part of the surface at any point in space, thus resulting in a smooth representation of a point cloud. Standard numerical optimization techniques are applied to compute the registration. As a grid structure is used, accuracy suffers from discretization, but a triliniear interpolation can increase accuracy with the sake of performance loss in computational time. A comparison of ICP and NDT [40] shows that the pure NDT is slightly faster than ICP and the convergence seems to be better. But ICP behaves more predictable with respect to noise in the initial alignment. The triliniar NDT, with a higher accuracy, is slower than the ICP.

Segmentation

The segmentation process tries to partition the scene model into sets that belong to different objects. Often 3D models of special shapes, like boxes, cylinders or balls are fitted into these regions to model the perceived objects. Common segmentation criteria are *normals* of the points in a point cloud [22]. [19] perform a segmentation of a kitchen-like environment. They also use normals to classify regions and later fitting of cuboid primitives to represent dashboards. Model fitting is also performed by [23], as they try to detect buildings in a reconstruction process for large environments. [86] use a sampling technique to find basic shapes like planes, cylinders, cones and tori in a point cloud.

[87] segment trees in an outdoor environments, to use them as stable features for naviga-

tion. [88] use range images for segmentation, in combination with color information, for their autonomous exploration application.

The segmentation approach of [89] first performs a decomposition of space into regular cubes, then it tries to find planes with the use RANSAC principle. Finally a refinement step with a region growing strategy is applied. Further segmentation methods base on edges [90], curvature estimates [91] or smoothness constraints [92].

Mesh generation

Mesh generation techniques that are able to create a surface mesh, can be roughly categorized into two predominant directions. The first uses *implicit surfaces* and the second relies on *Delaunay triangulation*, resulting in a polygonal mesh representation of the surface.

Pioneering work in the field of **implicit surface modeling** was done by [49]. Tangent planes are fitted into local neighborhoods, then a marching cubes algorithm is used to compute the complete surface. [93] exploit the medial axis of the point set to improve implicit surface generation. It shows good results with reconstruction of geometrical complex objects. [94] propose a multi level partition approach, where the space is partitioned with an Octree and the local surface in a cell is approximated with a quadratic function. A wighting function blends the local patches together.

The approach of [46] and [47] uses the *Radial Basis Function (RBF)* as implicit function for splines. It creates the surface with a smoothing filter kernel that can cope inherently with noisy data sets. [89] propose a method that first segments the space with an Octree into smaller cells, similar to [94], and then a plane is fitted in the points of each cell. Finally the planes of the cells are merged into one coherent surface. This approach is designed for robotic indoor applications and has thus a focus on fast reconstruction. [84] capture the same idea of this approach, for an autonomous outdoor exploration application.

Some algorithms need **estimated normals** in the point cloud, for example the *Poisson reconstruction* method by [45]. Unlike approaches with *Radial Basis Function (RBF)*, it considers the whole point cloud at once. The generated output is a smooth curvature, even in presence of noise in the data sets. The *Algebraic Point Set surfaces (APSS)* algorithm [95] uses marching cubes in combination with a robust projection step, finalized by local fitting of spheres. *Robust Implicit Moving Least Squares (RIMLS)* [96] is similar to *APSS*, but it is able to better preserve sharp features on the surface.

Many approaches for surface mesh generation are based on the **Delaunay triangulation**. [97] contributes fundamentals for mesh generation algorithms by analyzing the properties for common geometric representations, including partitions with the Delaunay criterion. The author introduces the strategy to label tetrahedrons into *inside* and *outside* simplices. This categorization allows to deduce the surface of an object. The *CRUST* algorithm [98] uses Voronoi and Delaunay structures to create a surface mesh. But it has to perform a post-processing step, where point

normals have to be estimated. The *COCONE* [99] method is a theoretical and practical improvement of *CRUST*, because it does not need to estimate normals. *COCONE* exploits the labeling the strategy as presented by [97]. *CRUST* and *COCONE* do not produce *watertight* surface meshes. A *watertight* surface means that the boundary of an object is the closure of the object. In other words it has no holes in the mesh. But as trade-off *watertight* meshes might approximate the surface less well. *PowerCRUST* [100] produces *watertight* surfaces. It is based on a power diagram, rather then tetrahedrons. During its processing step it adds additional vertices (that satisfy the power distance criteria), which leads to a higher memory consumption. The smoothing and reconstruction might be computationally expensive. *TightCOCONE* [101] reliefs the burden of the extra points compared to *PowerCRUST*. First it runs the *COCONE* algorithm, then it refines the output by peeling off the as *outside* labeled simplices. The result is a *watertight* surface. Further improvements have been made with respect to noisy point clouds with the *RobustCOCONE* algorithm [102] that is provable more robust to small errors and the *EigenCRUST* [103] algorithm that can better handle noisy data.

The well known α-shapes algorithm by [104] is also based on the Delaunay triangulation, which is further refined by only including some simplices that conform to the α criterion. All simplices that intersect some sphere that does not include any points of the point cloud are carved away. The radius of the sphere is the squared α value. The α-shape algorithm is most suitable for uniform sampled point clouds.

Recent developments exploit further information than the pure point clouds. The *Pro-FORMA* approach [24] uses a single camera only, while a user has to rotate an object, which will be modeled. First a Delaunay tetrahedralization is calculated, then triangles are probabilistically carved away based on the visibility. Finally textures are added to the surface.

Labatut et al. [105] use camera images to create a first rough shape approximation, called *visual hull*. Afterwards the Dalaunay tetrahedra are labeled as empty or occupied, with respect to the *visual hull*. The result is a triangle mesh of the surface.

Other approaches that are independent of Delaunay triangulations have been proposed. [106] presents the ball-pivoting algorithm. It is a region growing strategy with a pivoting ball that can reconstruct a triangle mesh out of a point cloud, starting from a seed triangle. The Frontal Algorithm by [107], inspired by [108], calculates special points, and then the mesh is generated according to a rule set. An interesting contribution is the work of [109], as they developed an approach to measure how similar different meshes are. This is possibly a valuable contribution to compare the output of different mesh generation algorithms, to deduce the *best practice* approaches.

3.3 Public available libraries for 3D perception and modeling

This section surveys existing software libraries that are related to the 3D perception and modeling domain, biased towards the application in a mobile manipulation task. A library was selected for further investigation if one or more of the following requirements can be met:

- The library supports algorithms in the 3D perception and modeling domain. That means it can offer functionality for *depth perception, filtering, registration, segmentation, mesh generation* or *visualization*.
- The library supports data-types that are essential for this domain.
- The library is a potential user of the output data-types. As this library survey is biased towards mobile manipulation tasks, a library that can use the generated models for motion planning, collision checking, etc. is a good candidate for investigating the used data-types. This is motivated by the harmonization idea that shall foster the reuse of software in different contexts. In the mentioned mobile manipulation application the context would shift from the 3D perception and modeling domain to the mobile manipulation domain, but the data-types should remain the same.

Further motivation for selection is public availability, in the sense of open source. A closed source library, as it is intrinsic by definition, cannot be investigated, reused, refactored and finally re-published as open source software. A more relaxed requirement is the platform independence of the operating system. All libraries are implemented in C/C++ as this is the predominant programming language for computational expensive algorithms from the 3D perception and modeling or even the robotics domain. The libraries are roughly ordered with respect to the processing steps (cf. Figure 2.4).

6DSLAM

The *6D Simultaneous Localization and Mapping (6DSLAM)* library[36] implements methods to solve the SLAM problem with six degrees of freedom that mean for a 3D position and 3D orientation. The library is based on the work of [25] and was chosen because of the implementation of various ICP variants including a cached [63] and a parallel k-d tree as an efficient way to address the correspondence problem.

MRPT

The *Mobile Robot Programming Toolkit (MRPT)*[37] is a library that allows to create applications for robotic tasks. Therefore it comprises functionality for obstacle avoidance, SLAM or motion planning. With an implementation of the MonoSLAM [18] algorithm it is able to perform

[36] `http://slam6d.sourceforge.net/`
[37] `http://babel.isa.uma.es/mrpt/index.php/Main_Page`

depth perception with a single camera only. The MRPT library also includes an implementation of ICP, which forms the main reason for selection.

IVT

The *Integrating Vision Toolkit (IVT)*[38] is a library with a focus on computer vision. It supports algorithms for stereo reconstruction, blob feature detection like SIFT and it has an ICP implementation. Especially the latter one is the motivation for selection.

FAIR

The *Fraunhofer Autonomous Intelligent Robotics Library (FAIR)*[39] was initially developed by the Fraunhofer-Institut für Intelligente Analyse- und Informationssysteme (IAIS) as a software package for the *VolksBot* [110] research project. The library supports many filtering algorithms, has an ICP implementation and offers drivers for depth perception like a laser scanner or Time-of-Flight camera interfaces. The presence of the previously mentioned algorithms which are specifically designed for robotics are the main reason why this library is included into the survey.

ROS

The *Robot Operating System (ROS)*[40] is a complete environment for robotic applications, rather than a traditional software library. It offers a middleware in the sense of a network of distributed computers that appears to the user as one coherent system [111]. So called *ROS nodes* communicate to each other via *messages* and *services*. The formats of these messages are defined in special text files, and can be seen as a standardized data-types in the ROS environment. ROS has a set of tools that make the programming easier, for example the *rosmake* tool is able to resolve dependencies to other ROS modules. The build process is also able to automatically transfers the *message* definitions into source code.

The algorithms and functionality can be found in different repositories, which results in a huge amount of functionality. Software is organized in *packages*, which are grouped thematically into *stacks*. For depth perception for example a *laser pipeline* is offered that is able to assemble point clouds. Different filtering techniques like normal estimation, ICP for the registration and several segmentation algorithms are available. But there is (yet) limited mesh generation functionality. As ROS has *stacks* for mobile manipulation planning algorithms it also serves as a potential user of triangle meshes.

The ROS software is not fully platform independent, as it only supports Linux derivates and Mac OS X. Windows is not natively supported, but the underlying build system uses the platform independent *CMake* build system, thus there are no conceptual barriers to port the code to Windows.

[38] `http://ivt.sourceforge.net/`
[39] `http://sourceforge.net/projects/openvolksbot/`
[40] `http://www.ros.org/wiki/`

ROS is currently one of the most important open source projects for robotics, so the selection for investigation is obvious. Please note that the investigated version is v0.7. As ROS is a rapidly changing and evolving project, some of the previous mentioned statements might get obsolete.

VTK

The *Visualization Tool Kit (VTK)*[41], developed by Kitware Inc.[42], is a software for 2D and 3D data processing and visualization. It has its main application in biomedical imaging, like for example visualization of regions of the brain. The library has implementations for polygon reductions, implicit modeling and Delaunay triangulation. It is selected here because it supports mesh generation algorithms.

ITK

As extension to VTK the *Insight Segmentation and Registration Toolkit (ITK)*[43] supports registration and segmentation algorithms for images. Although the tool kit uses images as data-types, most algorithms allows n-dimensional data as input. As it implements the registration and segmentation stages it is chosen as a suitable library for the 3D perception and modeling domain.

Meshlab/VGC

Meshlab[44] is a program for manipulation and interactive editing of 3D models. It is based on the core library for mesh processing *VGC*[45]. The VGC library is developed by the Visual Computing Lab[46] of the ISTI - institute of the Italian National Research Council. Meshlab has implementations for mesh cleaning, reduction re-meshing, mesh generation and registration by the ICP algorithm.

Although segmentation techniques are limited to non-automatic procedures that means user interaction is required, the numerous supported data-tapes, the rich functionality with respect to meshing and the presence of registration functionality lead to the selection of Meshlab/VGC.

CGAL

The *Computational Geometry Algorithms Library (CGAL)*[47] is one of most important open source projects for algorithms in the computational geometry domain. Thus it supports algorithms for 2D and 3D triangulations, including Dalaunay triangulation and α-shapes, Poisson surface reconstruction, mesh simplification, mesh subdivision, normal estimation and many more. As

[41] `http://www.vtk.org/`
[42] Kitware Inc. also develops the *CMake* build system.
[43] `http://www.itk.org/`
[44] `http://meshlab.sourceforge.net/`
[45] `http://vcg.sourceforge.net/index.php/Main_Page`
[46] `http://vcg.isti.cnr.it/joomla/index.php`
[47] `http://www.cgal.org/`

this library serves a rich functionality for mesh generation it is a reasonable candidate for further investigation within this work.

Gmsh

Gmsh[48] is a modeling tool with graphical user interaction. Its main purpose is to easily manually create 3D meshes. To assist the user it offers some filtering techniques like the Octree reduction or meshing algorithms like Delauny triangulation. Beside the filtering and registration capabilities this tool was chosen, because it implements various kinds of data-types for 3D models.

Qhull

The Qhull[49] library implements the *quick hull*[50] algorithm for generation of a convex hull of a point set. It has meshing algorithms for Voronoi digrams and Delaunay triangulation. This and the fact that it is already iterated into the BRICS_MM library motivates the consideration of this library.

OpenSceneGraph

The *OpenSceneGraph (OSG)*[51] project is a powerful library for 3D model visualization. It is essentially an object-oriented wrapper for the platform independent *OpenGL* interface. OSG has the concept of a *scenegraph* that means data is stored in a hierarchically manner by a tree structure. The nodes store either 3D data, which are then called *Geodes* (stands for Geometry Node), or references to child nodes. With this approach it is possible to arrange geometries. For example in a kitchen-like environment a cup, a table or a dashboard could be each captured by a different node, whereas the root note represents the kitchen environment. Even if the same geometry like for a dish needs to be applied several times, the data that describes the geometry, does not need to be duplicated as the encapsulating node could be referenced multiple times, but with different pose information [112]. Theses abilities are certainly useful for the robotic domain.

OSG was chosen for this work, because of the features mentioned above, the good support for different operating systems, and because it is already successfully integrated into the BRICS_MM library.

CoPP/BRICS_MM

The *Components for Path Planning (CoPP)* library[52] is a framework for mobile manipulation planning algorithms, initially developed by [113]. This library is a potential user of the output

[48] `http://www.geuz.org/gmsh/`
[49] `http://www.qhull.org/`
[50] What the quick sort is for sorting problems, is the quick hull for convex hull generation.
[51] `http://www.openscenegraph.org/projects/osg`
[52] `http://sourceforge.net/projects/copp/`

data from the 3D perception and modeling domain. The reconstructed 3D objects and environments could serve as input for the planning facilities of CoPP. Parts of the CoPP library have been refactored or reused by the BRICS_MM library. Both libraries share the same data-type for triangle mesh representation. Please note that BRICS_MM is not yet available as open source software therefore it is not listed as a separate entry in this survey, but it will be made public in the near future.

Openrave

Openrave[53] is similar to CoPP or BRICS_MM. It is a simulation framework for mobile manipulation algorithms. As for CoPP and BRICS_MM, Openrave might be a potential user of 3D models, encoded as a triangle meshes for planners or collision checkers.

KDL

The *Kinematics and Dynamics Library (KDL)*[54], is a library to model rigid body kinematic chain representation and calculation, and thus is more related to manipulation rather than 3D perception and modeling. KDL is a part of the *Orocos*[55] software component framework for robotic tasks. The motivation for selection is that KDL is a robotic library that heavily deals with 3D points and frame transformations. It is worth to look, how the library represent basic data-types like 3D points.

ANN

The *Approximated Nearest Neighborhood (ANN)*[56] library is a well known library for k-Nearest Neighbor search, developed by [64]. It focuses on efficient high-dimensional k-Nearest Neighbor search tasks. The Nearest Neighbor search is a commonly used sub-algorithm for many algorithms, like for example normal estimation or as solution for the correspondence problem in the ICP algorithm (cf. Algorithm 2.1, line 2). ANN is chosen for two reasons: first it implements a state-of-the-art solution for k-Nearest Neighbor search and second it often serves as reference for comparison for more recent approaches, as for example in [25] or in [69].

FLANN

The *Fast Library for Approximate Nearest Neighbors (FLANN)*[57] is a recently developed solution by [69] for the k-Nearest Neighbor search. It has implementations for hierarchical k-means or multiple random k-d trees. The FLANN library has a remarkable feature: it automatically detects the best suitable search algorithm. This is interesting, as the library has to use metrics to

[53] `http://openrave.programmingvision.com/index.php/Main_Page`
[54] `http://www.orocos.org/kdl`
[55] `http://www.orocos.org/`
[56] `http://www.cs.umd.edu/~mount/ANN/`
[57] `http://people.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN`

determine which algorithm fits best. This has a strong synergy to the concept of identifying what is *best practice* by measuring the performance of algorithms.

STANN

The *Simple, Thread-safe Approximate Nearest Neighbor (STANN)*[58] library for k-Nearest Neighbor search addressed the computational complexity of this problem by exploiting parallel computation with multiple threads. This library is included in this survey as it implements an efficient solution for k-Nearest Neighbor search on modern computer hardware that tends to have more and more parallels computing facilities.

Comparison table

The following comparison table (cf. Table 3.1) summarizes all libraries and indicates which stage(s) of the 3D perception and modeling stages they support. There is an additional column to clarify which version is used. This is important as the presented libraries might have changes in the future and some facts or statements about them might get obsolete. There are also columns to show if a library was developed with a robotic context in mind, or if it might serve as a ”user” of the used data-types of the 3D perception and modeling domain. Please note that there is no single library that has all processing stages for 3D perception and modeling domain implemented. The presented libraries will serve as a foundation to find harmonized data-types and interfaces for reusable and measurable software modules. By benchmarking those modules the *best practice* algorithms for 3D perception and modeling shall be identified.

[58] `http://sites.google.com/a/compgeom.com/stann/`

Table 3.1: Summary of public available libraries and their contribution to the 3D perception and modeling domain.

Library	Version	Depth perception	Filtering	Registration	Segment-ation	Mesh generation	Visual-ization	Potential user	Robotic library	Notes
6DSLAM	v1.0	✓	✓	✓	-	-	✓	-	✓	
MRPT	v0.7.1	✓	-	✓	-	-	✓	-	✓	
IVT	v1.3.5	✓	-*	✓	-	-	-*	-	-	* = only 2D
Fair	rev4	✓	✓	✓	-*	-	✓	-	✓	* = only 2D
ROS	v0.7	✓	✓	✓	✓	-	✓	✓	✓	
VTK	v5.4.2	-	✓	-	-	✓	✓	-	-	
ITK	v3.16.0	-	✓	✓	✓	-	-	-	-	
Meshlab	v1.2.1	-	✓	✓	-	✓	✓	-	-	
CGAL	v3.4.0	-	✓	-	-	✓	-*	-	-	* = uses *geomview*, which is only available for Linux
Gmsh	v2.4.2	-	✓	-	-	✓	✓	-	-	
Qhull	rev444	-	-	-	-	✓	-	-	-	already integrated into BRICS_MM
CoPP/BRICS_MM	rev444	-	-	-	-	-	✓	✓	✓	user of triangle mesh
openrave	?	-	-	-	-	-	✓	✓	✓	user of triangle mesh
OSG	v2.8.2	-	-	-	-	✓	✓	-	-	has *scenegraph* concept
KDL	v1.0.0	-	-	-	-	-	-	✓	✓	user of 3D point
ANN	v1.1.1	-	-	-	-	-	-	-	-	k-NN Search
FLANN	v1.2	-	-	-	-	-	-	-	-	k-NN Search
STANN	v0.71a	-	-	-	-	-	-	-	-	k-NN Search

Chapter 4

CONCEPT

In this Chapter the process of identifying *best practice* algorithms is explained. First the general approach is depicted, then it is applied to the domain of 3D perception and modeling.

4.1 Basic Approach

The presented approach is inspired by the work of [21]. The process of identifying *best practice* algorithms can be categorized into different phases. There are five steps: *Exploration*, *Harmonization*, *Refactoring*, *Integration* and *Evaluation*. The goal of this process is to have a framework of software components that allows to replace atomic elements to easily create a set of benchmarks that enables to compare and judge the algorithms. These benchmarks are the base for impartial deduction of *best practice* for a given task. Summarized in one sentence: ”Software engineering principles will be applied to existing algorithms to make them easier to compare”.

4.1.1 Exploration

In the *Exploration* phase, knowledge about the algorithmic domain has to be acquired. Without a solid understanding of the field, decomposition of algorithms into atomic and reusable parts would be impossible. To gather information about the domain a **state-of-the-art literature survey** has to be conducted. The results of the survey for algorithms for 3D perception and modeling can found in Section 3.2.

As the goal of this approach is to refactor existing algorithms, a survey about **state-of-the-art software libraries** that implement relevant algorithms has to be performed. The found libraries form the basis of analysis and refactoring of common data-types and common algorithms. Note that this approach is limited to open-source public available libraries. If predominant algorithms are not available as source code, algorithms might have to be reimplemented by their description in the literature, as long as the description is expressive enough and no hidden assumptions are unspecified. Section 3.3 lists recent libraries that contribute to the 3D perception and modeling domain.

4.1.2 Harmonization

The *Harmonization* phase tries to find harmonized data-types and interfaces for atomic elements. It can be further subdivided into the three following steps:

1. Identify **common data-types**. Identify the commonly used data-types, then investigate how they are represented in existing libraries. The attributes and functionality that is commonly used by existing libraries will form the requirements for the harmonized data-type. These requirement will have to be met in the *Refactoring* and *Integration* step. That means it must be implemented. The harmonized data-types will be reused for the harmonized interfaces for algorithms. Requirements for harmonized data-types needed for 3D perception and modeling applications are presented in Section 4.3

2. Identify **black-boxes**. This is the step of de-modularizing algorithms into atomic elements. The knowledge how they are decomposed comes from the *Exploration* phase. The most common modules shall be found and represented as stand alone modules. In this work these modules are represented as UML software component diagrams. The design choices should be driven with openness and flexibility in mind. Section 4.4 shows the most common black-boxes for 3D perception and modeling.

3. Identify common **interfaces**. The atomic elements identified in the previous step need to communicate to other atomic elements. To get access to a component one has to use a set of interfaces. These interfaces should be harmonized to be reusable. One important aspect here is that different clients need different interfaces. For example a potential user just needs the very basic interface that is most convenient to use, but another user needs to know about more internal details of an algorithms. In this case different interfaces should be offered. Harmonized interfaces for algorithms of the 3D perception and modeling domain can be found in Section 4.4.

All algorithms and data-types are designed with a potential user of in mind, like for example a mobile manipulation planning library that needs a 3D model of the environment to create paths or trajectories for the hardware platform. To foster openness, flexibility and reusability of the refactored algorithms, a modularization is aspired with respect to the principles *Coordination*, *Configuration*, *Computation*, and *Communication* (*4Cs*) [114, 115, 116]. The aspect of *Computation* describes the implemented behavior, in this case an algorithm from the 3D perception and modeling domain. The *Configuration* defines which *Computations* are performed that means which algorithms or sub-algorithms are actually used to solve a certain problem. *Communication* describes how and what the *Computation* units communicate with each other. Typically data-types are the entities which are communicated between algorithms. *Coordination* can be seen on a more abstract level as it specifies when *Communication* must take place. Applied to the 3D perception and modeling domain that could mean that a particular algorithm is only invoked in specified situations. For example a robot does not need to have have precise model for grasping a cup while it has not yet reached the kitchen, where the cup is located.

The process of modularization, decoupling and abstracting to more general interfaces certainly involves trade-offs between usability or generality and highly coupled or optimized algorithms. But as the intention is to find *best practice* algorithms the decoupling has a higher priority over highly optimized algorithms. However, in a robot development process an optimization step might follow after the best practice algorithms have been determined for a specific application.

4.1.3 Refactoring

The *Refactoring* step can be characterized by **filling black boxes with life**. That means the desired behavior for an algorithm must be defined by implementing it. One possible solution is to create a wrapper or adapter [9] to an algorithm from an existing library, which are already known from the *Exploration* phase. If this is not possible, the behavior must be completely reimplemented. The outcome of the *Refactoring* phase for 3D perception and modeling is presented in Chapter 5.

