NTNU
Norwegian University of
Science and Technology

Robust Volumetric 3-D Reconstruction in a Dynamic Environment

Roy Konrad Angelsen

Master of Science in Cybernetics and Robotics
Submission date: May 2018
Supervisor: Annette Stahl, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem description

Neodroid is a research project at SINTEF Ocean with the purpose of developing a general purpose humanoid robot that is able to perform complex repetitive tasks that are generally performed by humans today.

This study aims to develop a basis for the Neodroid vision system. More precisely, this task is broken down into two main parts. The first part consists of reconstructing the volume in which the robot is to operate, referred to as workspace, from multiple viewing angles. The second part consists of detecting where changes have occurred in the volume as seen from an overview camera, and determining where to place the cameras mounted on the robots' arms, to obtain a close-up view of these novel regions of interest.

Abstract

As a part of a research project at SINTEF Ocean AS named *Neodroid* aiming at developing a general purpose humanoid robot to execute a variety of complex tasks, the need for perceptive ability arises. The robot consists of two robotic manipulators and is intended to solve tasks demonstrated by a human in a virtual environment. This study aims at developing a vision system that is capable of keeping an accurate reconstruction of the robot workspace. To achieve this, a volumetric representation method consisting of a three-dimensional grid of voxels is used. To keep this reconstruction up to date, the changing regions of the volume need to be scanned. This is achieved by performing novelty detection with RDE[1] on depth images from a camera viewing the entire volume in a top-down configuration. The novel regions are then reconstructed in three dimensions and clustering is performed to draw bounding boxes around these. Camera viewpoints are generated on a circular path around each novel region to obtain an accurate measurement.

[1] Recursive Density Estimation

Sammendrag

Denne studien er gjort i forbindelse med et forskningsprosjekt på SINTEF Ocean AS der formålet er å utvikle en generell humanoid robot kalt *Neodroid*, som består av to robotarmer og tre dybdesensorer. Denne roboten har som formål å utføre komplekse oppgaver som den får demonstrert av et menneske ved hjelp av virtuell virkelighet (VR). Behovet for en presis forståelse av omgivelsene den opererer i er derfor essensiell. Denne studien tar sikte på å utvikle og implementere en metode for å la roboten holde en oppdatert representasjon av det tredimensjonale rommet den opererer i. For å oppnå dette, rekonstrueres roboten arbeidsområde som et vokselbilde. For å holde rekonstruksjonen oppdatert benyttes et kamera med evne til å oppfatte dybde til å måle arbeidsområdet. Ett kamera er montert slik at det får overblikk over arbeidsområdet. Endringer i arbeidsområdet finnes fra bildestrømmen fra dette kamera med algoritmen RDE (Recursive Density Estimation). Disse regionene tilnærmes som tredimensjonale bokser ved å analysere grupperingene av endringene som sett fra oversiktskamera, før de deretter skal bli fotografert med dybdekamera montert på robotens armer fra nært hold og forskjellige vinkler.

Preface

This report is the result of a study performed at the 10th semester towards the degree of Master of Technology at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The study has been completed in collaboration with SINTEF Ocean AS as part of an ongoing research project.

The result of this study is largely in the form of software implementations. These implementations are to a large degree developed independently, with exceptions in the from of some third-party open source software libraries listed in chapter 4. Equipment that has been used include one Intel® RealSense™ SR300 and one D415, both provided by SINTEF Ocean AS.

Supervision of scientific work has been provided by John Reidar Mathiassen at SINTEF Ocean AS during the entire course of the study.

Supervisor: Anette Stahl, ITK
Co-supervisor: John Reidar Mathiassen, SINTEF Ocean AS

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Robotic systems play an increasingly important role in modern industry. By performing tasks such as machining, milling or moving objects from point A to point B, industrial robotic systems have proven their use in solving tasks that require high precision motor abilities (Spong et al., 2006). However, there is still a lot of improvement to be done if robots are to reliably perform tasks that seem simple to a human such as inspection of raw materials in food- and agricultural industry, certain house chores or inspection and sorting of miscellaneous objects. These are tasks that seem simple to a human being but may be quite complex in light of today's technology.

The research project named *Neodroid* at SINTEF Ocean AS aims to develop such a robot. The purpose of the Neodroid robot is to be able to perform complex tasks that can be solved by two human arms. In order to solve such tasks, the robot is equipped with two robotic manipulators, each equipped with a depth sensor mounted in eye-in-hand configuration. In addition, a depth sensor with a large field of view mounted in top-down configuration relative to the workspace is used for overview of the scene. For the robot to be able to perform tasks such as the ones mentioned above in environments that are subject to frequent change, the need for robust and accurate perception arises. This study aims to develop software that creates a reconstruction of the workspace volume from a series of depth images captured from different angles as well as determine camera viewpoints from where novel regions are to be viewed.

The task to be solved by the robot is specified in the form of a demonstration by a human in virtual reality (VR). Synthetic examples over the problem domain is then ran-

domly generated and solved in a virtual environment by the machine learning and decision mechanism of the robot. In order for this learning process to be effective, the learning and decision mechanism of the robot has to have access to a three-dimensional (3-D) reconstruction of the workspace volume. This constitutes the main motivation for the development of the custom volumetric reconstruction software in this study i.e. to have full control over how the reconstruction is generated and the ability to alter this procedure.