4.1.4 Integration

In preparation of benchmarking the implemented algorithms, they need to be **integrated into a test-bed**. This test-bed might be a real robot, a simulation framework or a software framework that can prepare data sets to be used as input for a benchmark. The goal is to embed the algorithms into an environment that allows to get access to *best practice*. Although the efforts to integrate algorithms to existing frameworks or libraries are typically time consuming, they give insights about how well the harmonized data-types and interfaces cooperate with existing code. From a developers point of view the *Refactoring* and *Integration* determine the main work during software development. Integration of the 3D perception and modeling algorithms is performed, in the form of a library called *BRICS_3D*, which contains the refactored algorithms and capabilities to use data sets for benchmarks. Details about the *Integration* are shown in Section 5. Section 6.1 gives further information about the used test-bed.

4.1.5 Evaluation

The final step is the *Evaluation* or benchmarking of the refactored and harmonized algorithms. These benchmarks can be performed on a real robot, within a simulation framework or with recorded or artificial data sets. Details about the methodology of benchmarking in robotics is currently a research subject on its own, as mentioned in Section 3.1. As the BRICS project will make the refactored algorithms public, the code can be used to produce *comparable* benchmarks, as new algorithms can be integrated into the BRICS_3D library and compared to previous results. Also the problem of *repeatability* can be relieved as everyone can download the code and re-run the benchmarks. An initial set of benchmarks for 3D perception and modeling, with data sets, is presented in Chapter 6.

4.2 Review of 3D perception an modeling elements

The domain of 3D perception and modeling can be subdivided into several subcategories (cf. 2.3): *depth perception, filtering, registration, segmentation, mesh generation* and *visualization*. Arranged as a reconstruction pipeline, it can transform 3D point clouds into triangle meshes. All algorithms have to work on the same data-types and therefore it is beneficial to have harmonized data-types for Cartesian point clouds and triangle meshes. The harmonized data-types shall enable reusability in other robotic software projects. If all algorithms work on the same data representation, the algorithms are better comparable. Another benefit is that different libraries, working on harmonized data-types, are easier to integrate to each other, as no data conversions have to be performed.

This work identifies common algorithms for the different processing stages. *Depth perception* algorithms are not refactored within this work, as these algorithms depend on the underlying hardware technology. Several common algorithms exist in the *filtering* stage: normal estimation, noise reduction and Octree filtering. The latter one is of most importance, as it also serves as a method to partition the space, which is required by some meshing algorithms. Thus it is further investigated in this work. The *registration* stage is dominated by the ICP algorithm. Therefore, the ICP will be refactored in this work and decomposed into sub-algorithms. The *segmentation* stage is left for future work, as this step is regarded as optionally in a surface reconstruction task for mobile manipulation tasks. Many mesh generation algorithms rely on Delaunay triangulation and this is why is regarded as atomic algorithm. *Visualization* functionality does not need to be refactored because standardized interfaces already exist.

A general algorithm that does not fall in one of the above categories, is the k-Nearest Neighbors search algorithm. It is used by *filtering*, *registration*, *segmentation* and *mesh generation* algorithms. The k-Nearest Neighbors search is an atomic algorithmic component.

The remainder of this Chapter will present the common data-types and algorithms in further detail.

4.3 Harmonization of common data-types

The 3D perception and modeling domain with bias toward mobile manipulation applications has three major data-types:

- The *Cartesian point*, as a representation of a surface sample or a distinct feature in the three dimensional space.
- *Cartesian point cloud*, as collection of Cartesian points. This representation is often used in 3D perception and modeling approaches, as it is a natural way to represent aggregated laser range data.

- *Triangle mesh*, as an approximation of a surface. Most polygonal mesh generation algorithms create triangle meshes [117], especially as many use the Delaunay triangulation. Triangle meshes are also one of the most common representations for collision checkers [113]. And even for mesh compression methods, triangle meshes are the predominant data-type [118].

The following section will investigate each of these data-types.

4.3.1 Cartesian point representation

A minimalistic implementation of a Cartesian point for three dimensions at least has a representation of the x, y and z coordinates. The investigated libraries are those that primarily work with points. In this case the libraries with *registration* capabilities and KDL in the role of a potential user library.

Representation in existing libraries

- **6DSLAM**: The point representation in the 6DSLAM library is reduced to the very basic functionality. Its main purpose is to work with the ICP algorithm. The primitive used data-type for the x, y and z coordinates is **double** (cf. Figure A.1). The point class has functionality for input and output streaming and allows to apply a homogeneous transformation to modify the position of a point.

- **MRPT**: MRPT uses an inheritance hierarchy (cf. Figure A.2) to represent points. This hierarchy intermixes points and poses as the generalization of both is a class named `CPoseOrPoint`. The coordinates are encoded with the primitive type **double**. The class provides basic vector functionality. Beside this class, there is also a *light weight* version of a point, called `TPoint3D`. Probably to have a more convenient version, as it serves as input in the constructor or a more efficient way store the information.

- **IVT**: In the IVT points are treated as a vector with three elements (cf. Figure A.3). The coordinate data-type is represented as **float** and is aggregated in a struct. Basic matrix operation functionality is decoupled and implemented in a separate class.

- **FAIR**: In the FAIR library a Cartesian point is implemented in a simple struct with three **double** values (cf. Figure A.4). It contains a pointer to a struct with additional information, like color, intensity, etc. The representation of coordinates is realized with **double**. The functionality vector algebra is encoded in the point cloud container (cf. Figure A.12). The usage of the point struct is optimized towards the ICP algorithm.

- **ROS**: In ROS data-types are typically defined in message files (*.msg* file extension). These message definitions are translated while compilation into source code, header files respectively. The *Point32.msg* defines a Cartesian point:

```
# This contains the position of a point in free space(with 32 bits
    of precision).
# It is recommeded to use Point wherever possible instead of
   Point32.
#
# This recommendation is to promote interoperability.
#
# This message is designed to take up less space when sending
# lots of points at once, as in the case of a PointCloud.

float32 x
float32 y
float32 z
```

Listing 4.1: ROS message definition for a Cartesian point.

The generated code (cf. Figure A.5) results in a representation with three **float** values. Further functionality comprises streaming in the sense that it can be serialized to be sent to another *node* over a network. Though the point message serves as common interface within ROS, the Cartesian point representation in some packages (e.g. ”Gmapping ”or ”convex_decomposition”) might have other internal representations which need to be converted to or from the message data-type.

- **ITK**: The ITK uses a C++ template for a Cartesian point (cf. Figure A.6). The parameters, dimension and the underlying type need to be defined by a user of this data structure. As ITK is a non-robotic library, but one that provides algorithms for n-dimensional problems, it does not maintain point representation for different dimensions. Functionality for vector processing like addition, subtraction and test for equality are provided.

- **Meshlab**: Similar to ITK, Meshlab uses a C++ template (cf. Figure A.7). But only the primitive types for the x, y and z coordinates can be defined. The dimension is cannot be changed within the template. The implementation is dependent on the template-based header library *Eigen*[59] for Matrix computations.

- **KDL**: Although KDL does not directly address the 3D perception and modeling domain, there might be robotic applications that have to cope with both, kinematic issues and perception tasks. So why should they not be able to use or easily interface with a common point representation? KDL stores the coordinates in an array of **double** values (cf. Figure A.8). Data is accessible via according getter and setter methods. KDL supports frame transformations and basic vector operations.

[59] `http://eigen.tuxfamily.org/index.php?title=Main_Page`

Harmonized representation in BRICS

The harmonized Cartesian point representation in BRICS tries to capture what is common in the previous discussed point representation. Most libraries use a representation with simple x, y, and z variables, especially those that use the points as input for the ICP algorithms in a robotic context. The predominant primitive data-type is **double**, but the harmonized version should have some flexibility to switch to **float**, as this is less memory intensive and potentially faster on the same hardware platforms. For example this consideration is of interest for (industry) PCs with 32bit processors, as there is the possibility to switch to a **float** representation that better fits to 32bit processors. This design choice in a robot development is typically done once for a specific system, so it is reasonable to configure this during compile time and not dynamically.

Basic matrix operation functionality is commonly used among the investigated libraries, as the Cartesian point also serves as vector of dimensionality three. Thus a harmonized point should support simple vector algebra as addition, subtraction and multiplication with a scalar. An important feature is to implement multiplication with a matrix, to enable homogeneous transformation, to rotate and translate a point. Although streaming support is not regarded as a commonly available feature, it is convenient to have it. It allows to easily dump the output to relief debugging or logging activities, and it makes conversions to other point representations simpler.

A harmonized data-type must preserve flexibility to future extension. A possible extensions of a point could be color information, (like red, green and blue channels), estimated normals, weights, probabilities, flags is a point is (in)valid or a feature vector descriptor for distinctive properties (like SIFT for images), to name some. It is possible to implement this via class inheritance, but with growing requirements the inheritance hierarchy would have a combinatorial explosion of possible combinations. To overcome this problem the *decorator* software pattern will be applied [9]. This allows to wrap a point with another point skin. The outer appearance is still a point but the inner representation has additional information. Another advantage of this technique is that new information can be added dynamically, just by adding another decoration layer.

The representation of the investigated libraries and the resulting requirements for a harmonized representation are summarized in Table 4.1. To represent a scene of an environment, single Cartesian points are often grouped into point clouds to approximate a surface.

Table 4.1: Comparison table for Cartesian point representation in existing libraries and requirements for BRICS.

Library	Simple x,y,z representation	Primitive data-type	Used for ICP	Supports matrix operations	Streaming support	Additional information (e.g. color)	Robotic library	Notes
6DSLAM	✓	double	✓	✓	✓	-	✓	
MRPT	-	double	✓	✓	-	-	✓	uses point hierarchy; point cloud uses another internal representation
IVT	✓	float	✓	-	-	-	-	
FAIR	✓	double	✓	-*	-	✓	✓	* = matrix functionality in point cloud class encoded
ROS	✓	float	✓	-	✓	-	✓	generated from *message* file
ITK	-*	arbitrary*	✓	✓	-	-	-	* = uses C++ templates for type and dimension
Meshlab	-	arbitrary*	✓	-	-	-	-	* = use C++ templates for type; depends on *Eigen* library
KDL	✓	double	-	✓	-	-	✓	Depends on *Eigen* library
Requirements for BRICS	✓	double and float	✓	✓	✓	✓	✓	Decorator pattern for additional information

4.3.2 Cartesian point cloud representation

The Cartesian point cloud is a set of Cartesian points. Three dimensional point clouds are already a 3D model of the world, as the points can be seen as samples of the perceived surfaces. A point cloud representation at least has to store a set of points. The same libraries as in the Cartesion point 3D harmonization are investigated again, except for the KDL representation, which is not intended to be used in a point cloud context.

Representation in existing libraries

- **6DSLAM**: The point clouds are encapsulated in a **`Scan`** class (cf. Figure A.9). This already gives a strong semantic on the underling depth perception technology: a laser range scanner. The **`Scan`** class stores the points in a simple vector of points and has an interface for point reduction filtering by a k-d tree. The reduced points are then stored in a separate double array, to exploit optimized data access in later processing steps. Functionality for homogeneous transformation and streaming are supported.

- **MRPT**: As MRPT typically uses point clouds for navigation and SLAM applications, the

point clouds are represented as *maps*. The *maps* are structured in an abstract inheritance hierarchy, to allow a simple point representation to be exchanged for example with color annotated points (cf. Figure A.10). An interesting observation is that the Cartesian point representation, as presented in the preceding section, is not reused. Instead each coordinate is stored in its own vector of **float** values. Similar to the 6DSLAM library, MRPT offers an interface to a k-d tree construction in the map representation, but in addition an interface for point registration is available. The result is a point cloud representation that is designed for robotic navigation tasks, but not for reuse in other contexts. Streaming functionality in the sense of loading and saving from and to files is supported.

- **IVT**: The IVT library does not have a dedicated class for point clouds. To represent sets of points for example in the interface for the ICP algorithm in the class **`CICP`** (cf. Figure A.11), a pointer to an array of points is used. As no information about the size of the data array is addressed directly, this might lead to access of data that is out of bounds of the array.

- **FAIR**: In the FAIR library point clouds are stored in a vector of pointers to points that is embedded in the **`CCartesianPointCloud`** class (cf. Figure A.12). Functionality for homogeneous transformation of point clouds, size reduction and streaming is available in the point cloud representation.

- **ROS**: Similar to the point representation (cf. Listing 4.2), ROS defines point clouds in messages. The corresponding message *PointCloud.msg* is listed as follows:

```
#This message holds a collection of 3d points, plus optional
    additional information about each point.
#Each Point32 should be interpreted as a 3d point in the frame
    given in the header

Header header
geometry_msgs/Point32[] points  #Array of 3d points
ChannelFloat32[] channels       #Each channel should have the same
     number of elements as points array, and the data in each
    channel should correspond 1:1 with each point
```

Listing 4.2: ROS message definition for a Cartesian point cloud.

The generated source code (cf. Figure A.13) for C++ contains a vector of points. As this class is a ROS *message* it supports streaming capabilities. Further functionality like frame transformations is shifted to other classes.

- **ITK**: ITK has a **`PointSet`** class to represent Cartasion point clouds (cf. Figure A.14). As for the Cartesian point, a C++ template is used to declare type and dimension by the user of the ITK library. The internal point set representation is encapsulated into a container class.

Streaming capabilities are supported by the **`PrintSelf`** method. A function for nearest point search is also available.

- **Meshlab**: Meshlab does not represent point clouds in a dedicated class. As seen in the interface for registration, **`PointMatching`** (cf. Figure A.15), the matching process requires a vector of points.

Harmonized representation in BRICS

The harmonized Cartesian point cloud will have a vector of points as this is the most common, among the investigated libraries, and the most convenient way to represent it. Supported operations should be streaming capabilities, similar to the Cartesian point requirements (cf. Section 4.3.1), homogeneous transformation as this just means to forward the transformation to each point in the cloud, and simple manipulation operations for example to add new points.

Functionality for point size reduction will not be a part in the representation as it might not be needed for all 3D perception and modeling applications. This is moved to the algorithms of the *Filtering* stage. Table 4.2 summarizes representations in existing libraries and resulting requirements for the harmonized point cloud. Beside the Cartesian point cloud, triangle meshes are a common way to model a 3D scene.

Table 4.2: Comparison table for Cartesian point cloud representation in existing libraries and requirements for BRICS.

Library	Dedicated class for point cloud	Representation	Supports transfor-mations	Stream-ing support	Point reduction	Robotic library	Notes
6DSLAM	✓	Vector of points	✓	✓	✓	✓	point cloud is seen as a *scan*
MRPT	✓	3 vectors of floats (one for each coordinate)	-	✓	✓	✓	point cloud is part of *map* hierarchy; supports color information
IVT	-	array of points	-	-	-	-	
FAIR	✓	vector of pointers to points	✓	✓	✓	✓	
ROS	✓	vector of points	-	✓	-	✓	generated from *message* file
ITK	✓	container class	-	-	-	-	
Meshlab	-	vector of points	-	-	-	-	
Require-ments for BRICS	✓	vector of points	✓	✓	-	✓	

4.3.3 Triangle mesh representation

Typically the triangle and triangle mesh representations are strongly related in the sense that some libraries use an *explicit* triangle representation and others an *implicit* representation. *Explicit* means that a triangle is represented by a dedicated class. A triangle mesh is then a set of triangle objects. Often the triangles classes are composed of the three vertices - the edges between the vertices represent the triangle. The *implicit* representation does not have a special class for a triangle. It uses a list of points and a list of indices that refer to the points. Three consecutive values in the indices list describe indices of three points in the corresponding point list. Thus treating triangles and triangle meshes separately, as it is done for the Cartesian point and Cartesian point cloud, is not straight forward. The following section will primarily investigate libraries that support mesh generation, mesh visualization or are potential *users* of a triangle mesh, like motion planing libraries for mobile manipulation.

Representation in existing libraries

- **VTK**: The VTK library offers two container classes for mesh representations: **`vtkPolyData`** and **`vtkUnstructuredGrid`**. Both can be used for sets of polygons, lines or points. A triangle set is composed of dedicated objects for triangles (cf. Figure A.16). The internal representation of a triangle is encoded with a set of line segments, rather then three triangle vertices. Streaming functionality is supported, but homogeneous transformations are not included in the mesh interfaces.

- **Meshlab**: Meshlab supports many 3D model representations like for example point sets, edge mesh, triangle mesh or tetrahedral meshes. Meshlab has both representations: *implicit* and *explicit*. For the implicit variant (cf. Figure A.18) the user has to self-define the vertex and face type. An according source code would look like the following statement, with **`MyVertex`** and **`MyFace`** as user-defined classes:

  ```
  class MyMesh: public TriMesh< vector<MyVertex>, vector<MyFace> >;
  ```

  With the user-defined type for the vertices, Meshlab supports compile-time flexibility to enhance a vertex by color, normals etc. The explicit representation of a triangle uses an array with three vertices (cf. Figure A.17). The vertices are C++ templates, so the user can define the primitive type of the coordinates.

  As both the implicit and the explicit way to represent triangle meshes have a common access method it is possible to exchange the underlying representation. At least some algorithms can work with both representations.

- **CGAL**: The CGAL library makes excessively use of C++ templates. CGAL uses a container type to encapsulate meshes, where each triangle is regarded as a *facet*. Facets consist of *half-edges*. The conversion between different representations, as well as streaming capabilities, are available.

- **Gmsh**: Similar to Meshlab, Gmsh supports many 3D shapes, approximated with lines, triangles, quadrangles, tetrahedra, prisms, hexahedra and pyramids. To describe a triangle mesh, Gmsh uses an *explicit* representation. The mesh `Mmodel` consists of a set of `MTriangle` objects (cf. Figure A.19). A single triangle has an array of pointers to the vertices. Gmsh has extdended versions of a triangle, realized via inheritance. The triangle interface provides getter and conversion methods for faces or various other representation. It also collaborates with other libraries like VTK.

- **Qhull**: Qhull implements implicit triangle meshes in a double linked list of structs (cf. Figure A.20). Each vertex is considered an array of coordinates with dimension three. As Qhull does not use classes at all[60] it cannot provide functionality for conversion to other representations or steaming within the data-type representation.

- **CoPP/BRICS_MM**: The triangle mesh representation in CoPP and BRICS_MM are the same. The triangle is explicitly modeled with three variables of a three-dimensional vector type: `Vector3 p1`, `Vector3 p2` and `Vector3 p3` (cf. Figure A.21). This representation has facilities for normal computation, centroid computation, normal computation area computation and streaming.

- **openrave**: openrave has both: the *explicit* and *implicit* variant. The explicit `TRIANGLE` class consists of three vectors, each representing a vertex (cf. Figure A.22). An interesting observation is that openrave does not seem to use the `TRIANGLE` class in a mesh. In fact all meshes are represented with the *implicit* representation in the `TRIMESH` (cf. Figure A.23). Here a vector of vertices and a vector of corresponding vertices is used. The triangle mesh has streaming support but no conversion to other 3D models are available, as no other 3D models are used for the mobile manipulation motion planners.

- **OSG**: The OSG library encodes a triangle mesh in the `TriangleMesh` class, which is a specialization of a `Shape` super class (cf. Figure A.24). The mesh uses a `Vec3Array` for vector vertices, while each vertex is a three dimensional **float** array. To form the triangles, an index list `IndexArray` for the corresponding triangle indices is used. This triangle mesh class does not support transformation to other 3D models, nor does it support streaming directly. Typically a `TriangleMesh` is hooked into a *Geode* (cf. Section 3.3) and is forwarded to the OpenGL layer to be rendered.

- **ROS**: ROS has a *geometric_shapes* package in the *motion_planning_common* stack. This package considers a `Mesh` as a specialization of a shape (cf. Figure A.25). The triangle mesh is represented in an *implicit* manner. An array of **double** values store the vertices. Each consecutive three values form a vertex. Similar to this, each three consecutive **int**

[60] Qhull was started in the mid 1990s as a C project that means before C++ was even standardized in 1998.

values form a triangle in the array for the indices. Conversion to other 3D models and streaming support are not included in the triangle mesh class.

Harmonized representation in BRICS

Reviewing the previously analyzed libraries, there are two common ways to represent a triangle mesh: First, the *implicit* representation with a vector of vertices and a vector of indices. Three consecutive indices, referencing the vertices vector, form a triangle. The advantage is the memory efficient storage, as vertices do not need to be inserted multiple times into the mesh if one vertex belongs to several triangles. The disadvantage is that both vectors have to be carefully maintained while adding or removing triangles. Furthermore it is not flexible for future extension, because a triangle might have additional information like normals, color, a validity flag, a probability or a texture reference.

The *explicit* version has a vector of triangles, whereas each triangle consists of three vertices. The representation is more flexible in the sense that a basic triangle class can be extended or *decorated* in future developments, similar to the Cartesian point. But on the other hand it might be less memory efficient.

As an interesting observation, Meshlab already supports both types of representation. Inspired by this, a harmonized triangle mesh should support both representations. An abstract class allows access with a common interface, so a potential user can choose which implementation fits most to an application or a problem. However that does not necessarily means both representations are always fully exchangeable.

A common functionality is the streaming support which allows to easily read and write data to standard output, files or other implementations of triangle meshes. A support of transformation to other representations of 3D models (e.g. splines) is not feasible, because first, the triangle mesh is already the predominant representation, and second, only few libraries support this feature. As an additional feature a harmonized triangle mesh should support homogeneous coordinates transformations similar to the Cartesian point clouds. The motivation for this is that a potential user might create a mesh first, or the used depth perception device already delivers a mesh, and then register it into a global frame with an appropriate algorithm.

Table 4.3 recapitulates the common representations and capabilities, and the resulting requirements for a harmonized triangle mesh data-type.

Note that the concept for a triangle mesh can be easily applied to a *tetrahedral* complex. This data-type might occur as an intermediate step to produce a surface mesh (for example in Dalaunay triangulation based mesh generation approaches). The *implicit* variant uses a vector of

vertices and a vector of indices. Each consecutive four points correspond to the four vertices of one tetrahedron.

The *explicit* variant is a vector of tetrahedrons. A single tetrahedron consists of three triangles. The triangle class is the same for the tetrahedron and the *explicit* triangle mesh.

Table 4.3: Comparison table for triangle mesh representation in existing libraries and requirements for BRICS.

Library	Explicit or implicit triangle representation	Triangle mesh representation	Explicit triangle representation (if present)	Streaming support	Transformation to other representations	Robotic library	Notes
VTK	explicit	array of triangles	set of lines	✓	-	-	
Meshlab	both	vector of vertices and facets or vector of triangles	array of 3 vertices	-	-	-	user defined data-types; both representations can be used
CGAL	explicit	container class that holds facets	-	✓	✓	-	heavily templated
gmesh	explicit	vector of triangles	array of 3 pointers to vertices	✓	✓	-	
Qhull	implicit	double linked list of structs for vertices and indices	-	-	-	-	
CoPP & BRICS_MM	explicit	vector of triangles	3 vertices	✓	-	✓	
openrave	both	vector of vertices and indices	3 vertices	✓	-	✓	triangle representation unused
OSG	implicit	vector of vertices and indices	-	-	-	-	
ROS	implicit	array of vertices and indices	-	-	-	✓	
Requirements for BRICS	both	vector of vertices and facets or vector of triangles	array of 3 vertices	✓	-	✓	

4.4 Refactoring and harmonization of common algorithms

In this section common atomic algorithmic components are presented. They are described with UML software component diagrams. As the harmonized and refactored algorithms will be implemented with the C++ programing language, the component interfaces will presented in C++.

For 3D perception and modeling at least the following atomic components can be identified:

- Octree algorithm

- Iterative Closes Point algorithm
- k-Nearest Neighbors search algorithm
- Delaunay triangulation

This list does claim to be complete, but these are the most obvious and common elements as deduced from the *Exploration* phase (cf. Section 3.2). The remainder of this section investigates each of the components and discusses the interfaces.

4.4.1 The Octree component

The Octree algorithm is the de-facto standard to reduce point clouds and it is used for voxel representation or for surface mesh generation approaches. The Octree can be regarded as a common atomic element for 3D perception and modeling applications.

The Octree component has two different roles: first as *reduction filter* and second as structured *partition* of the space into cubes. To account for both roles, two separated provided interfaces are offered for each functionality. The first functional interface is the **`IOctreeReductionFilter`**. It provides capabilities to reduce point clouds. The provided method **virtual void** **`reducePointCloud`**(**`PointCloud3D`**$*$ **`originalPointCloud`**, **`PointCloud3D`**$*$ **`resultPointCloud`**)$= 0$; needs a point cloud as input and creates a new point cloud with the reduced size. The other functional interface **`IOctreePartition`** provides functionality to partition a point cloud in a set of smaller point clouds. The function **virtual void** **`partitionPointCloud`**(**`PointCloud3D`**$*$ **`pointCloud`**, **`std`**::**`vector`**$<$**`PointCloud3D`**$>*$ **`pointCloudCells`**)$= 0$; accepts a point cloud as input parameter and creates a new vector of point clouds that represents the cells with the points.

To decouple the configurable parameters, a third provided interface is offered. The parameter that needs to be defined is the *voxel size*. Therefore the interface **`IOctreeSetup`** has a getter and a setter method to manipulate the parameter **`voxelSize`**. The Octree component does not depend on an other modules and thus has no required interfaces. Figure 4.1 shows the according component diagram.

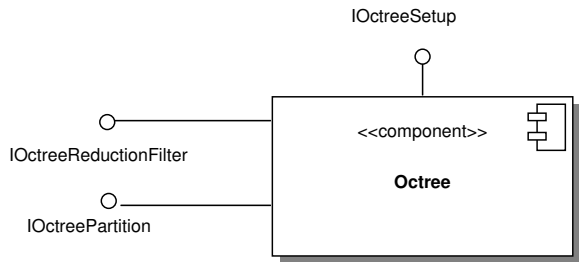


Figure 4.1: UML component diagram of the Octree algorithm.

4.4.2 The Iterative Closest Point component

The most common way the to register multiple point clouds into one common coordinate frame, is to use the Iterative Closest Point (ICP) algorithm.

Reviewing Algorithm 2.1, the ICP has two major sub-elements: a step that establishes point correspondences and a step that can estimate rigid transformations. Both steps can be solved by various approaches. To be able to exchange atomic parts, both steps will be encapsulated as subcomponents. These subcomponents will be addressed by the required interfaces of the ICP component: **`IPointCorrespondence`** and **`IRigidTransformationEstimation`**.

This ICP component offers the simple matching functionality in the provided interface: **`IIterativeClosestPoint`**. This minimal interface needs to accept two point clouds: *model* and *data*, and calculates the translation and rotation that needs to be applied to the data so that it is aligned to the model. The rotation and translation can be summarized in a homogeneous transformation matrix, and is accessible with the **`resultTransformation`** output parameter: **virtual void** **`match`**(**`PointCloud3D`**∗ **`model`**, **`PointCloud3D`**∗ **`data`**, **`IHomogeneousMatrix44`** ∗ **`resultTransformation`**)= 0;. Details of the homogeneous transformation of point and point clouds **`IHomogeneousMatrix44`**, can be found in Section 5.3.1.

Beside the above explained simple interface, a second interface **`IIterativeClosestPointDetailed`** is offered that reveals more internal details to the user of this component. A potential user of this component might what to define new termination criteria or a system scheduler has the responsibility to invoke this component iteratively according to a scheduling policy. The interface has getter and setter methods for the *data* and the *model* point cloud and a method **virtual double** **`performNextIteration`**()= 0; that invokes only one iteration of the ICP and returns the error according to Equation 2.2. Functions like **`getLastEstimatedTransformation`** and **`getAccumulatedTransfomation`** allow to get intermediate and accumulated results of the transformation. This is a *stateful* interface that means the results rely on previous states for example invocations of **`setData`**, **`setModel`** or **`performNextIteration`**. This implies a *contract* on the interface: to correctly use this interface first set *data* and *model*, then invoke **`performNextIteration`**, as often as desired.

Note that the **`IIterativeClosestPoint`** is a *stateless* interface, and the behavior is always the same (even if multiple threads invoke the matching functionality), while the behavior of **`IIterativeClosestPointDetailed`** depends on the history of preceding events [12]. Both interface types are clearly separated.

The interface that fulfills the *configuration* role is the **`IIterativeClosestPointSetup`**. It allows to set and get the convergence threshold, the maximum number of iterations and it allows to configure the required subcomponents. The subcomponents are further describes in the follow-

ing sections. All interfaces for the ICP components are also depicted in an UML class diagram, see Figure A.31. Figure 4.2 presents the component diagram for the ICP.

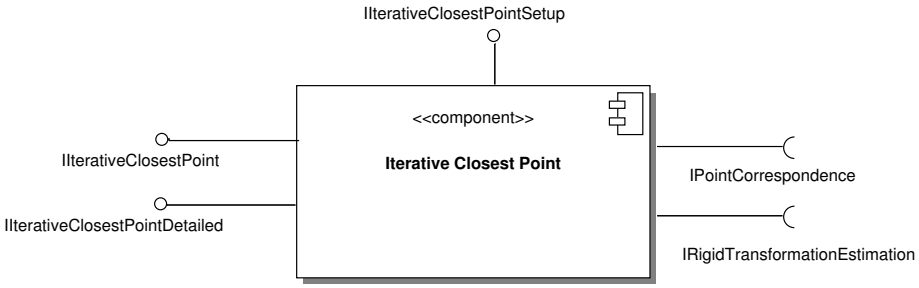


Figure 4.2: UML component diagram of the Iterative Closest Point algorithm.

The Point Correspondence subcomponent

The component for establishing the point-to-point correspondences, needs two point clouds as input data. And returns a list of corresponding points. To represent the corresponding points a new class is in introduced: **`CorrespondencePoint3DPair`**. It essentially consists of two Cartesian points that model the correspondence.

The provided interface **`IPointCorrespondence`** has just one method that allows to calculate the point-to-point correspondence **virtual void** **`createNearestNeighborCorrespondence`** (**`PointCloud3D`**$*$ **`pointCloud1`**, **`PointCloud3D`**$*$ **`pointCloud2`**,**`std`**::**`vector`**$<$ **`CorrespondencePoint3DPair`**$>*$ **`resultPointPairs`**)= 0;. The component does not need to be configured, as it has no parameters, nor it needs a required interface. As internal realization the k-Nearest Neighbor search component, which will be presented later, could be used, but this is completely left to the implementation. Figure 4.3 shows the according UML component diagram.

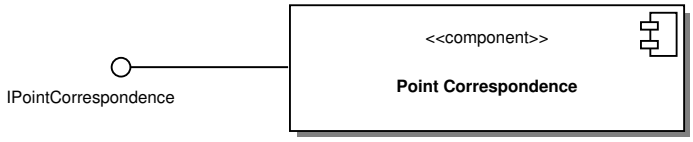


Figure 4.3: UML component diagram of the Point Correspondence component.

The Rigid Transformation Estimation subcomponent.

The second required interface for the ICP algorithm, is the Rigid Transformation Estimation component. It provides one interface: **`IRigidTransformationEstimation`**. This interface has a list of point correspondences as input and a homogeneous transformation as output parameter, to store the resulting transformation. The return value is the resulting error according to Equation 2.2: **virtual double** **`estimateTransformation`**(**`vector`**$<$**`CorrespondencePoint3DPair`** $>*$ **`pointPairs`**, **`IHomogeneousMatrix44`**$*$ **`resultTransformation`**)= 0;

The component has no required interfaces, nor it has parameters that need to be configured. Figure 4.4 shows the component diagram.

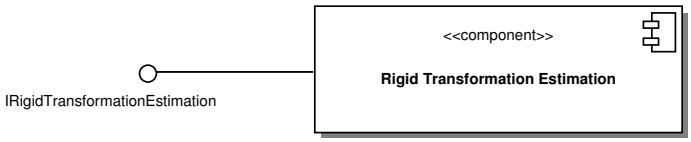


Figure 4.4: UML component diagram of the Rigid Transformation Estimation component.

4.4.3 The k-Nearest Neighbor search component

The k-Nearest Neighbor search component is a general algorithm to compute the k nearest neighbors to a point (or a vector of values in general). Nearest Neighbor search operations are for example used by *registration* or *normal estimation filtering* algorithms, thus it can be seen as a common atomic element for 3D perception and modeling.

The component will account for two different roles, the first is a general user that might want to use the component in a completely other context than robotics, and the second users uses Cartesian points with dimension three. The more general interface is called **`INearestNeighbor`** and allows to set a multidimensional vector *data*. **`findNearestNeigbor`** uses a vector as query, as well as k, and it will return a vector of indices to the k nearest neighbors (cf. Figure A.34).

The interface **`INearestPoint3DNeighbor`** is specific to the 3D perception and modeling domain, as it uses Cartesian points. Instead of a multidimensional data vector, the data is defined by a point cloud. The query is a **`Point3D`**, rather than a vector: **virtual** **`vector`**<**int**> **`findNearestNeigbor`**(**`Point3D`**∗ **`query`**, **int** **`k`**=1)= 0; The default value for k, for this interface and the above one, is 1.

Both interfaces are *stateful*, as most implementations first crate an appropriate search structure, like for example a search tree. Search queries are then accelerated by using that structure. Whenever in the k-Nearest Neighbor interfaces the *data* is set, these search structures are created. As a consequence the user has to follow the *contract* that first the *data* is set and afterwards the queries are invoked. In addition to that, the component implementation has to check erroneous input like search queries that have a mismatching dimension with the *data*.

The configuration interface **`INearestNeighborSetup`** allows to get the **`dimension`** and to set and get an optional parameter for the maximal allowed distance, to regard an element as neighbor. This component has no required interfaces and is presented as UML component diagram in Figure 4.5.

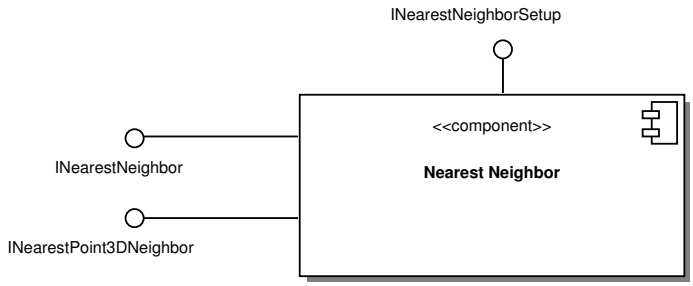


Figure 4.5: UML component diagram for the k-Nearest Neighbor search component

4.4.4 The Delaunay Triangulation component

The Delaunay Triangulation algorithm is commonly used as atomic element among the mesh generation algorithms, like for example the algorithms of the *CRUST* and *COCONE* family or the α-shapes method, depend on this triangulation (cf. Section 3.2).

The primary role of the Delaunay Triangulation component is to create a triangulation from a point cloud. The result of a 3D triangulation is a set of tetrahedrons, which is accessible as output parameter: **virtual void** `triangulate`(`PointCloud3D`∗ `pointCloud`, `ITetrahedronSet`∗ `tetrahedrons`)= 0; The representation of a tetrahedron set is discussed in Section 4.3.3. Beside the 3D triangulation an application might only need 2D triangulations embedded into a 3D space. That means one axis is ignored and the elevation to this axis is flattened. This could be the case if the triangulation is directly applied to a depth image, whereas the depth axis is ignored. In this case the 2D triangulation would be faster, because the problem space is reduced by one dimension. The following method offers this capabilities. Input is a point cloud and a triangle mesh is the output parameter. The parameter `ignore` allows to specify, which axis should be ignored: **virtual void** `triangulate`(`PointCloud3D`∗ `pointCloud`, `ITriangleMesh`∗ `mesh`, `axis ignore` = `z`)= 0;

All triangulations obey the *Delaunay property* (cf. Section 2.3.5), and do not need any further parameters. That is why there is no configuration interface for this component. It also has no required interfaces. Figure 4.6 illustrates the UML component diagram fro the Delaunay Triangulation.

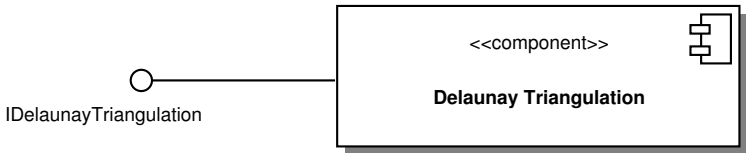


Figure 4.6: UML component diagram for the Delaunay Triangulation component

Chapter 5

IMPLEMENTATION

This section presents details of the *Refactoring* (cf. Section 4.1.3) and *Integration* (cf. Section 4.1.4) phases. Implementation of the requirements for harmonized data-types are explained, then the realization of the components for common algorithms are presented. The software is integrated into the *BRICS_3D* library. Note that the development of the BRICS_3D library continues beyond the scope of this master thesis.

5.1 Choice of programming language and tools

The implementation is done in the *C++* programming language. It is a multi-paradigm language that supports object-oriented programming, but it does not enforce it. This is by far the predominant language in the field of robotics, computational geometry and 3D perception and modeling in large. Most libraries in these domains are already written in *C++* or *C*. See also Section 3.3.

The development is accompanied by a couple of tools. As *Integrated Development Environment (IDE)* the *Eclipse (Galileo)* platform is used, in combination with the *CDT* plug-in for *C* and *C++* developments. The source code is documented with *Doxygen*, to automatically generate manual pages. The *Eclox* plug-in for Eclipse helps to create Doxygen conform source code comments. To foster operating system independence, the BRICS_3D library is compiled with the *CMake* build system. A plug-in called *cmake editor* for Eclipse enables syntax-highlighting and code-completion capabilities for the CMake configuration files. Further tools are the *Subversion (SVN)* code revision system in combination with a Subversion plug-in for Eclipse that directly allows to update and commit source code.

For profiling, in terms of how many time has been spend in which function, the *OProfile* is used. To check memory consumption and memory leaks the *Valgrind* suite is utilized. Both tools have a good integration into the Eclipse platform with the *Linux Tools* plug-in. As unit testing framework the *CppUnit* library is incorporated. Success and failure of single unit tests can be visualized with the *ECUT* plug-in for Eclipse. The benefit of unit tests is tremendous during the refactoring phases. To name an example, the *decorator pattern* for the Cartesian points was applied after an initial and simple version of the point representation. As the point data-type is used by nearly all other algorithms it is crucial to be able to check if the desired behavior does not changes, while refactoring the code.

Bouml is a powerful tool to create UML diagrams. Among others, it supports class and component diagrams. It has the ability to generate source code skeletons for C/C++. It even has capabilities to reverse engineer source code. That means UML class diagrams are generated from existing code. Manual adjustments still have to be done, because Bouml has problems with C/C++ macro expansions or some dependencies are not resolved when C++ templates are used. All UML diagrams presented in this work, are created with Bouml.

A note on the development process: the development roughly follows agile software development principles [119], with tools that allow test-driven development, a code revision system that encourages modification of existing code, collective code owner-ship in the SVN source code repository, coding conventions and appropriate documentation in the source code. A continuous integrations system that automatically compiles the software on different operating systems with different compilers, is planned for the near future.

5.2 Implementation Overview

The implementation aims towards an *example chain* of processing stages that can produce a triangle mesh from a point cloud. As backbone for 3D perception and modeling applications, the harmonized data-types Cartesian point, Cartesion point cloud and the triangle mesh are implemented. The *depth perception* stage ins realized by functionality to load data sets, which are stored in depth images, simple text files or a file format that is used by IPA's Care-O-bot[61] platform. The Octree for the *filtering* stage, k-Nearest Neighbor search, the ICP algorithm as *registration* method and a 2D Delaunay triangulation for *mesh generation* are implemented. Point clouds and triangle meshes can be *visualized*. Segmentation algorithms are not implemented. The red border in Figure 5.1 contains which parts are realized, with respect to the processing stages.

[61] `http://www.care-o-bot-research.org/`

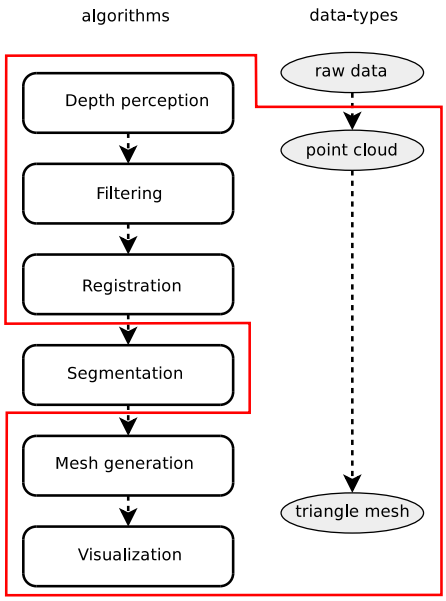


Figure 5.1: Overview of implemented data-types and algorithms

5.3 Common data-types

5.3.1 Cartesian point representation

The Cartesian point representation is implemented in the class **`Point3D`** (cf. Figure A.26). As proposed in Section 4.3.1 the point representation has a simple x, y, z representation. To satisfy the requirements, to be able to choose the data-type at compile time, the Coordinate data-type can be changed by adjusting a typedef for the coordinate type: **typedef double** **`Coordinate`**;

The coordinate values can be easily accessed and with the steaming operator **operator**<< and **operator**>>. Printing a pint to the standard output is convenient, as it means just invoking: **`std`**::**`cout`** << **`examplePoint`**;.

The basic matrix functionality is implemented with operators. The operators **operator**+ and **operator**− allow to add and subtract two points, while the **operator**∗ enables multiplication with a scalar value. The homogeneous transformation is an important function for the Cartesian point representation and is implemented in the following function: **virtual void** **`homogeneousTransformation`**(**`IHomogeneousMatrix44`** ∗**`transformation`**);

The **`IHomogeneousMatrix44`** class (cf. Figure A.27) is an abstract interface to a homogeneous transformation matrix. This abstract class has essentially one function **`getRawData`** that returns a pointer to a data array that stores the values of the transformation matrix. This array stores the values in column-row order[62] and has a fixed of size 16. This is the most general and

[62]The first four entries in the array belong to the first matrix column, the next four elements to the second column, and so on.

simplest form to represent a matrix. The interface also has functions to multiply matrices with each other, to print the values with the streaming operator **operator**<< or assign new values with the **operator**= function.

The interface is implemented in the **HomogeneousMatrix44** class (cf. Figure A.27). It uses the *Eigen* library to implement matrix multiplications and convenient set-up in the class constructor. The transformation function in **Point3D** depends only on the abstract interface, rather on the implementation of the homogeneous matrix. This conforms to the *Dependency Inversion Principle* [119]. The goal is that no harmonized data-type relies on any external library.

A point might have additional information like color or a normal vector. To allow good extendability, the *decorator pattern* [9] is applied. The **Point3DDecorator** has the same interface as the **Point3D** as it inherits from it. Additionally it holds a reference to an instance of a **Point3D**. Whenever a function of the decorator is invoked it is internally forwarded to this point reference.

An example realization of a point extension, is the **ColoredPoint3D** class that adds new variables for the additional color information. It inherits form the **Point3DDecorator**, thus it can wraps a point into a layer or skin that appears to the outer world as a regular **Point3D**, but internally it has additional information that is acessible with the **ColoredPoint3D** interface.

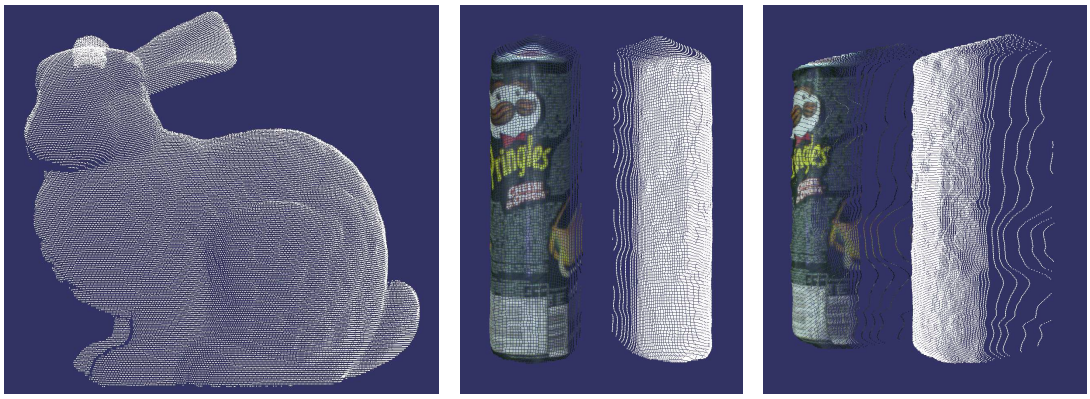
It is possible to perform multiple decorations on a point. In this case, it can be seen as some kind of onion that has different layers - each adds a new portion of information. Queries to the outer layer **Point3D** are forwarded to the core in cascaded way.

5.3.2 Cartesian point cloud representation

The point cloud is a collection of Cartesian points. As concluded in the requirements for a harmonized point cloud representation (cf. Section 4.3.2), it consist of a vector of points: **std**:: **vector**<**Point3D**>∗ **pointCloud**;. The reference to the vector can be accessed via **getPointCloud**. The vector can contain either normal points of type **Point3D** or decorated points. Actually for the point cloud there is no difference.

The point cloud class **PointCloud3D** (cf. Figure A.28) implements abilities to apply a homogeneous transformation to all points. The **homogeneousTransformation** method forwards the **IHomogeneousMatrix44** to every point in the vector. A similar behavior have the streaming methods **operator**<< and **operator**>>, as data is forwarded from or to the points.

The point cloud offers simple capabilities to make the data persistent, as it is able to load and store from text files. Furthermore data can be saved to the *.ply* format that is supported by many 3D modeling and visualization tools. Figure 5.2 demonstrates some example point clouds.



(a) (b) (c)

Figure 5.2: Examples of point clouds. (a) shows the *Stanford* bunny as point cloud (b) shows a *Pringles* can. The left point cloud has the decorated color points and the right cloud has no decoration. (c) depicts the same point cloud, but slightly rotated.

5.3.3 Triangle mesh representation

Triangle meshes are often represented in an *implicit* or an *explicit* manner. As explained in Section 4.3.3, the harmonized representations should support both versions, to grand flexibility.

The **`TriangleMeshImplicit`** class (cf. Figure A.29) implements the *implicit* version. It has a vector **`vertices`** that holds the points of type **`Point3D`**. In combination with the **`indices`** vector triangles can be represented. The *explicit* mesh representation **`TriangleMeshExplicit`** has a vector of triangles (cf. Figure A.29). Each triangle is modeled by the class **`Triangle`**. It has an array of dimension 3 to store the vertices belonging to a triangle (cf. Figure 2.2). Both mesh implementations allow access to their individual vectors via getter and setter methods.

To make both variants exchangeable, they share the same interface **`ITriangleMesh`** (cf. Figure A.29). It gives a common access to a the vertices of a triangle via: **virtual** **`Point3D`**∗ **`getTriangleVertex`**(**int** **`triangleIndex`**, **int** **`vertexIndex`**)= 0;. Triangles can be added and removed with **`addTriangle`** and **`removeTriangle`**. The adequate maintenance of the underling structures has to be handled different by both mesh implementations. The interface offers functionality to apply homogeneous transformation matrices. To transform a mesh, the matrix is propagated to the stored **`Point3D`** objects.

Streaming capabilities are available by the **operator**<< and **operator**>> methods, similar to a point cloud. A subtle issue arises with the usage of such operators in an abstract interface, within C++. The operators need to have the modifier **friend** to be used easily. Otherwise a stream could only be send from one triangle mesh to another triangle mesh. It would not be possible to stream to a file or the standard output. Unfortunately an abstract function cannot have the modifier **friend**. To resolve the dilemma, the non-abstract steaming operators forward the steams to the abstract

read and **write** functions. An illustration of an example triangle mesh can be seen in Figure 5.3.

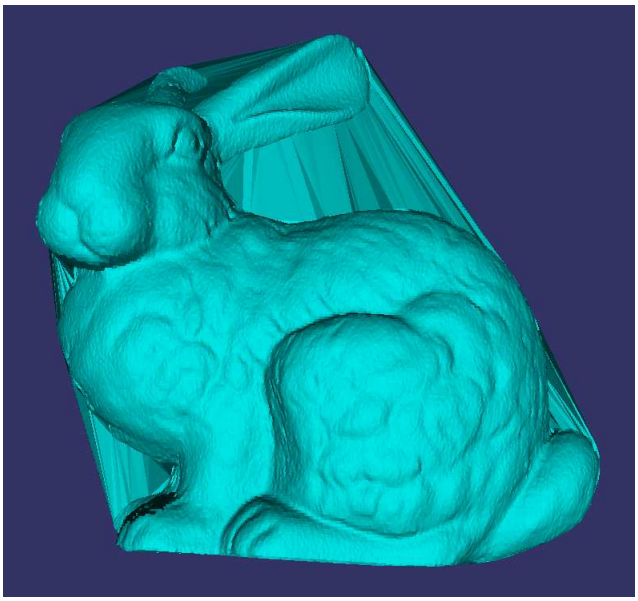


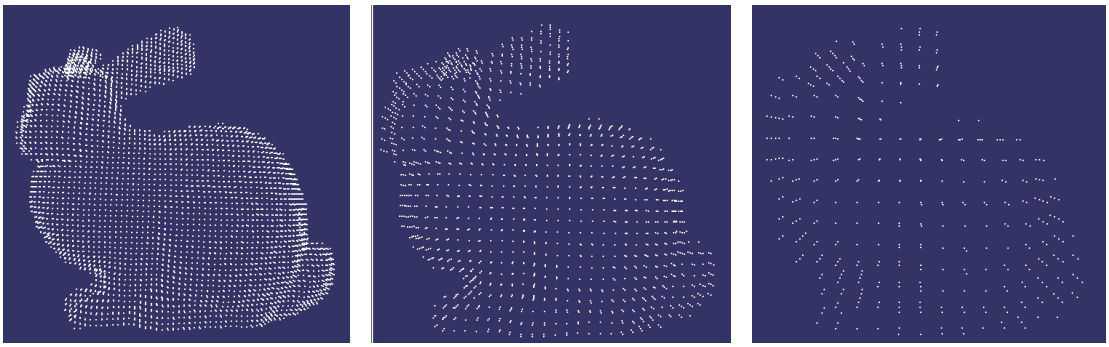
Figure 5.3: Example of a triangle mesh. The image shows the *Stanford* bunny with a triangle mesh of its surface.

5.4 Common algorithms

5.4.1 The Octree component

The Octree component is realized in a single class **Octree** (cf. Figure A.30). It implements all the provided interfaces. The functionality to create an Octrees is taken from the 6DSLAM library (cf. Section 3.3). The **Octree** class can be seen as a wrapper to the 6DSLAM library.

The Figure 5.4 gives an example of the Octree reduction filter **IOctreeReductionFilter**, which is applied to the Stanford bunny data set. The unfiltered version, as already depicted in Figure 5.2(a) has 40256 points is the point cloud. The Octree algorithm with a *voxel size* of 0.002 reduces this point cloud to 5048. A *voxel size* of 0.004 creates 1444 points and a *voxel size* of 0.005 further increases the size to 418.



(a) (b) (c)

Figure 5.4: Example of Octree reduction. (a) shows the *Stanford* bunny as reduced point cloud with *voxel size* = 0.002. (b) shows the data with a *voxel size* = 0.004. (c) presents further reduction with *voxel size* = 0.005.

5.4.2 The Iterative Closest Point component

The **`IterativeClosestPoint`** (cf. A.31) class serves as a generic implementation of the Iterative Closest Point algorithm. It follows the *strategy* software design pattern [9], with the slight modification that *context* and *strategy* are implemented in the same class. **`IterativeClosestPoint`** holds two references **`assigner`** and **`estimator`** to the abstract interfaces of the subcomponents **`IPointCorrespondence`** and **`IRigidTransformationEstimation`**. The references are trigged during the iteration of the algorithm (cf. Algorithm 2.1). The concrete instances are defined beyond the scope of the **`IterativeClosestPoint`** class and are configurable through the **`IIterativeClosestPointSetup`** interface. That means that the actual point correspondence and the rigid transformation estimation algorithms are exchangeable during runtime.

C++ has no possibility to encode a required component interface. Though the fact that concrete implementations for the above mentioned the sub-algorithms are needed, shall reflect the required interfaces of ICP component here. This component also has to take into account that the **`IIterativeClosestPointDetailed`** is a *stateful* interface, while the other one is *stateless*. The *statful* interface stores the intermediate steps in member variables for the **`model`** and **`data`** point clouds, and the transformation matrices. The *stateless* interface uses its own version of the **`model`** and **`data`** variables and thus hides their member pendants. This prevents data corruption if both interfaces are called in an intermixed manner.

The Point Correspondence component

The Point Correspondence component creates point-to-point correspondences between two point clouds, by computing the Nearest Neighbor from each point from the first point cloud to the points of the second point cloud. Two implementations of this component are available (cf.

Figure A.32). The first bases on the optimized k-d tree for dimension 3. It is implemented in the 6DSLAM library and the class **`PointCorrespondenceKDTree`** creates a wrapper to comply the **`IPointCorrespondence`** interface.

The second implementation **`PointCorrespondenceGenericNN`** uses the general k-Nearest Neighbor component (cf. 5.4.3) with dimension = 3 and neighborhood $k = 1$.

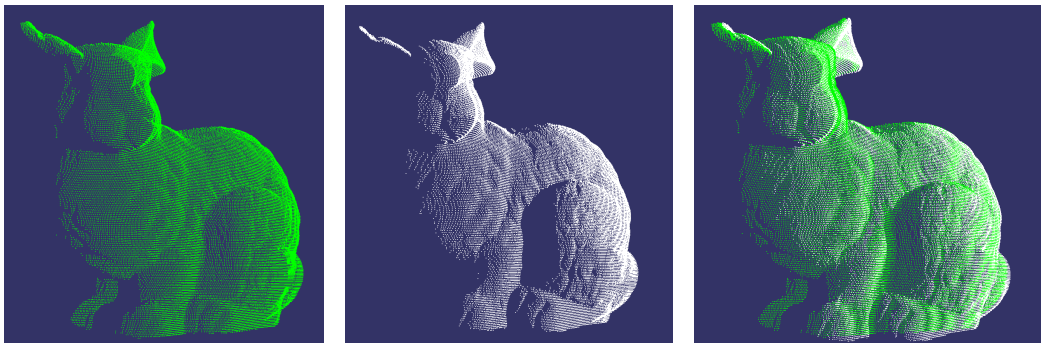
The Rigid Transformation Estimation component

An implementation of the for Rigid Transformation Estimation component only has to satisfy the **`IRigidTransformationEstimation`** interface. Five different algorithms to solve the estimation are available (cf. Figure A.33). They are taken from 6DSLAM library and adapted to the component interface.

The **`RigidTransformationEstimationSVD`** implements the transformation with the *Singular Value Decomposition (SVD)* approach, **`RigidTransformationEstimationQUAT`** uses the *quaternion* based method, **`RigidTransformationEstimationORTHO`** exploits *orthogonal* properties in combination with the *eigensystem*, **`RigidTransformationEstimationHELIX`** implements the *helical motion* estimation and **`RigidTransformationEstimationAPX`** realizes the *linear approximation.*

To cope with all possible variants of Point Correspondence and Rigid Transformation Estimation, a factory class **`IterativeClosestPointFactory`** can assemble the subcomponents for the ICP component. It accepts a XML configuration file, parses it and creates the according instances.

Figure 5.5 gives an impression of the ICP algorithm. Two different, but overlapping data sets, here depicted with green points in Figure 5.5(a) and with white points in Figure 5.5(b), are registered into one consistent coordinate frame, as seen in Figure 5.5(c).



(a) (b) (c)

Figure 5.5: Example of ICP registration. (a) shows the first point data set (b) shows the second data set and (c) presents the resulting point cloud by applying the ICP to (a) and (b).

5.4.3 The k-Nearest Neighbor search component

The k-Nearest Neighbor search component can use three different implementations, as depicted in the UML class diagram in Figure A.34.

`NearestNeighborANN` realizes the search functionality with the *ANN* library. The algorithms in the *FLANN* library comply to the component interfaces with the **`NearestNeighborFLANN`** class. The third implementation **`NearestNeighborSTANN`** uses functionality from the *STANN* library. Further information to the libraries can be found in Section 3.3.

The k-d tree implementation, as it is used in the Point Correspondence implementation can not be reused, as it is to restricted with dimension 3 and neighborhood $k = 1$. All three implementations automatically deduce the dimension from the input data. To prevent undefined behavior of the component, an **`assert`** statement checks if a query has the correct dimensionality, with respect to the data.

5.4.4 The Delaunay triangulation component

The Delaunay Triangulation component is partially implemented with the Delaunay functionality from the *OpenSceneGraph (OSG)* library, with the calss **`DelaunayTriangulationOSG`** (cf. Figure A.35). OSG only supports 2D tringulations of 3D dimensional points, thus the 2D **`triangulate`** method wraps the OSG function. Beside this, the data-type **`ITetrahedronSet`** is also left for future implementation.

The previously mentioned triangle mesh in Figure 5.3, was generated by the implemented triangulation.

5.5 Framework integration

Further source code has been developed to embed the components into the BRICS_3D library. Some classes are presented here.

The class **`DepthImageLoader`** can load *depth images*. Afterwards, the depth images can can be forwarded to the **`DepthImageToPointCloudTransformation`** class, to generate point clouds. The transformation has a threshold to slice the background off, if needed. That means, all pixels that are further away from the perception device than the threshold, are discarded.

The **`IpaDatasetLoader`** can load the fused data sets of range and color images recorded on a Care-O-bot platform. The result is a point cloud with decorated color points. The *Pringles* can, previously seen in Figures 5.2(b) and 5.2(c) is an example of a successfully loaded data set.

The *visualization* capabilities are realized with the OSG library. Point clouds and triangle meshes can be displayed. In OSG it is benifitial for huge point clouds, to partition them into bunches of approximately no more than $10,000$ points, per *Geode*. Otherwise the performance drops significantly. The partition into multiple bunches accounts for the parallel architecture of the graphics adapter hardware.

`Benchmark` is a simple benchmark suite that allows to store benchmarking results in an automated way. Instead of printing (intermediate) results to the standard output, they are sent to the benchmark object. Each benchmark object needs to be initialized with a name that is used to create a logfile to store the results. A logfile is stored in a folder named by the current time-stamp in a YYYY-MM-DD_HH-MM-SS fashion, for example ”2010-02-15_17-02-22”.

Every new benchmark instance after the first one also stores its logfile in the same time-stamp folder. This means one time stamp represents one run of a process, from its creation to its termination. The intention is to repeat a benchmark by relaunching its process. That allows a benchmark to be scheduled by the operating system or a shell script.

Chapter 6

EXPERIMENTAL EVALUATION

Algorithms of the 3D perception and modeling domain have been refactored and harmonized, to make them easier to benchmark on a component level. This Chapter presents an initial set of benchmarks of the atomic components to judge, which algorithms are *best practice* for 3D perception and modeling. The Chapter will start with a description of the used test-bed, will discuss the used metrics and will present a number of benchmarks, with focus on the registration process.

6.1 Evaluation environment

The benchmarks are performed on an off-the-shelf laptop with 2GHz dual core processor, with 3GB memory and a Nvidia graphics adapter. The operating system is an Ubuntu 9.10 with Kernel version 2.6.31-20. The source code is compiled with the gcc compiler version 4.4.1. Compiler options are set to debug, that means no optimizations are activated. The experiments have been performed with source code revision 554 of the subversion repository.

The benchmarks are conducted with recorded data sets. The used data set is the *Stanford Bunny*. All benchmarks are performed with the help of the `Benchmark` class to allow systematic documentation. All experiments are performed multiple times and results show mean and standard deviation σ.

6.2 Performance metrics

The set of benchmarks measure different properties of the refactored and harmonized algorithms. The metrics are *execution time*, *memory consumption* and *error* values, if suitable.

- **Processing time:** This measures the required processing time. It is deduced from the difference of one time stamp for and one after invocation of an algorithm. The measured unit is ms. This metric falls into the category *costs* as discussed in Section 3.1.
- **Memory consumption**: The memory consumption is measured with the *Valgrind* profiling tool. Like execution time, it measures the *costs* of an algorithm. Results are presented with MB as measurement unit.
- **Error**: The error values that can be measured depend on the algorithm. For example the Rigid Transformation Estimation returns an error value. This gives hint about the *quality*

of the output of an algorithm. This metric belongs the the category *utility* as presented in Section 3.1.

Depending on the algorithm, further metrics are applied, for example how many iterations have been performed within the ICP.

6.3 Performance of Cartesian point data-type

The purpose of this benchmark is to measure the influence of the coordinate type in the Cartesian point representation. Two metrics are applied: first, the processing time while performing a homogeneous matrix transformation is measured and second, the memory consumption is measured.

The benchmark is performed as follows: a point cloud is crated with $10,000$ points. Then the matrix transformation is applied. Iteratively $10,000$ points are added and the transformation matrix is applied again. To ensure repeatability of this experiment the random generator always has the same initial *seed* of 0. The whole experiment has been repeated 10 times with either the **double** or the **float** representation.

Table 6.1 presents the results for the measured quantity *processing time*. The same data is plotted in Figure 6.1. The outcome is that there is no significant difference in the processing time, on the used test-bed.

The memory profiles, crated with *Valgrind* tool are shown in Figure 6.2. The **float** based representation consumes 50457033 Bytes as peak, while the **double** based representation consumes 88205769 Bytes. The latter one requires roughly two times more memory than the other one. This result does not surprise as a **float** variable needs with its 32bit representation only half the memory than the 64bit **double** representation.

If enough memory is present, the **double** representation is favored on the used test-bed, as it has no processing time drawbacks, but offers a higher precision of results.

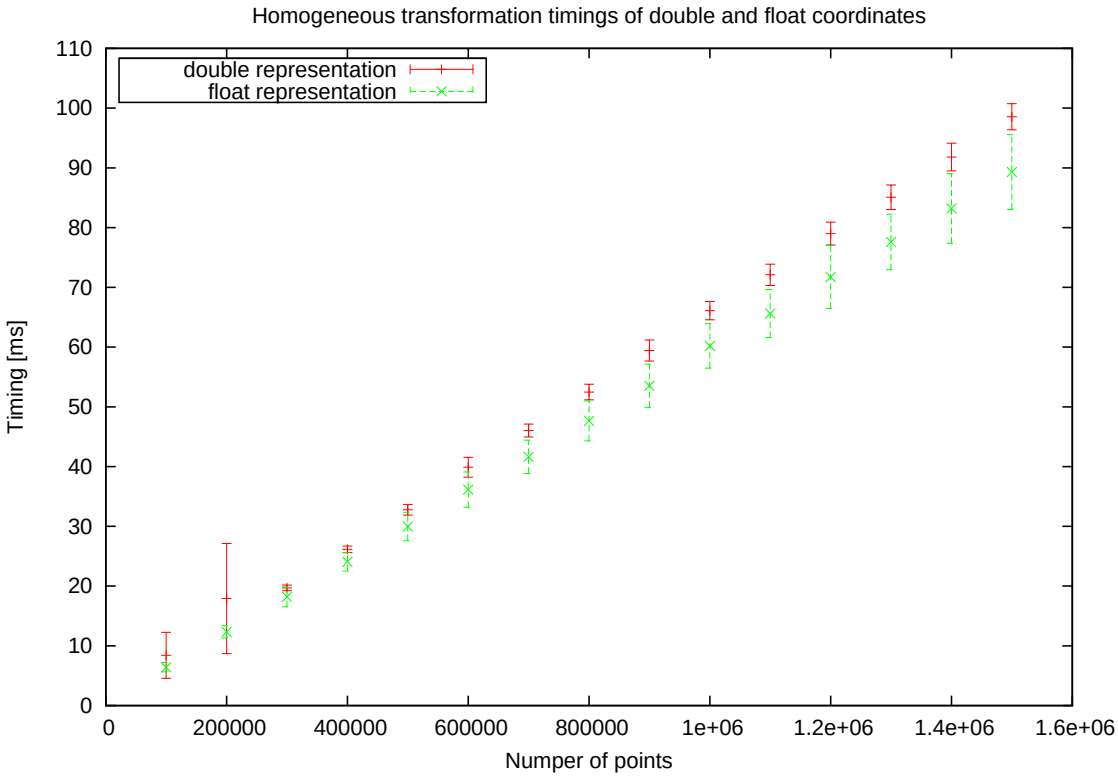


Figure 6.1: Benchmark of influence of coordinate type in Cartesian point representation.

Table 6.1: Benchmark of influence of coordinate type in Cartesian point representation.

Number of points in thousand	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500
double	8.41	17.92	19.7	26.16	32.77	39.8	46.03	52.48	59.43	66.09	72.1	79	85.08	91.82	98.56
float	6.42	12.35	18.2	24.06	29.96	36.15	41.63	47.64	53.53	60.2	65.62	71.72	77.6	83.21	89.31
σ double	3.86	9.22	0.47	0.56	0.9	1.66	1.08	1.3	1.76	1.53	1.78	1.91	2.05	2.32	2.19
σ float	0.77	1.02	1.67	1.54	2.38	2.96	2.8	3.34	3.63	3.73	4.02	5.26	4.64	5.84	6.25

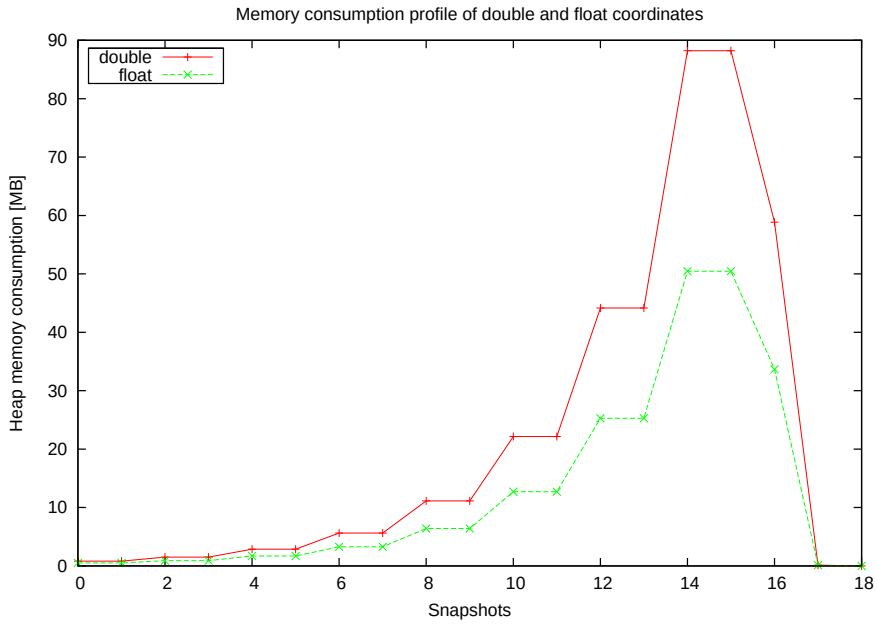


Figure 6.2: Memory profiles for coordinate types in Cartesian point representation

6.4 Performance of Point Correspondence

This benchmark measures the performance of the Point Correspondence component. The *bun000.ply* data set is used to create a point cloud. It has a size of 40256 points. A second point cloud is created by cloning the fist and and then applying a translation vector $(0.1, 0.1, 0.1)$. The processing time to compute the point-to-point correspondences is measured. The correspondences are known: the i-th point of the first cloud belongs to the i-th point in the second cloud. This allows to deduce if a correspondence is correct. The measured quantity is the number of correct assigned points divided by the total number of points in a cloud. This shall reflect the *utility* of the generated output.

All available implementations for the Point Correspondence component are used. The k-d tree and the ANN, FLANN and STANN implementation for the k-Nearest Neighbor search component are benchmarked. All algorithms use their default values. The maximal distance threshold hold is the default value of 50.

The experiment has been performed 10 times and the results are depicted in Figure 6.3. All algorithms achieve the same amount of correct correspondences: 100%. The results for the processing times show differences. ANN and FLANN are the fastest algorithms, closely followed by the k-d tree. The STANN implementation is significant slower. For this test-bed and for this data set, ANN and FLANN can be considered *best practice*.

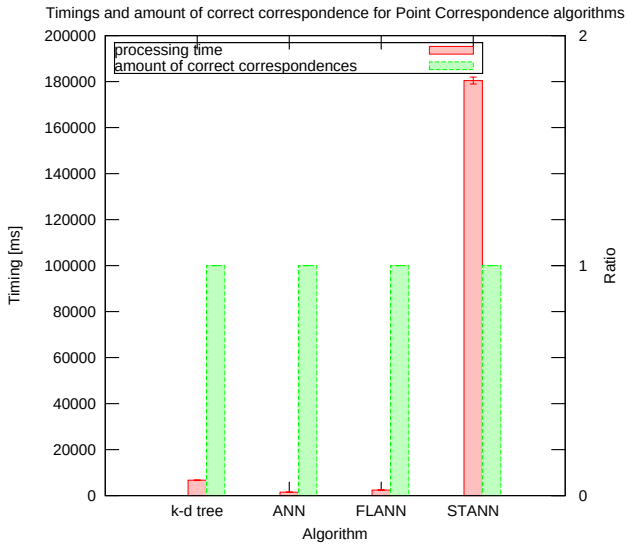


Figure 6.3: Benchmark results for the Point Correspondence algorithms

6.5 Performance of Rigid Transformation Estimation

The purpose of this benchmark is to evaluate the Rigid Transformation Estimation component. The set up is similar to the above benchmark. The *bun000.ply* data set is used to create two point clouds with know displacement to each other. The displacement is defined by the translation vector $(1, 1, 1)$. The measured quantities are the processing time, the resulting root mean square (RMS) error of the point cloud distances and a metric that measures how similar the estimated and the inverted known transformation matrices are. Each matrix value is incorporates into an RMS error value.

The implementations for Rigid Transformation Estimation comprise the Singular Value Decomposition SVD, the quaternion based approach QUAT, the helical motion (HELIX) and the linear approximation approach APX. The implementation for the orthogonal properties ORTHO was not used, because of an unsolved failure during execution.

The results, as presented in Figure 6.4, reveal that the point cloud distance errors and the matrix errors are identical. There are differences in the processing time: QUAT, HELIX and APX are roughly on the same level whereas SVD is slower. In this case, for this data set and on this test-bed the QUAT, HELIX and APX can be considered *best practice* as they demonstrate equal performance.

6.6 Performance of Iterative Closest Point

To benchmark the Iterative Closest Point algorithm, all possible combinations of Point Correspondence and Rigid Transformation Estimation implementations will be compared. As input data the *bun000.ply* and the *bun045.ply* are used. This are exactly the data sets used in Figure 5.5.

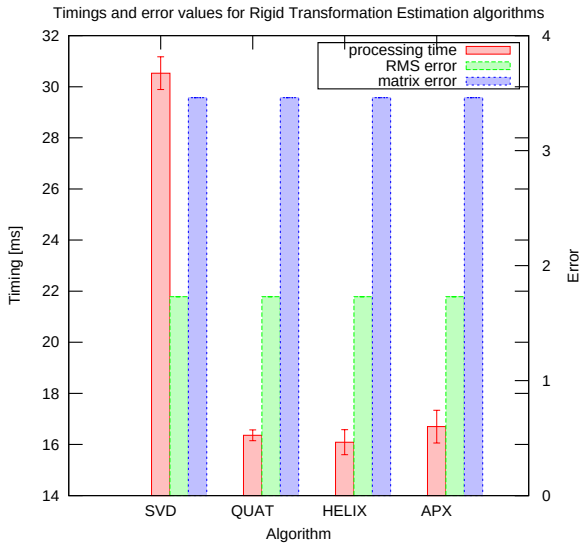


Figure 6.4: Benchmark results for the Rigid Transformation Estimation algorithms

Three metrics are used in the benchmarks. The precessing time to align the two point clouds is measured, as well as the needed number of iterations and the resulting RMS error after the final iteration.

The parameters of the ICP are set as follows: convergence threshold is 0.00001, the maximal point-to-point distance is 50 and the maximal amount of iterations is set to 100. Every matching process was repeated 10 times.

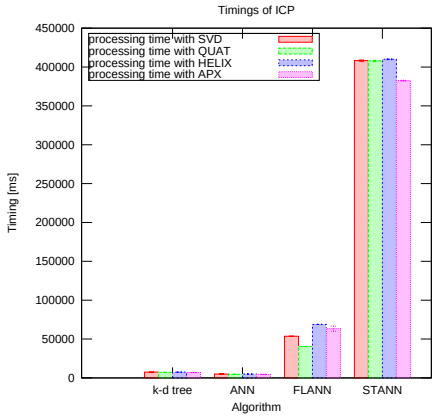
The results for the processing time are illustrated in Figure 6.5(a). Independent of the used rigid transformation estimation algorithms, the ANN point-to-point correspondence implementation outperforms the other algorithms. The STANN implementation is by far the slowest approach. This benchmarks also confirms that the point-to-point correspondence problem is the most computational part of the ICP, as the transformation estimation has only a minor influence on the timing behavior.

The amount of required iterations is roughly the same for all algorithms and is approximatively 20 iterations (cf. Figure 6.5(b)). The only exception is the FLANN approach that needs in combination with the QUAT estimation the least iterations (16) but with the HELIX it needs the most. In combination with the APX algorithm the FLANN does not work deterministically, as the numbers of required iterations is not constant in the benchmark.

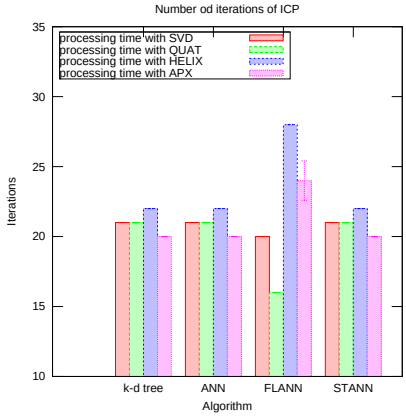
Except for the FLANN approach all point correspondence algorithms produce roughly the same resulting RMS error (cf. Figure 6.5(c)).

For the used test-bed and data set the ANN algorithm for establishing the point-to-point

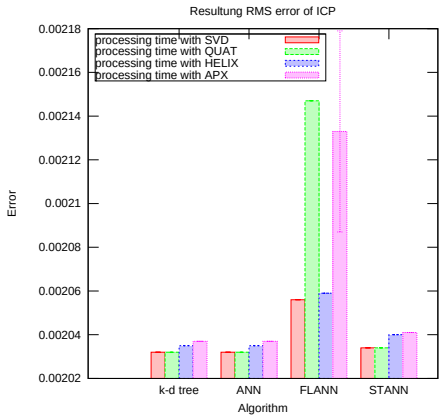
correspondences in combination with SVD or QUAT are *best practice* choices, because it is able to produce the most precise results with the least required processing time.



(a)



(b)



(c)

Figure 6.5: Benchmark results for the Iterative Closest Point algorithm

To conclude this Chapter, it is possible to deduce *best practice* algorithms by benchmarking the refactored components that encapsulate common, atomic algorithms. The benchmark results are only valid for the used test-bed and the used test data. Furthermore, the effects of wrapping and adopting the implementations are neglected. The message is not that a certain algorithm **is** *best practice*, rather than **it is possible** to get access to *best practice* algorithms.

Chapter 7

CONCLUSION

This Chapter summarizes the contributions and results of this work and depicts open issues.

7.1 Summary

This work has applied software engineering aspects, in particular *software components*, to refactor existing algorithms into common atomic elements. These elements can be benchmarked to deduce the *best practice* algorithms for a specific task.

The 3D perception and modeling domain has been structured into the subareas: *depth perception, filtering, registration, segmentation, mesh generation* and *visualization*. The state-of-the-art has been conducted to identify the predominant data-types and algorithms in these categories. The *Octree* algorithm is an atomic component for *filtering* and *mesh generation*, the *Iterative Closest Point (ICP)* algorithm is the predominant *registration* method, *Delaunay triangulation* is commonly seen in surface *mesh generation* approaches and the general k*-Nearest Neighbors* search algorithm is required by many other algorithms.

Existing libraries have been analyzed to find harmonized data-types that are required by the above algorithms. Requirements for harmonized data-types for the Cartesian point, the Cartesian point cloud and the triangle mesh representations have been proposed and implemented.

The identified common algorithms have been encapsulated into software components. Harmonized interfaces for these components have been proposed. The algorithms are implemented by refactoring existing source code of public available libraries.

The software components have been embedded into the BRICS_3D framework. This framework allows to load, process, and visualize data sets and it enables benchmarking of the algorithms.

An initial set of benchmarks demonstrates systematic benchmarking on the software component level of the algorithms. Thus it is possible to deduce a *best practice* algorithm for a specific task.

7.2 Future work

Some **open issues** that remain for future work, are listed as follows:

- More harmonized components and implementation for 3D perception and modeling. This work has refactored only some of the algorithms. The *segmentation* stage is not yet ad-

dressed. Normal estimation, noise reduction, alternative registration methods like NDT or HSM3D and mesh generation with the α-shapes algorithm are promising candidates for future implementation, to name some.

- Further work includes creation of new performance metrics for 3D models. The Metro approach [109] is able to measure how similar different meshes are, and might be used for such metrics.
- Incorporation of uncertainty in the data sets is not yet addressed.
- Modeling of grasps and contacts of objects has been neglected so far.
- Modeling of articulated objects is a matter for future work.
- Appliance of the *scenegraph* concept to robotic world modeling is an open issue. That means representation of the environment in a hierarchical and structured way.
- Developement of simulated depth perception sensors might be a valuable contribution to benchmark 3D perception and modeling algorithms. Experiments could be performed *in the loop* with known ground truth.
- The integration of the refactored algorithms into a real robot platform is highly desirable, to validate the applicability of the algorithms in real world scenarios.

The ability to make *best practice* choices of algorithms for a specific robotic task, in early stages of a robot development process, hopefully makes this process faster and easier.

BIBLIOGRAPHY

[1] BusinessDictionary.com, "best practice definition", http://www.businessdictionary.com/definition/best-practice.html, Accessed: March 09, 2010.

[2] J.C. Harris and J.P. Friedman, *Barron's real estate handbook*, Barrons Educational Series Inc, 2001.

[3] Dictionary.com, ""benchmark" in the american heritage dictionary of the english language, fourth edition", http://dictionary.reference.com/browse/benchmark, Accessed: January 21, 2010.

[4] Roland Siegwart and Illah R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, Bradford Book, 2004.

[5] Stuart Russel and Peter Norvig, *Artificial Intelligence: A Modern Approach*, New Jersey: Pearson Education, Inc, 2nd edition, 2003.

[6] A.A. Alatan, Y. Yemez, U. Gudukbay, X. Zabulis, K. Muller, CE Erdem, C. Weigel, A. Smolic, and A. METU, "Scene representation technologies for 3DTV - a survey", *IEEE transactions on circuits and systems for video technology*, vol. 17, no. 11, pp. 1587–1605, 2007.

[7] B. Siciliano and O. Khatib, *Springer handbook of robotics*, Springer-Verlag New York Inc, 2008.

[8] T.K. Dey, *Curve and surface reconstruction: algorithms with mathematical analysis*, Cambridge Univ Pr, 2007.

[9] G. Erich, H. Richard, J. Ralph, and V. John, *Design patterns: elements of reusable object-oriented software*, Addison Wesley Publishing Company, 1995.

[10] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 1999.

[11] C. Szyperski, J. Bosch, and W. Weck, "Component Oriented Programming", *Lecture Notes in Computer Science*, vol. 1743, pp. 184–184, 1999.

[12] D. Brugali and P. Scandurra, "Component-based Robotic Engineering Part I: Reusable building blocks", *IEEE Robotics and Automation Magazine,*, December 2009.

[13] D. Brugali and E. Prassler, "Software Engineering for Robotics", *IEEE Robotics & Automation Magazine*, vol. 26, no. 3, pp. 9, 2009.

[14] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I", in *IEEE Robotics & Automation Magazine*, 2006, vol. 13.

[15] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (slam): part II", in *IEEE Robotics & Automation Magazine*, 2006, vol. 13.

[16] David G. Lowe, "Distinctive image features from scale-invariant keypoints", in *International Journal of Computer Vision*, November 2004, vol. 60, pp. 91–110.

[17] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool, "Surf: Speeded up robust features", in *Computer Vision ECCV 2006*. 2006, vol. 3951/2006, pp. 404–417, Springer Berlin / Heidelberg.

[18] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse, "Monoslam: Real-time single camera slam", in *Transactions on Pattern Analysis and Machine Intelligence*, June 2007, vol. 29.

[19] R.B. Rusu, Z.C. Marton, N. Blodow, M. Dolha, and M. Beetz, "Towards 3d point cloud based object maps for household environments", *Robotics and Autonomous Systems*, vol. 56, no. 11, pp. 927–941, 2008.

[20] M. Levoy, "The digital michelangelo project", pp. 2–11, 1999.

[21] Edmund Milke and Stafan Christen, "Best practice in mobile manipulation - motion planning and control", Master's thesis, University of Applied Sciences Bonn-Rhein-Sieg, 2009.

[22] G. Hirzinger, T. Bodenmiiller, H. Hirschmuller, R. Liu, W. Sepp, M. Suppa, T. Abmayr, and B. Strackenbrock, "Photo-realistic 3d modelling-from robotics perception towards cultural heritage", *Recording, Modeling and Visualization of Cultural Heritage*, p. 361, 2006.

[23] S. You, J. Hu, U. Neumann, and P. Fox, "Urban site modeling from lidar", *Lecture Notes in Computer Science*, pp. 579–588, 2003.

[24] Q. Pan, G. Reitmayr, and T. Drummond, "ProFORMA: Probabilistic Feature-based On-line Rapid Model Acquisition", in *Proc. 20th British Machine Vision Conference (BMVC)*, London, September 2009.

[25] Andreas Nüchter, *3D Robotic Mapping The Simultaneous Localization and Mapping Problem with Six Degrees of Freedom*, Springer, 2008.

[26] PJ Besl and HD McKay, "A method for registration of 3-d shapes", *IEEE Transactions on pattern analysis and machine intelligence*, vol. 14, no. 2, pp. 239–256, 1992.

[27] Zhengyou Zhang, "Iterative point matching for registration of free-form curves", 1992, 271.

[28] J. O'Rourke and JE Goodman, *Handbook of Discrete and Computational Geometry*, CRC Press, 1997.

[29] J.D. Boissonnat and M. Teillaud, *Effective computational geometry for curves and surfaces*, Springer, 2007.

[30] J. Neider, T. Davis, and W. Mason, *OpenGL Programming Guide: The Red Book*, Addison-Wesley, 1994.

[31] D. Calisi, L. Iocchi, and D. Nardi, "A unified benchmark framework for autonomous Mobile robots and Vehicles Motion Algorithms (MoVeMA benchmarks)", *University of Rome*, 2008.

[32] Fabio P. Bonsignorio, John Hallam, and Angel P. del Pobil, "Good experimental methodologies in robotics: State of the art and perspectives", in *Proc. of the Workshop on Performance Evaluation and Benchmarking for Intelligent Robots and Systems, IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007.

[33] I. Ranó and J. Minguez, "Steps towards the automatic evaluation of robot obstacle avoidance algorithms", in *Proc. of Workshop of Benchmarking in Robotics, in the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. Citeseer, 2006.

[34] F. Amigoni, M. Reggiani, and V. Schiaffonati, "An insightful comparison between experiments in mobile robotics and in science", *Autonomous Robots*, pp. 1–13, 2009.

[35] N. Munoz, J. Valencia, and N. Londoño, "Evaluation of navigation of an autonomous mobile robot", in *Proc. of Int. Workshop on Performance Metrics for Intelligent Systems Workshop (PerMIS)*, 2007, pp. 15–21.

[36] Fabio Bonsignorio, John Hallam, and Angel P. del Pobil, "Good experimental methodologiy guidelines", Tech. Rep., EURON Special Interet Group on Good Experimental Methodology, 2008.

[37] D.S. Johnson, "A theoreticiańs guide to the experimental analysis of algorithms", *American Mathematical Society*, vol. 220, no. 5-6, pp. 215–250, 2002.

[38] F.P. Bonsignorio, J. Hallam, and A.P. del Pobil, "Defining the Requisites of a Replicable Robotics Experiment", in *Workshop on GOOD EXPERIMENTAL METHODOLOGY IN ROBOTICS, RSS*, 2009.

[39] F. Amigoni, S. Gasparini, and M. Gini, "Good experimental methodologies for robotic mapping: A proposal", in *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, 2007.

[40] M. Magnusson, A. Nüchter, C. Lörken, A.J. Lilienthal, and J. Hertzberg, "Evaluation of 3D Registration Reliability and Speed–A Comparison of ICP and NDT", in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Kobe, Japan*, 2009, pp. 3907–3912.

[41] A. Nüchter, K. Lingemann, J. Hertzberg, and H. Surmann, "6d slam-3d mapping outdoor environments", *Journal of Field Robotics*, vol. 24, 2007.

[42] B. Mederos, L. Velho, and L.H. De Figueiredo, "Smooth surface reconstruction from noisy clouds", *Journal of the Brazilian Computing Society*, vol. 1, 2004.

[43] T. Zinsser, J. Schmidt, and H. Niemann, "A refined icp algorithm for robust 3-d correspondence estimation", in *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, 2003, vol. 2.

[44] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C.T. Silva, "Computing and Rendering Point set surfaces", in *Proceedings of the conference on Visualization'01*. IEEE Computer Society Washington, DC, USA, 2003, pp. 21–28.

[45] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction", in *Proceedings of the fourth Eurographics symposium on Geometry processing*. Eurographics Association, 2006, p. 70.

[46] JC Carr, RK Beatson, BC McCallum, WR Fright, TJ McLennan, and TJ Mitchell, "Smooth surface reconstruction from noisy range data", *ACM GRAPHITE*, vol. 3, pp. 119–126, 2003.

[47] JC Carr, RK Beatson, JB Cherrie, TJ Mitchell, WR Fright, BC McCallum, and TR Evans, "Reconstruction and representation of 3d objects with radial basis functions", in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM New York, NY, USA, 2001, pp. 67–76.

[48] M. Pauly, M. Gross, and L.P. Kobbelt, "Efficient simplification of point-sampled surfaces", in *IEEE visualization*. Citeseer, 2002, vol. 2002, pp. 163–170.

[49] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Surface reconstruction from unorganized points", *COMPUTER GRAPHICS-NEW YORK-ASSOCIATION FOR COMPUTING MACHINERY-*, vol. 26, pp. 71–71, 1992.

[50] PHS Torr and A. Zisserman, "MLESAC: A new robust estimator with application to estimating image geometry", *Computer Vision and Image Understanding*, vol. 78, no. 1, pp. 138–156, 2000.

[51] N.J. Mitra and A. Nguyen, "Estimating surface normals in noisy point cloud data", in *Proceedings of the nineteenth annual symposium on Computational geometry*. ACM New York, NY, USA, 2003, pp. 322–328.

[52] L. Silva, O.R.P. Bellon, and K.L. Boyer, "Precision range image registration using a robust surface interpenetration measure and enhanced genetic algorithms", *IEEE transactions on pattern analysis and machine intelligence*, pp. 762–776, 2005.

[53] O. Cordón, S. Damas, and J. Santamaría, "A fast and accurate approach for 3D image registration using the scatter search evolutionary algorithm", *Pattern Recognition Letters*, vol. 27, no. 11, pp. 1191–1200, 2006.

[54] S. Carpin and A. Censi, "An experimental assessment of the HSM3D algorithm for sparse and colored data", in *International Conference on Intelligent Robots and Systems (IROS)*, 2009.

[55] D. Akca, "Matching of 3D surfaces and their intensities", *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 62, no. 2, pp. 112–121, 2007.

[56] A.E. Johnson and S.B. Kang, "Registration and integration of textured 3D data", *Image and Vision Computing*, vol. 17, no. 2, pp. 135–147, 1999.

[57] N. Gelfand, N.J. Mitra, L.J. Guibas, and H. Pottmann, "Robust global registration", in *Symposium on Geometry Processing*, 2005, vol. 2, p. 5.

[58] G.C. Sharp, S.W. Lee, and D.K. Wehe, "ICP registration using invariant features", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 90–102, 2002.

[59] R.B. Rusu, N. Blodow, Z.C. Marton, and M. Beetz, "Aligning point cloud views using persistent feature histograms", in *Proceedings of the 21st IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Nice, France*, 2008.

[60] R.B. Rusu, N. Blodow, and M. Beetz, "Fast Point Feature Histograms (FPFH) for 3D Registration", in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Kobe, Japan*, 2009.

[61] A. Makadia, AI Patterson, and K. Daniilidis, "Fully automatic registration of 3D point clouds", in *CVPR*. Citeseer, 2006, vol. 6, pp. 1297–1304.

[62] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm", in *Proc. 3DIM*, 2001, pp. 145–152.

[63] A. Nüchter, K. Lingemann, and J. Hertzberg, "Cached kd tree search for icp algorithms", in *3-D Digital Imaging and Modeling, 2007. 3DIM'07. Sixth International Conference on*, 2007, pp. 419–426.

[64] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions", *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[65] C. Silpa-Anan and R. Hartley, "Localisation using an image-map", in *Proceedings of the Australian Conference on Robotics and Automation*, 2004.

[66] C. Silpa-Anan and R. Hartley, "Optimised KD-trees for fast image descriptor matching", in *Proc. CVPR*, 2008, pp. 1–8.

[67] T. Liu, A.W. Moore, A. Gray, and K. Yang, "An investigation of practical approximate nearest neighbor algorithms", in *Advances in neural information processing systems*. 2004, Citeseer.

[68] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree", in *Proc. CVPR*, 2006, vol. 5.

[69] M. Muja and D.G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration", in *International Conference on Computer Vision Theory and Applications (VISAPP'09)*, 2009.

[70] B. Leibe, K. Mikolajczyk, and B. Schiele, "Efficient clustering and matching for object class recognition", in *Proc. BMVC*, 2006.

[71] K. Mikolajczyk and J. Matas, "Improving descriptors for fast tree matching by optimal linear projection", in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*. Citeseer, 2007, pp. 1–8.

[72] Michael Connor and Piyush Kumar, "Fast construction of k-nearest neighbor graphs for point clouds", in *IEEE Transactions on Visualization and Computer Graphics*, September 2009.

[73] D. Qiu, S. May, and A. Nüchter, "GPU-accelerated Nearest Neighbor Search for 3D Registration", 2009.

[74] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware", in *ACM SIGGRAPH Asia 2008 papers*. ACM, 2008, p. 126.

[75] A. Nüchter, O. Wulf, K. Lingemann, J. Hertzberg, B. Wagner, and H. Surmann, "3D mapping with semantic knowledge", *Lecture Notes in Computer Science*, vol. 4020, pp. 335, 2006.

[76] C. Langis, M. Greenspan, and G. Godin, "The parallel iterative closest point algorithm", in *Proceedings of the Third International Conference on*, 2001, pp. 195–202.

[77] A Nüchter, "Parallel and Cached Scan Matching for Robotic 3D Mapping", *Journal of Computing and Information Technology*, vol. 17, no. 1, pp. 51–65, 2009.

[78] KS Arun, TS Huang, and SD Blostein, "Least-squares fitting of two 3-D point sets.", *IEEE TRANS. PATTERN ANAL. MACH. INTELLIG.*, vol. 9, no. 5, pp. 698–700, 1987.

[79] B.K.P. Horn, H.M. Hilden, and S. Negahdaripour, "Closed-form solution of absolute orientation using orthonormal matrices", *Journal of the Optical Society of America A*, vol. 5, no. 7, pp. 1127–1135, 1988.

[80] B.K.P. Horn et al., "Closed-form solution of absolute orientation using unit quaternions", *Journal of the Optical Society of America A*, vol. 4, no. 4, pp. 629–642, 1987.

[81] M.W. Walker, L. Shao, and R.A. Volz, "Estimating 3-D location parameters using dual number quaternions", *CVGIP: image understanding*, vol. 54, no. 3, pp. 358–367, 1991.

[82] DW Eggert, A. Lorusso, and RB Fisher, "Estimating 3-D rigid body transformations: a comparison of four major algorithms", *Machine Vision and Applications*, vol. 9, no. 5, pp. 272–290, 1997.

[83] H. Pottmann, S. Leopoldseder, and M. Hofer, "Registration without ICP", *Computer Vision and Image Understanding*, vol. 95, no. 1, pp. 54–71, 2004.

[84] Radu Bogdan Rusu, Aravind Sundaresan, Benoit Morisset, Kris Hauser, Motilal Agrawal, and Jean-Claude Latombe, "Leaving Flatland: Efficient real-time three-dimensional perception and motion planning", *Journal of Field Robotics*, vol. 26, no. 10, 2009.

[85] Martin Magnusson, Achim Lilienthal, and Tom Duckett, "Scan registration for autonomous mining vehicles using 3D-NDT", *Journal of Field Robotics*, vol. 24, no. 10, pp. 803–827, 2007.

[86] R. Schnabel, R. Wahl, and R. Klein, "Efficient RANSAC for point-cloud shape detection", in *Computer Graphics Forum*. Citeseer, 2007, vol. 26, pp. 214–226.

[87] C. Brenneke, O. Wulf, and B. Wagner, "Using 3d laser range data for slam in outdoor environments", in *2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings*, 2003, vol. 1.

[88] C. Parra, R. Murrieta-Cid, M. Devy, and M. Briot, "3-d modelling and robot localization from visual and range data in natural scenes", *Lecture Notes in Computer Science*, pp. 450–468, 1998.

[89] J. Weingarten, G. Gruener, and R. Siegwart, "A fast and robust 3D feature extraction algorithm for structured environment reconstruction", in *International Conference on Advanced Robotics, Coimbra, Portugal*. Citeseer, 2003.

[90] A. Sappa and M. Devy, "Fast range image segmentation by an edge detection strategy", in *Proceedings of Third International Conference on 3-D Digital Imaging and Modeling*, 2001, vol. 3.

[91] PJ Besl and RC Jain, "Segmentation through variable-order surface fitting", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 2, pp. 167–192, 1988.

[92] T. Rabbani, F. van den Heuvel, and G. Vosselmann, "Segmentation of point clouds using smoothness constraint", *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 36, no. 5, pp. 248–253, 2006.

[93] E. Bittar, N. Tsingos, and M.P. Gascuel, "Automatic reconstruction of unstructured 3d data: Combining medial axis and implicit surfaces", in *Computer Graphics Forum*. Citeseer, 1995, vol. 14, pp. 457–468.

[94] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H.P. Seidel, "Multi-level partition of unity implicits", p. 173, 2005.

[95] G. Guennebaud and M. Gross, "Algebraic point set surfaces", *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, pp. 23, 2007.

[96] C. Oztireli, G. Guennebaud, and M. Gross, "Feature preserving point set surfaces based on non-linear kernel regression", in *Computer Graphics Forum*. Blackwell Publishing, 2009, vol. 28, pp. 493–501.

[97] J.D. Boissonnat, "Geometric structures for three-dimensional shape representation", *ACM Transactions on Graphics (TOG)*, vol. 3, no. 4, pp. 266–286, 1984.

[98] N. Amenta, M. Bern, and M. Kamvysselis, "A new Voronoi-based surface reconstruction algorithm", in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM New York, NY, USA, 1998, pp. 415–421.

[99] N. Amenta, S. Choi, TK Dey, and N. Leekha, "A simple algorithm for homeomorphic surface reconstruction", in *Proceedings of the sixteenth annual symposium on Computational geometry*. ACM, 2002, p. 222.

[100] N. Amenta, S. Choi, and R.K. Kolluri, "The power crust, unions of balls, and the medial axis transform", *Computational Geometry: Theory and Applications*, vol. 19, no. 2-3, pp. 127–153, 2001.

[101] T.K. Dey and S. Goswami, "Tight cocone: a water-tight surface reconstructor", *Journal of Computing and Information Science in Engineering*, vol. 3, pp. 302, 2003.

[102] T.K. Dey and S. Goswami, "Provable surface reconstruction from noisy samples", *Computational Geometry: Theory and Applications*, vol. 35, no. 1-2, pp. 124–141, 2004.

[103] R. Kolluri, J.R. Shewchuk, and J.F. O'Brien, "Spectral surface reconstruction from noisy point clouds", in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. ACM New York, NY, USA, 2004, pp. 11–21.

[104] H. Edelsbrunner and E.P. Mücke, "Three-dimensional alpha shapes", in *Proceedings of the 1992 workshop on Volume visualization*. ACM, 1992, p. 82.

[105] P. Labatut, J.P. Pons, and R. Keriven, "Efficient multi-view reconstruction of large-scale scenes using interest points, delaunay triangulation and graph cuts", *Computer Vision*, pp. 1–8, 2007.

[106] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, G. Taubin, et al., "The ball-pivoting algorithm for surface reconstruction", *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 4, pp. 349–359, 1999.

[107] J. Schöberl, "NETGEN An advancing front 2D/3D-mesh generator based on abstract rules", *Computing and Visualization in Science*, vol. 1, no. 1, pp. 41–52, 1997.

[108] S. Rebay, "Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer-Watson algorithm", *Journal of Computational Physics*, vol. 106, no. 1, pp. 125–138, 1993.

[109] P. Cignoni, C. Rocchini, and R. Scopigno, "Metro: measuring error on simplified surfaces", in *Computer Graphics Forum*, 1998, vol. 17, pp. 167–174.

[110] Thomas Wisspeintner, Walter Nowak, and Ansgar Bredenfeld, *RoboCup 2005: Robot Soccer World Cup IX*, vol. 4020, chapter VolksBot - A Flexible Component-Based Mobile Robot System, pp. 716–723, Springer Berlin Heidelberg, 2006.

[111] A.S. Tanenbaum and M Van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2nd edition, October 2006.

[112] P. Martz, "OpenSceneGraph Quick Start Guide", Tech. Rep., Skew Matrix Software, 2007.

[113] Morten Strandberg, *Robot path planning: an object-oriented approach*, PhD thesis, KTH, Signals, Sensors and Systems, 2004.

[114] L. Andrade, JL Fiadeiro, J. Gouveia, and G. Koutsoukos, "Separating computation, coordination and configuration", *Software Focus*, vol. 14, no. 5, pp. 353–369, 2002.

[115] M. Radestock and S. Eisenbach, "Coordinating components in middleware systems", *Software Focus*, vol. 15, no. 13, pp. 1205–1231, 2003.

[116] M. Radestock and S. Eisenbach, "Coordination in evolving systems", *Lecture Notes in Computer Science*, pp. 162–176, 1996.

[117] S.J. Owen, "A survey of unstructured mesh generation technology", in *7th International Meshing Roundtable*. Citeseer, 1998, vol. 3.

[118] J. Peng, C.S. Kim, and C.C. Jay Kuo, "Technologies for 3d mesh compression: A survey", *Journal of Visual Communication and Image Representation*, vol. 16, no. 6, pp. 688–733, 2005.

[119] R.C. Martin, *Agile software development: principles, patterns, and practices*, Prentice Hall PTR Upper Saddle River, NJ, USA, 2003.

Appendix A

UML Class diagrams

This Appendix presents UML class diagrams of the investigated parts in the public available libraries. Please note that some minor relevant methods are not displayed, to improve the readability.

A.1 Data-type representations in existing libraries

A.1.1 Cartesian point representations in existing libraries

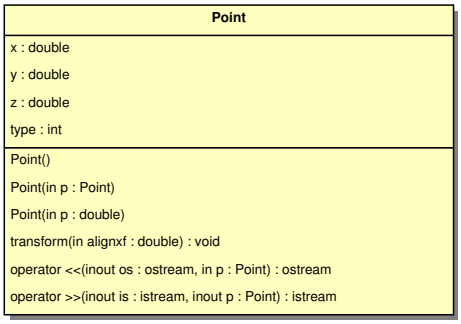


Figure A.1: UML class diagram of Cartesian point representation in 6DSLAM library.

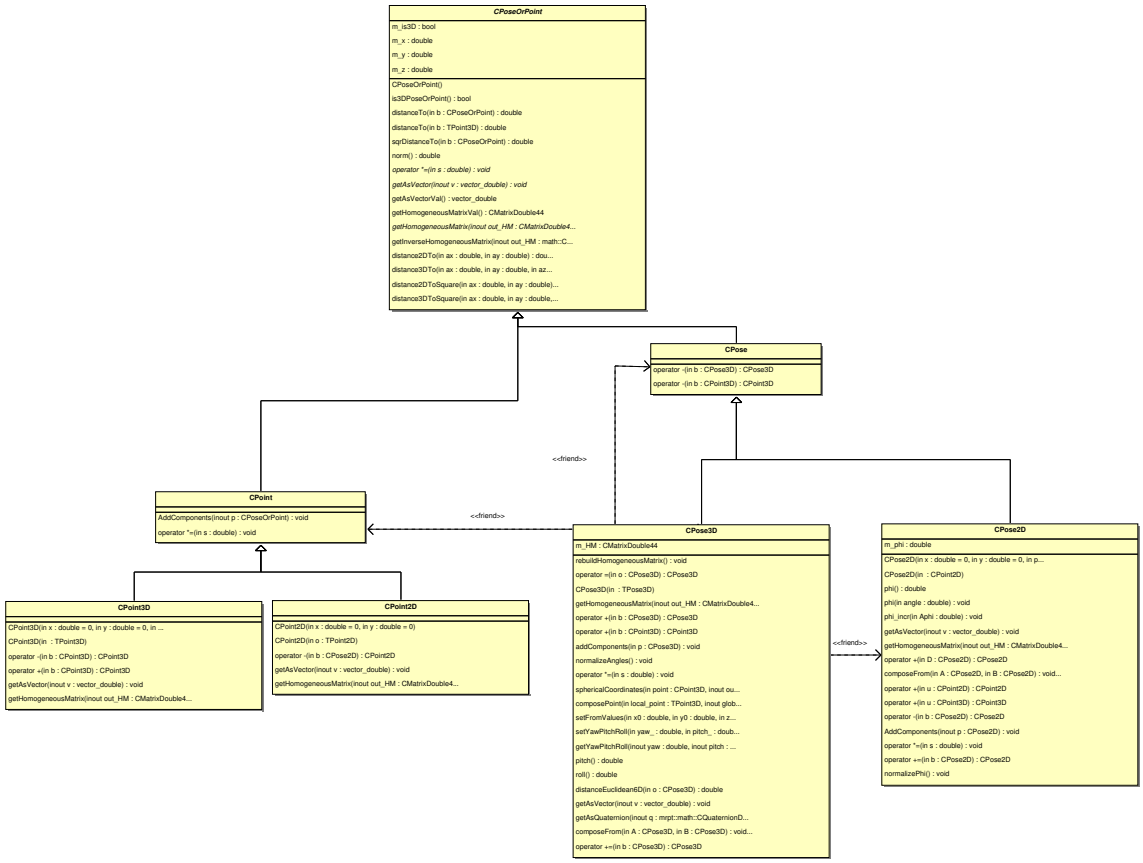


Figure A.2: UML class diagram of Cartesian point representation in MRPT library.

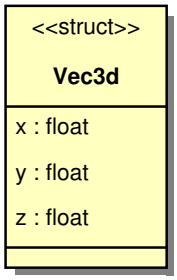


Figure A.3: UML class diagram of Cartesian point representation in IVT library.

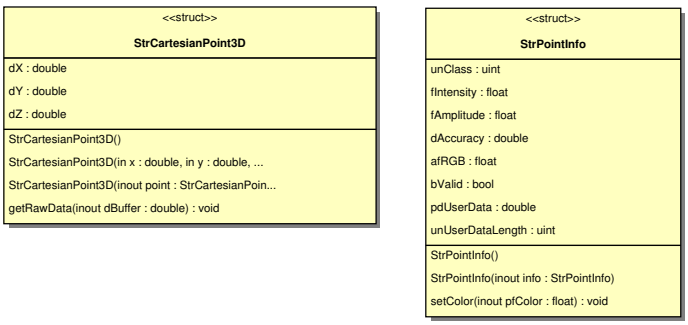


Figure A.4: UML class diagram of Cartesian point representation in FAIR library.

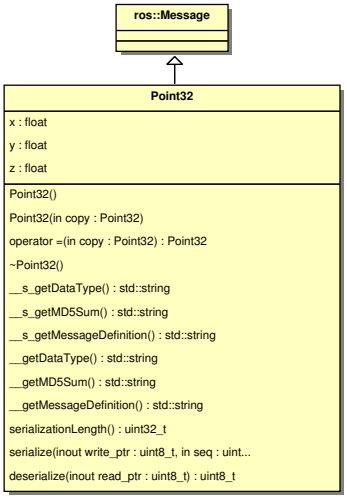


Figure A.5: UML class diagram of Cartesian point representation in ROS.

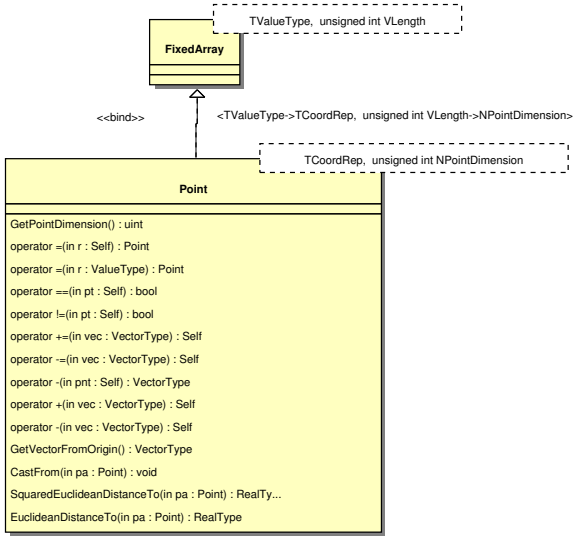


Figure A.6: UML class diagram of Cartesian point representation in ITK library.

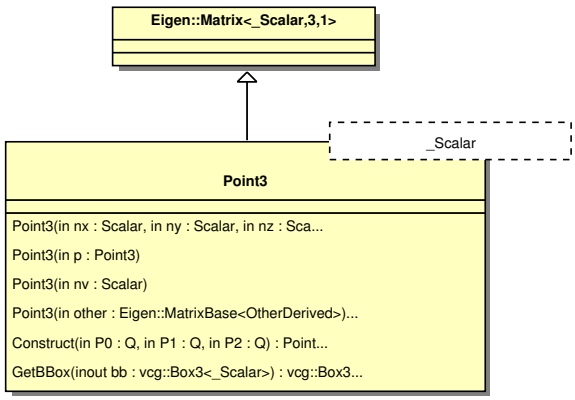


Figure A.7: UML class diagram of Cartesian point representation in Meshlab.

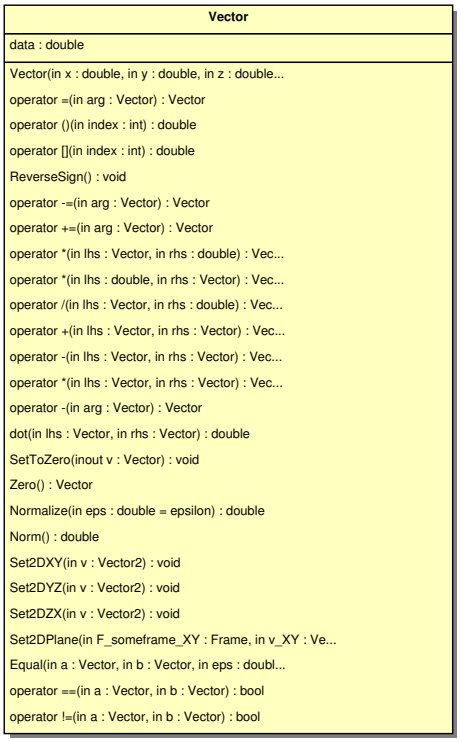


Figure A.8: UML class diagram of Cartesian point representation in KDL library.

A.1.2 Cartesian point cloud representations in existing libraries

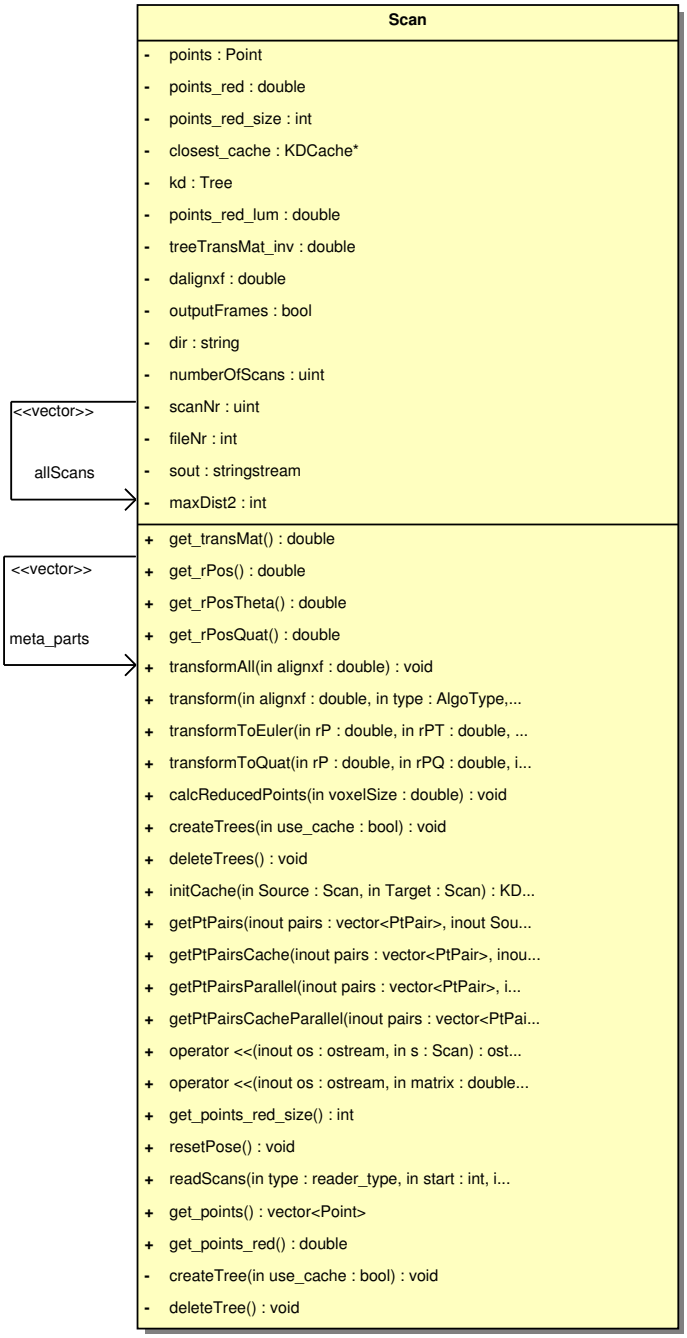


Figure A.9: UML class diagram of Cartesian point cloud representation in 6DSLAM library.

Figure A.10: UML class diagram of Cartesian point cloud representation in MRPT library.

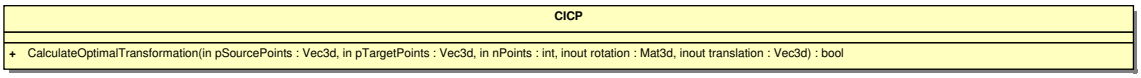


Figure A.11: UML class diagram of Cartesian point cloud representation in IVT library.

Figure A.12: UML class diagram of Cartesian point cloud representation in FAIR library.

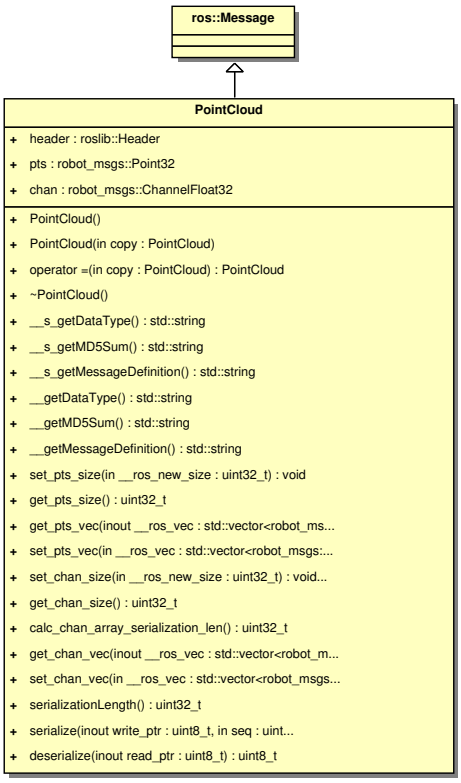


Figure A.13: UML class diagram of Cartesian point cloud representation in ROS.

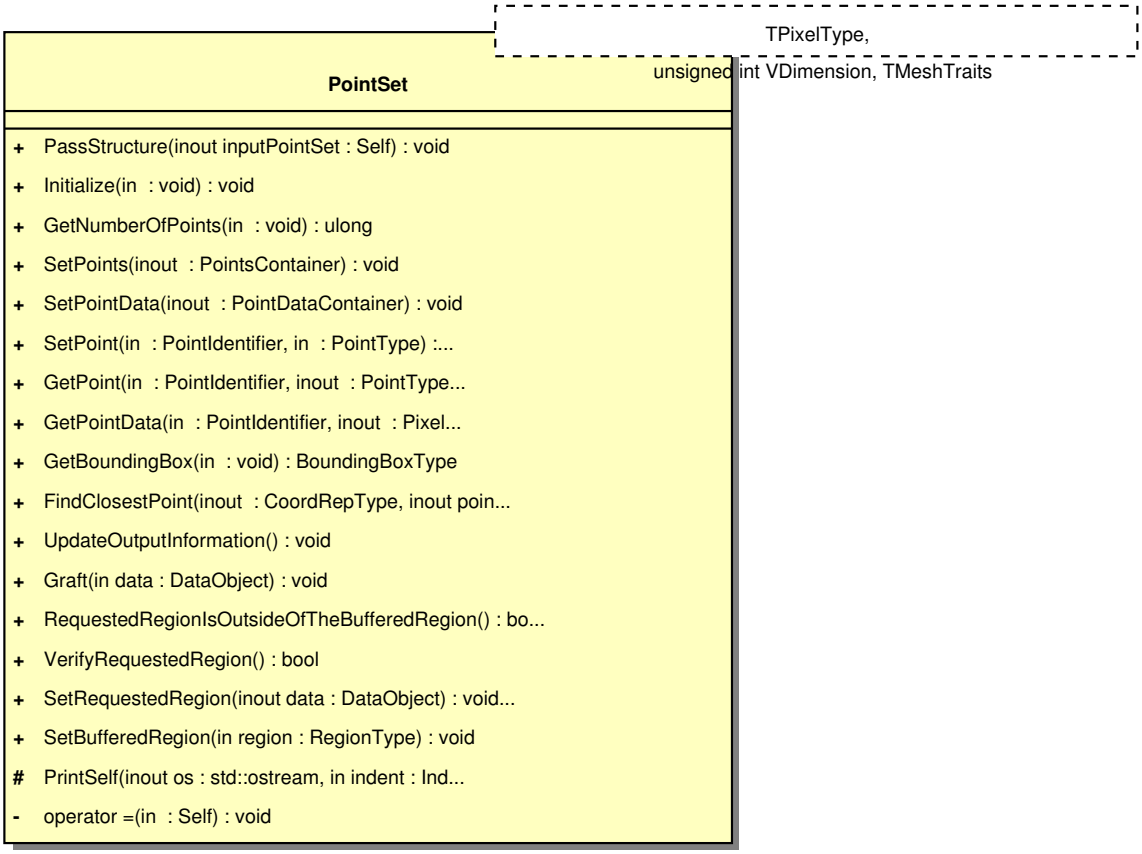


Figure A.14: UML class diagram of Cartesian point cloud representation in ITK library.

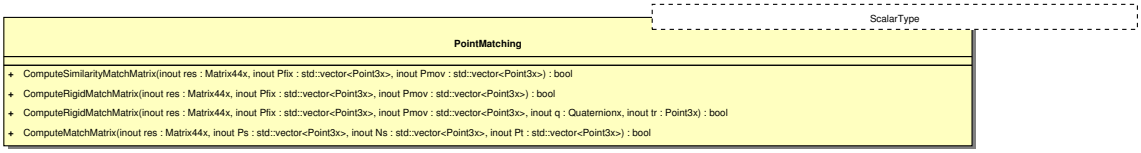


Figure A.15: UML class diagram of Cartesian point cloud representation in Meshlab.

A.1.3 Tringle mesh representations in existing libraries

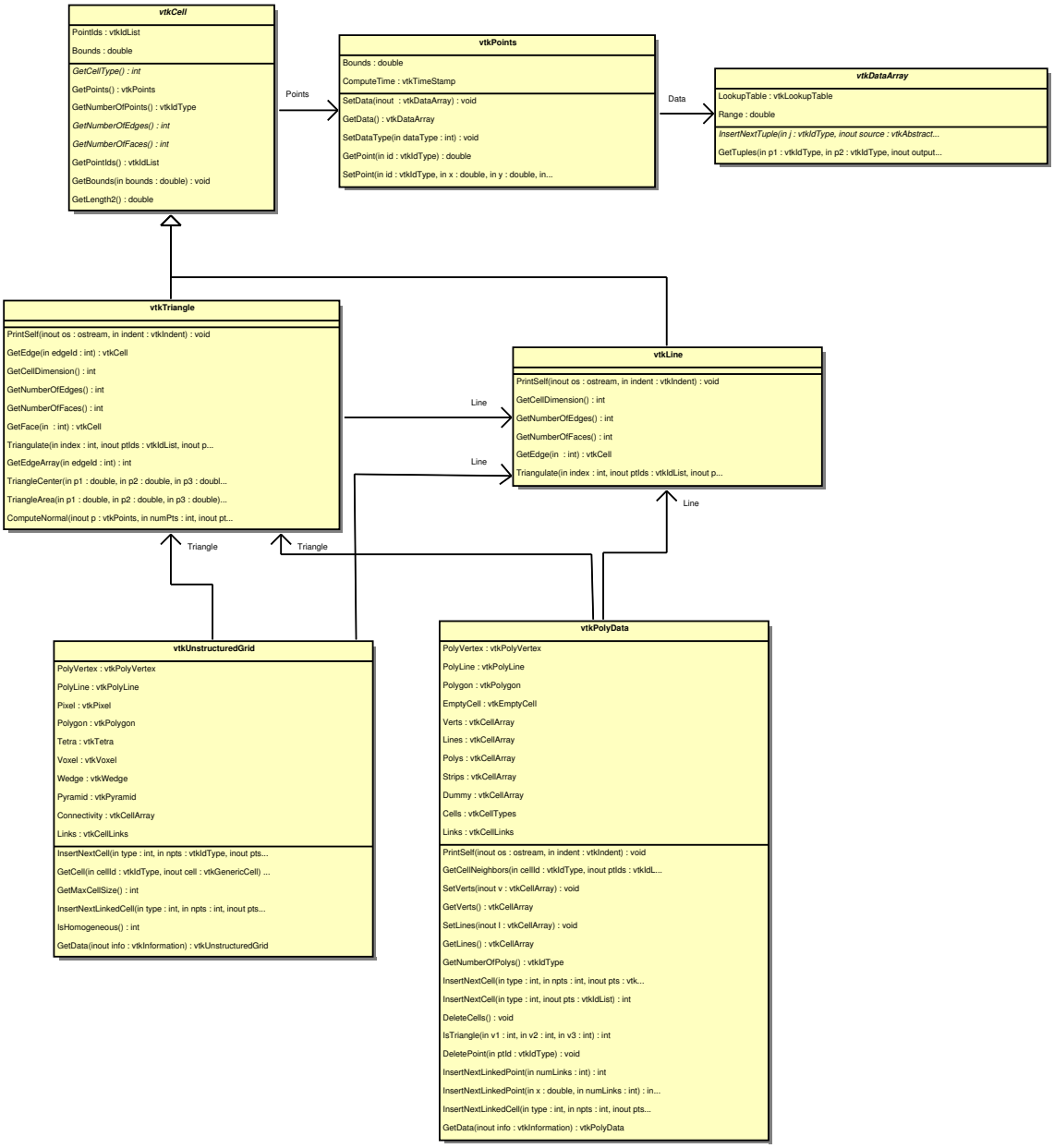


Figure A.16: UML class diagram of triangle mesh representation in VTK.

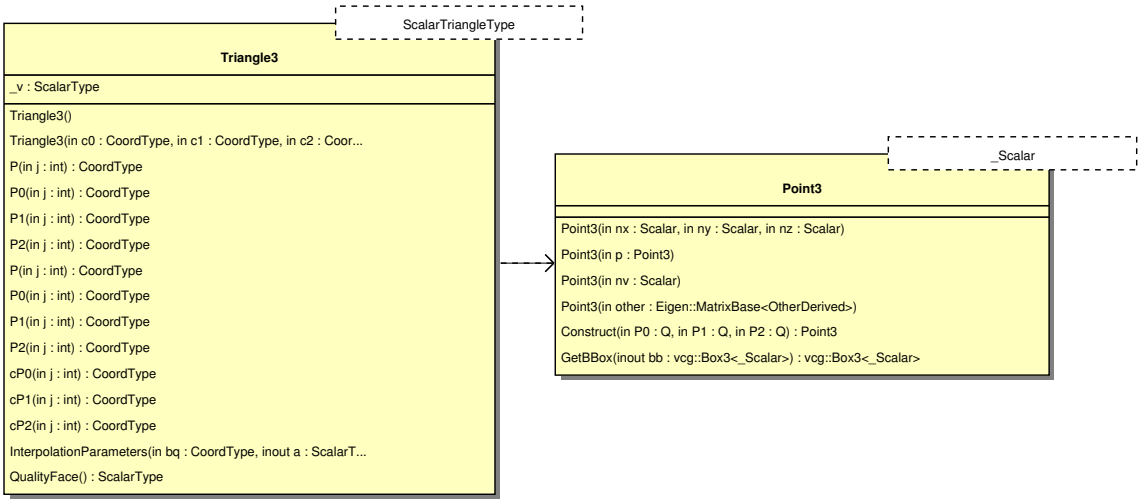


Figure A.17: UML class diagram of triangle representation in Meshlab.

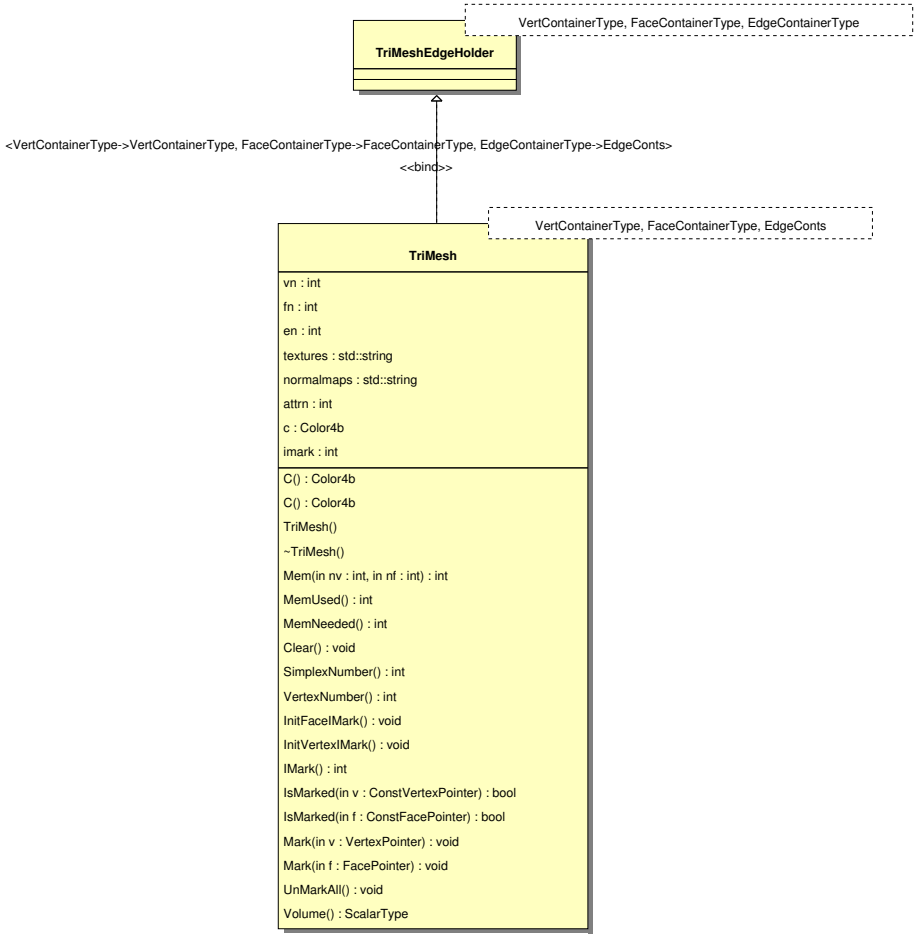


Figure A.18: UML class diagram of triangle mesh representation in Meshlab.

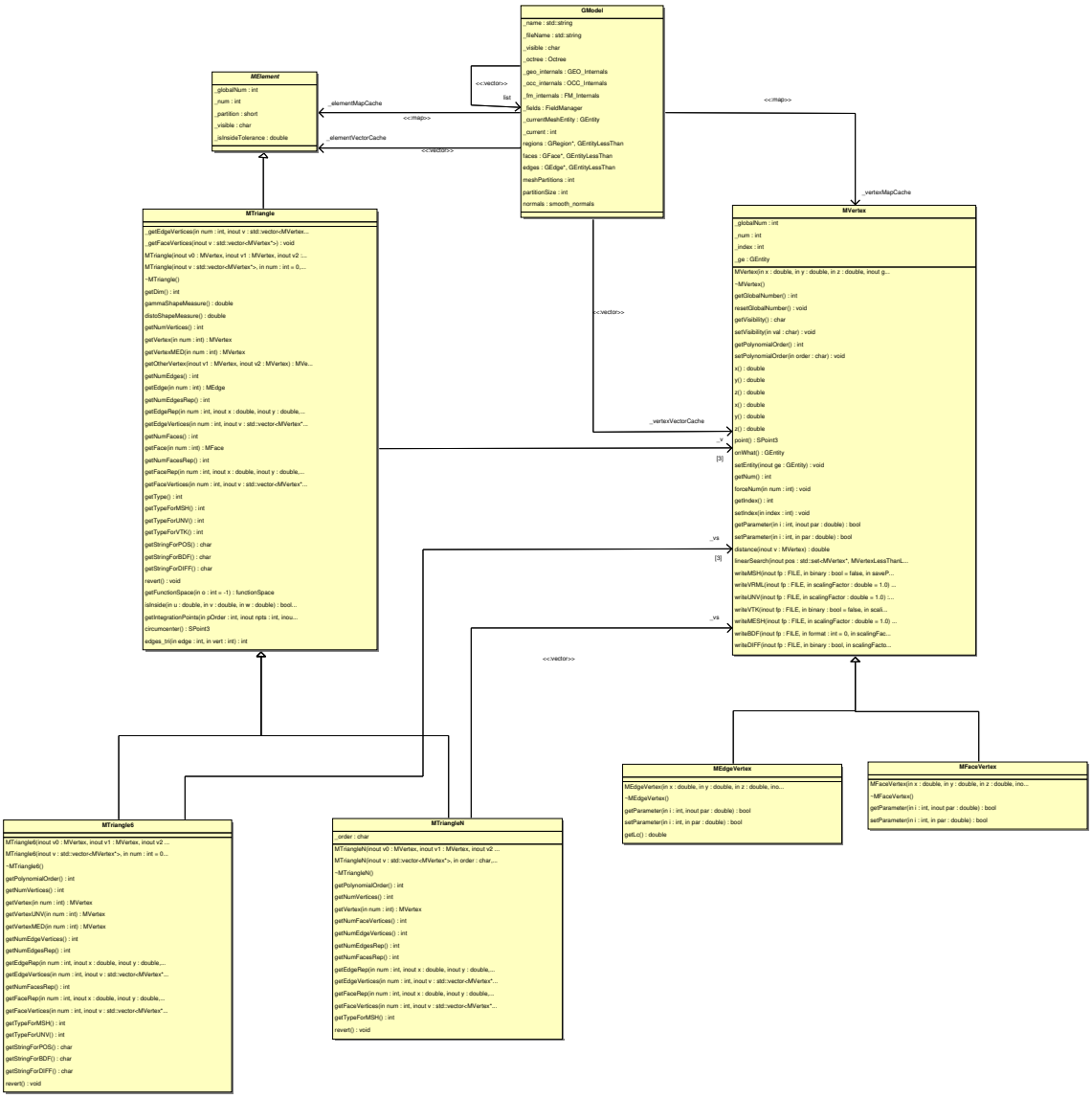


Figure A.19: UML class diagram of triangle mesh representation in Gmsh library.

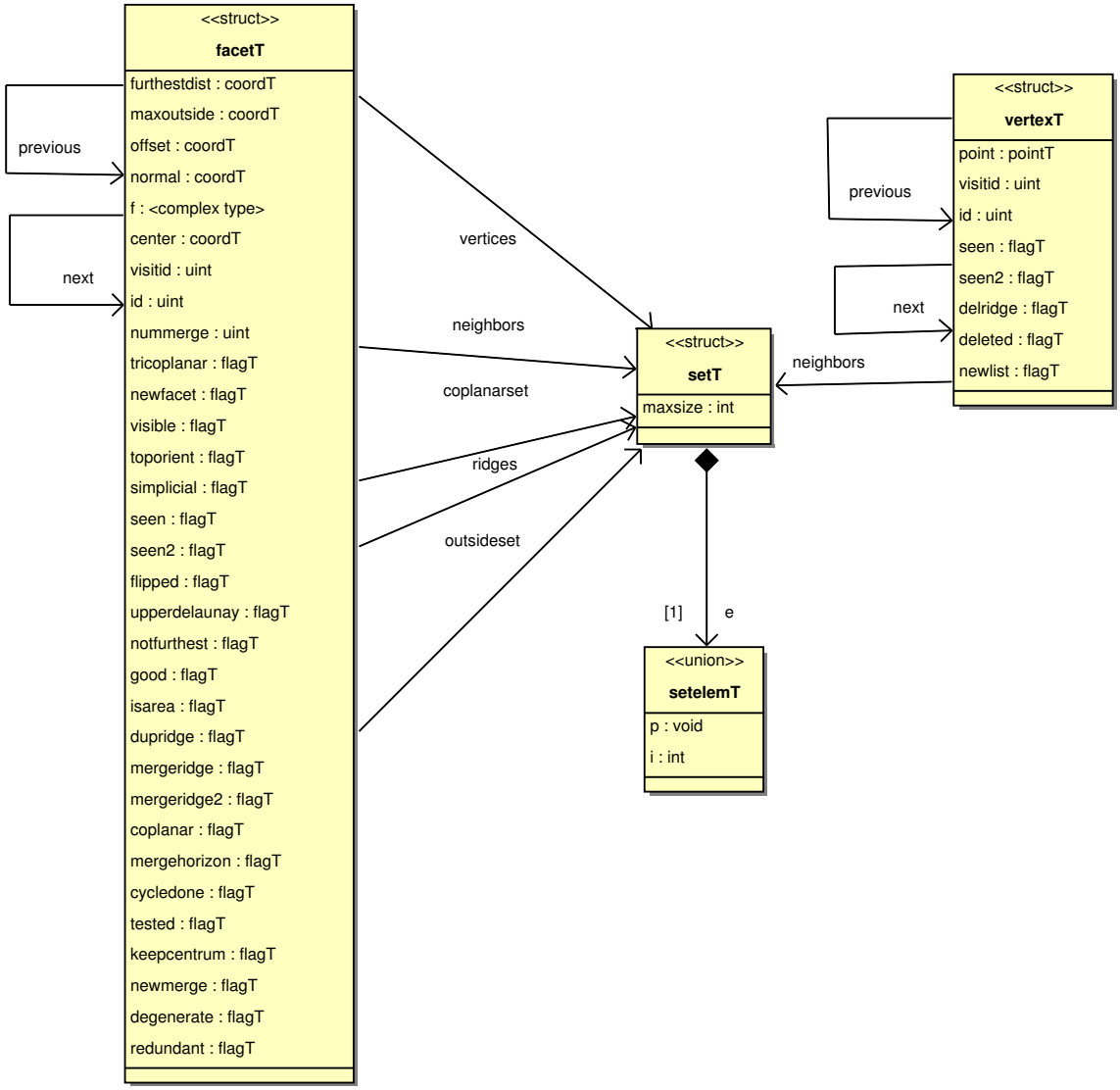


Figure A.20: UML class diagram of triangle mesh representation in Qhull library.

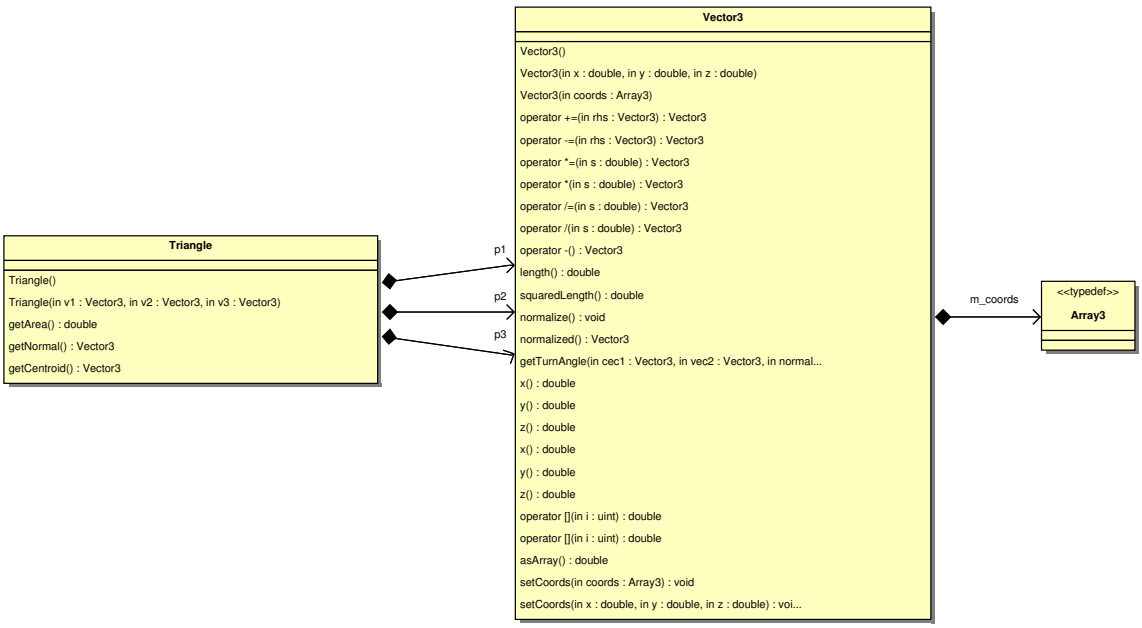


Figure A.21: UML class diagram of triangle mesh representation in CoPP and BRICS_MM library.

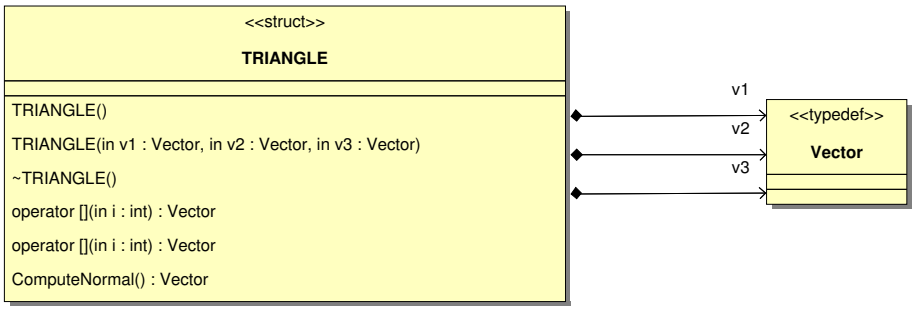


Figure A.22: UML class diagram of triangle representation in openrave library.

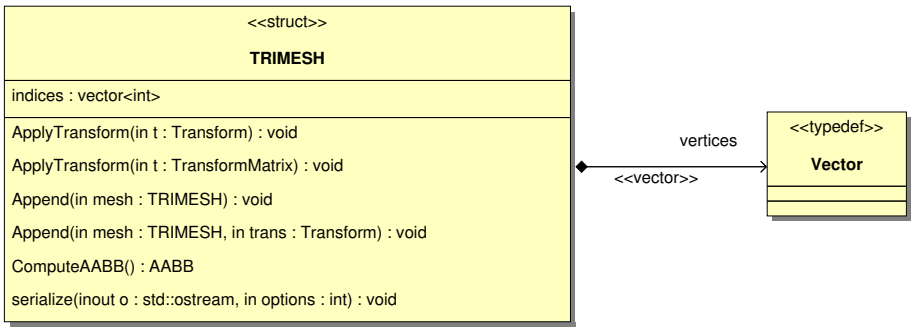


Figure A.23: UML class diagram of triangle mesh representation in openrave library.

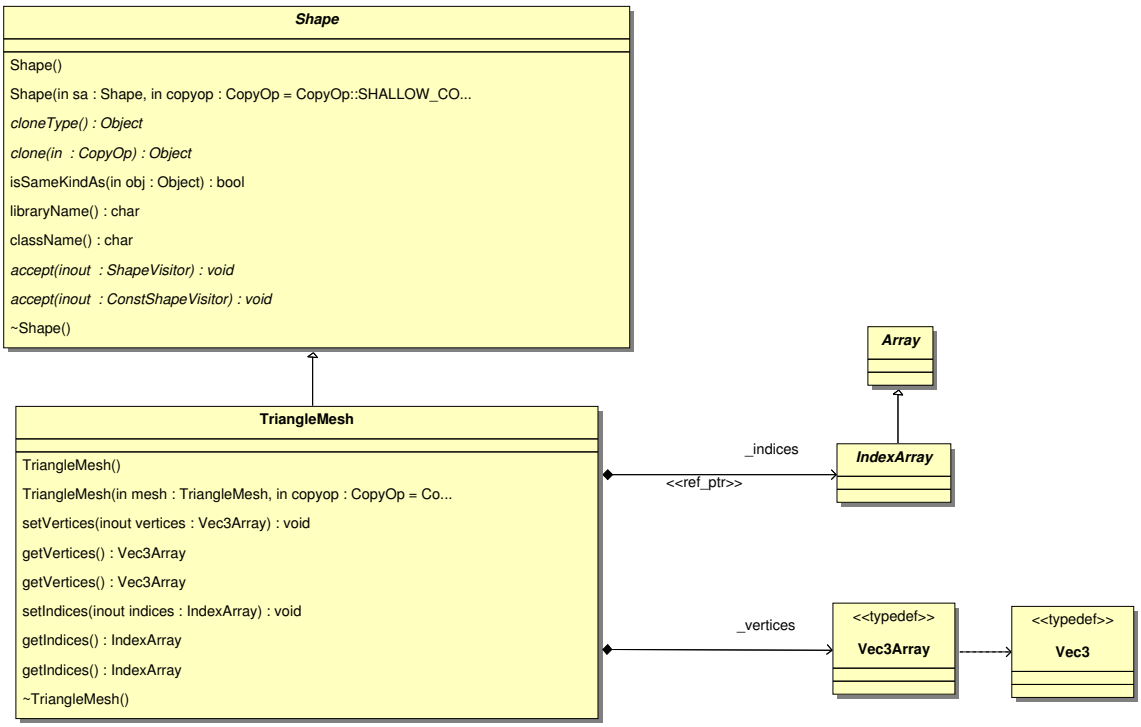


Figure A.24: UML class diagram of triangle mesh representation in OSG library.

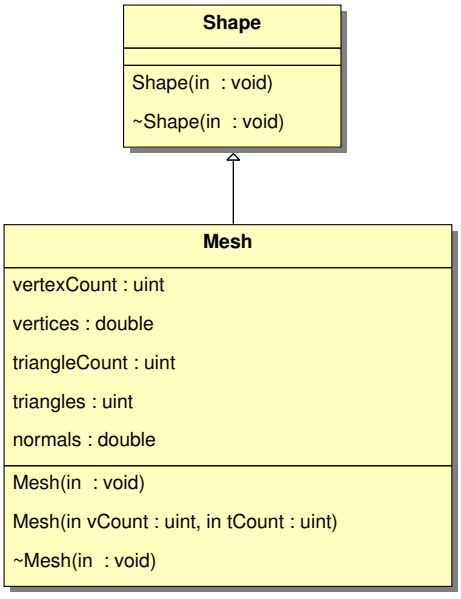


Figure A.25: UML class diagram of triangle mesh representation in ROS.

A.2 Implementation details

A.2.1 Implementation of harmonized data-types

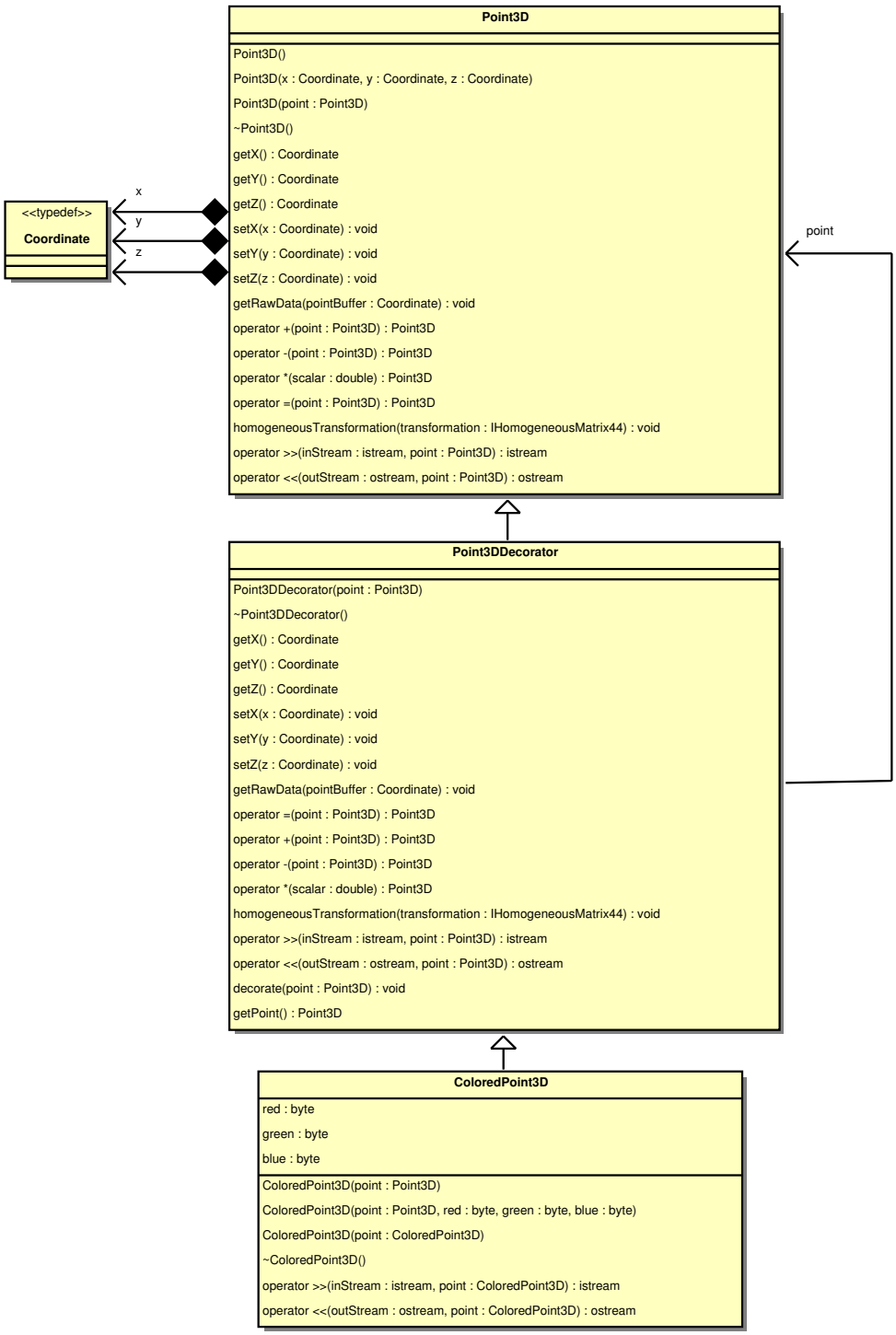


Figure A.26: UML class diagram of harmonized Cartesion point representation.

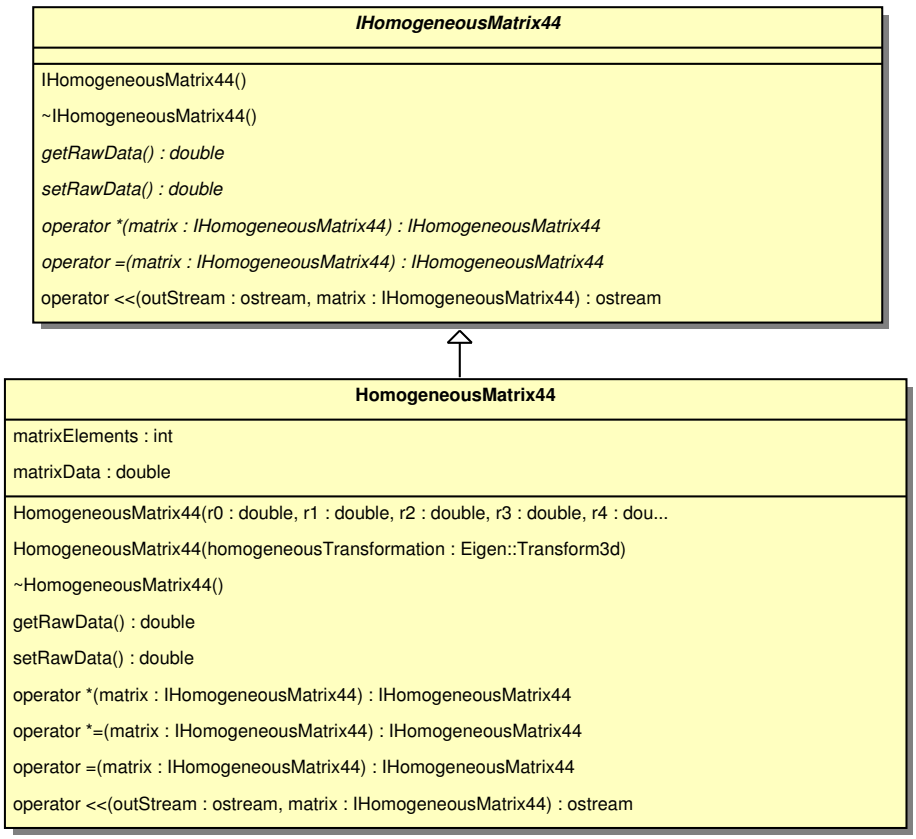


Figure A.27: UML class diagram of homogeneous transformation matrix.

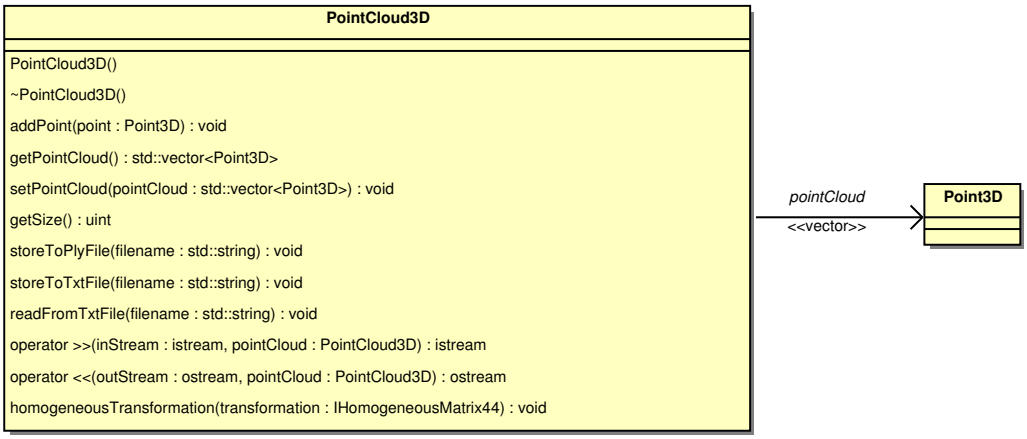


Figure A.28: UML class diagram of harmonized Cartesion point representation.

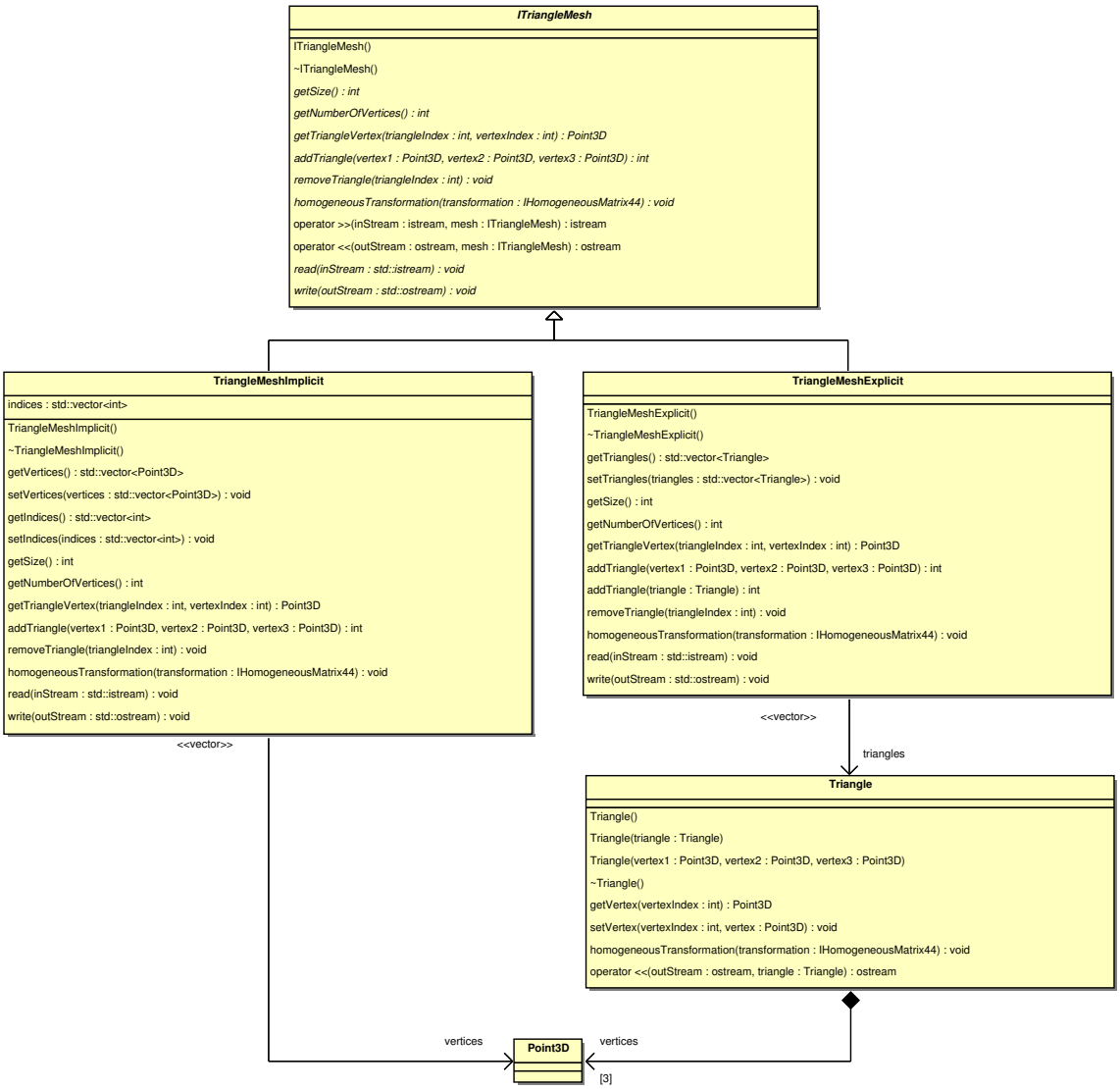


Figure A.29: UML class diagram of harmonized Cartesion point representation.

A.2.2 Implementation of components

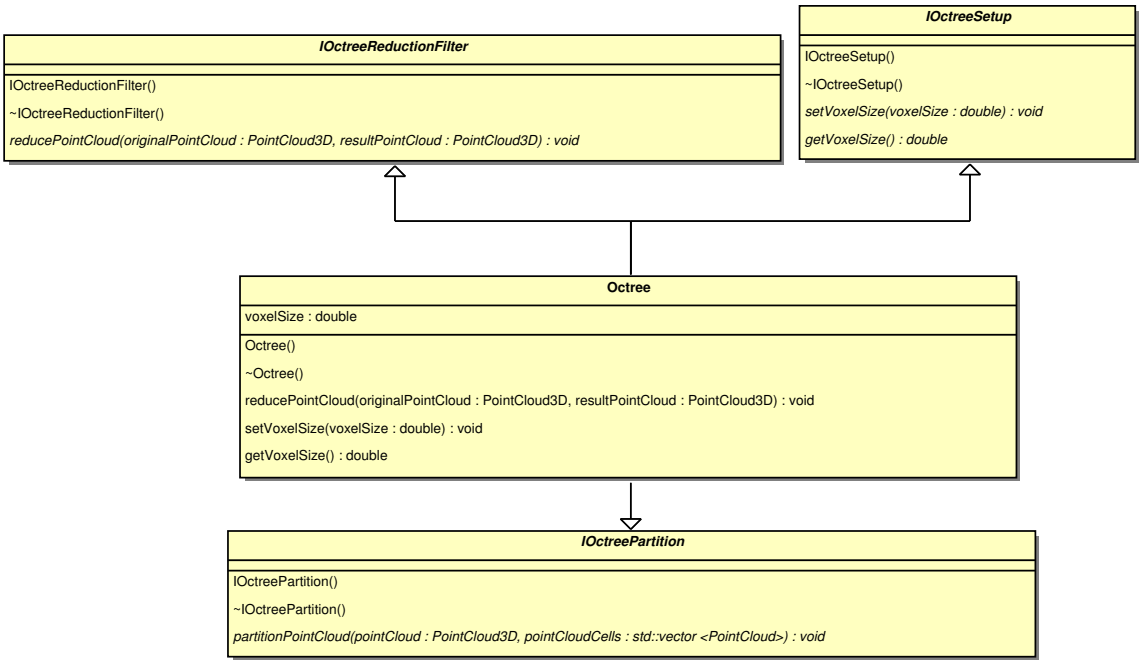


Figure A.30: UML class diagram of Octree component implementation.

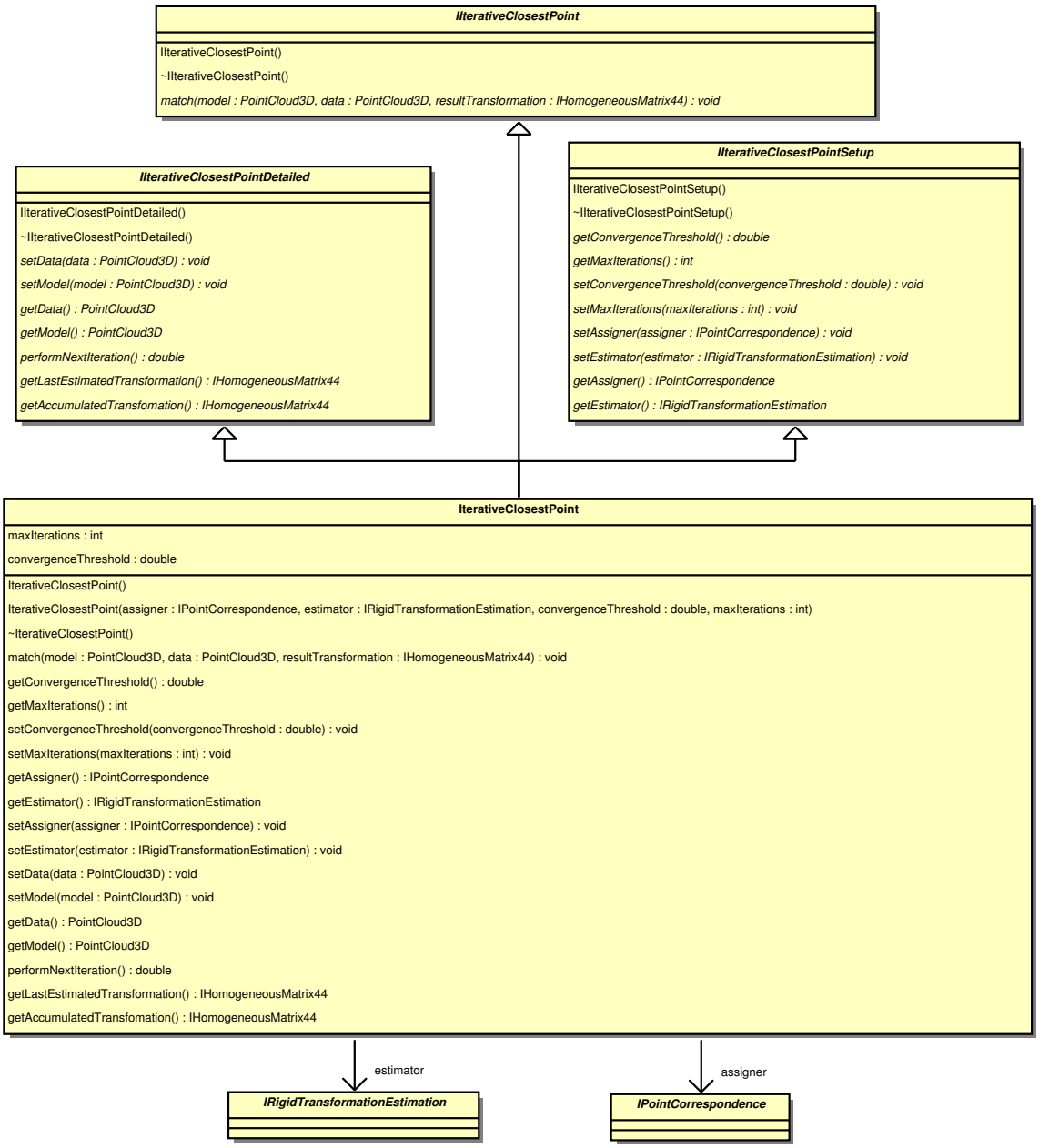


Figure A.31: UML class diagram for Iterative Closest Point component implementation.

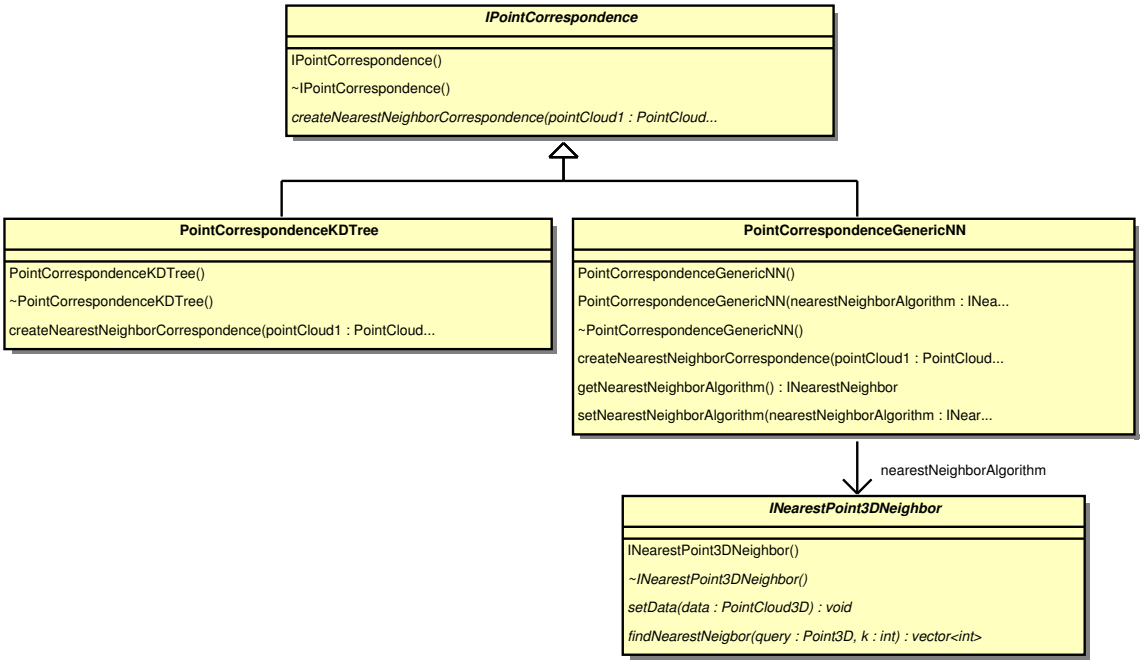


Figure A.32: UML class diagram for Point Correspondence component implementation.

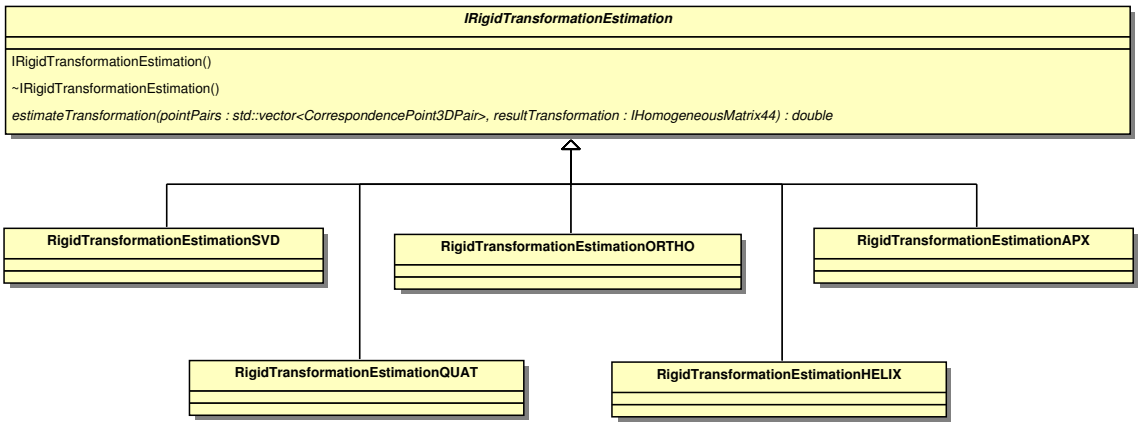


Figure A.33: UML class diagram for Rigid Transformation Estimation component implementation.

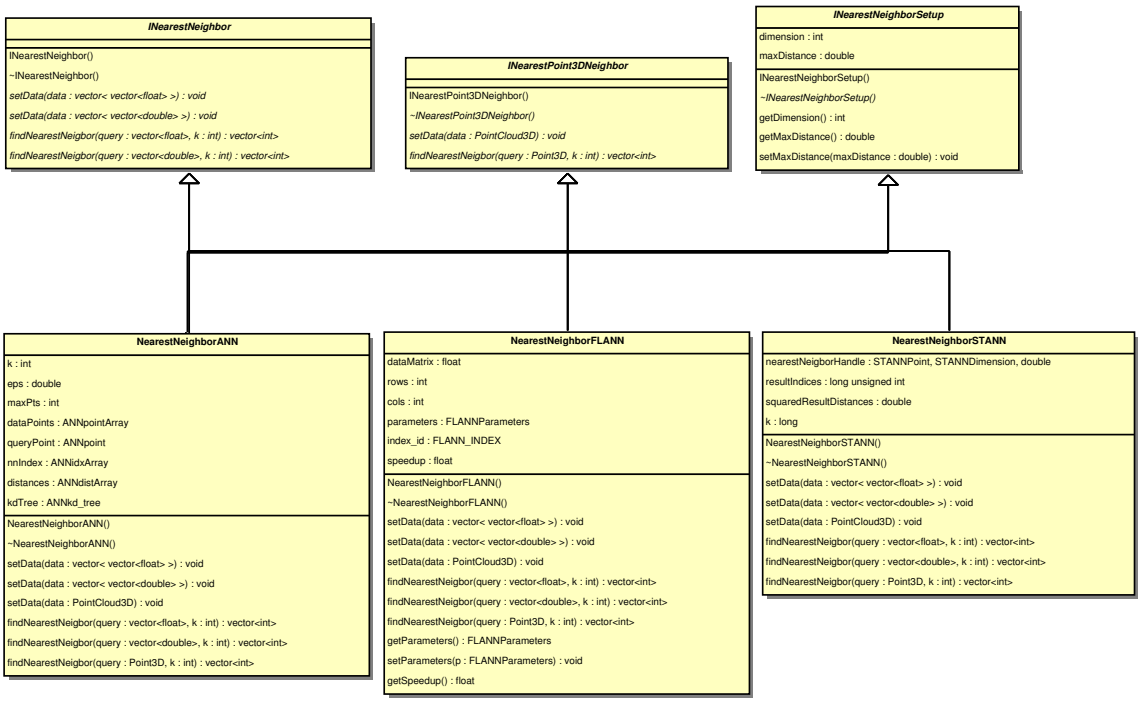


Figure A.34: UML class diagram for k-Nearest Neighbor search component implementation. Note that all implementations inherit from the three abstract interfaces.

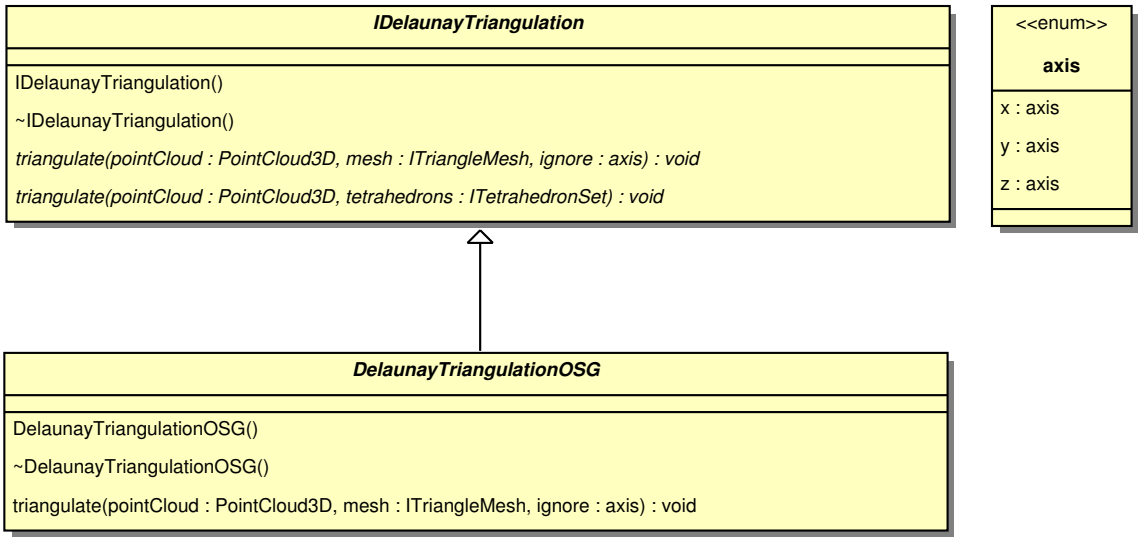


Figure A.35: UML class diagram for Delaunay Triangulation component implementation.