Another requirement on the vision module is that the workspace reconstruction is to be a voxel-based reconstruction. That is, the volume is represented as a grid of volume elements (voxels) which is the three-dimensonal counterpart of a picture element (pixel). This requirement is stated to maintain compatibility with the decision mechanism of the robot.

Use-cases of the vision module include providing the decision mechanism with a reconstruction of the volume. Additionally, whenever the decision mechanism issues commands to the manipulators, the manipulator controller module needs an accurate representation of the workspace volume in order to generate the security parameters to avoid surface collision.

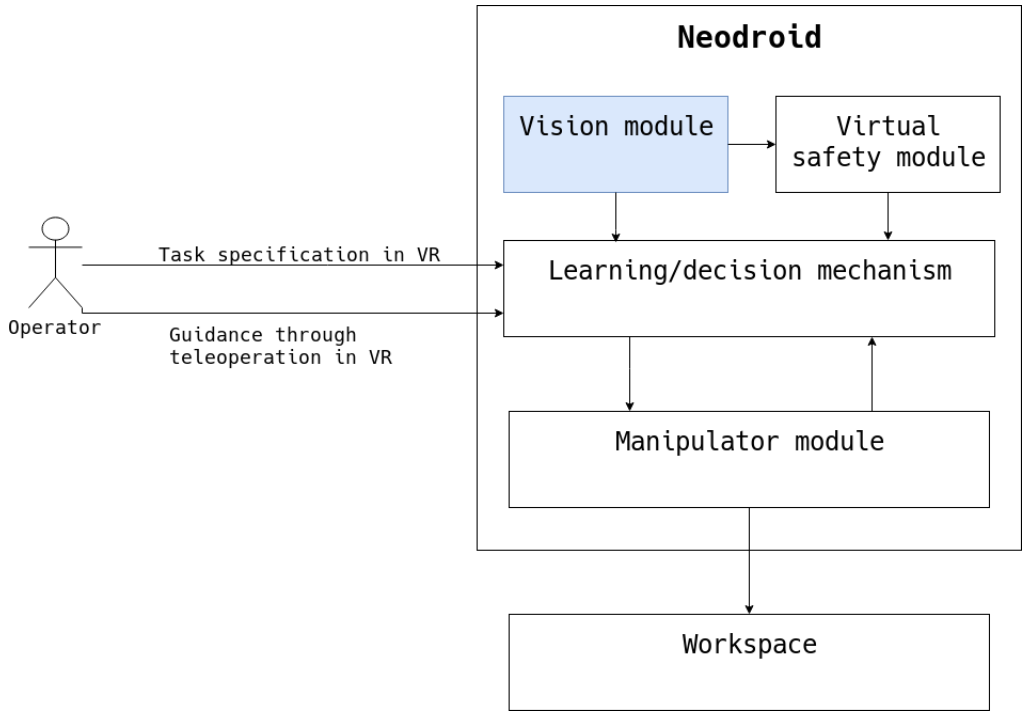

Figure 1.1: Overview of the modules that constitute the Neodroid system. Arrows indicate data flow.

1.2 Contribution

The combination of depth sensors mounted on robotic manipulators has been around for some time. One such system is the ATOS (GOM, 2018) which includes high-precision sensors for measurement of three-dimensional space and is used to automate inspection for quality control in e.g. industrial manufacturing.

A similar application of robotic manipulators used to perform general tasks similar to those a human might perform, is the Moley robotic kitchen (Moley, 2018). This system of robotic manipulators with end-effector grippers emulating human hands, is designed to perform a wide range of cooking tasks while suspended from a rail over the kitchen top. This is a highly complex task that involves problems encountered in design of the Neodroid system such as three-dimensional perception.

Although similar systems to the Neodroid robot have been developed, this particular combination of the system components to perform general tasks have not been successfully implemented to the author's knowledge. What separates the Neodroid from exisiting systems such as those mentioned above lies in the generality. Whereas other systems have successfully been implemented to perform a set of specific tasks, the Neodroid is planned to perform a vast amount of tasks that two human arms can. Additionally, the use of VR to demonstrate tasks that are to be solved autonomously, is a novel aspect of the Neodroid. However, when it comes to the vision system which is the scope of this study, the Neodroid is planned to implement functionality that has already been implemented in a large amount of other systems included the ones mentioned above. These vision-related tasks include three-dimensional reconstruction (Newcombe et al., 2011) and novelty detection (Morris and Angelov, 2014).

Chapter 2

Background theory

In order to aid the understanding of the techniques employed in this study, an overview of relevant theory will be presented. The reader is assumed to have a fundamental understanding of the concepts and notations associated with calculus, linear algebra, and modelling of dynamic systems.

2.1 Mean and variance

In order to understand some of the concepts related to statistical background modelling in a images, some essential formulas in the field of probability theory are introduced. One important attribute to obtain from a collection of data points referred to as a data set, is a measure of how much the data points vary. A measure of variability in a data set consisting of n points is the *population variance* (Walpole et al., 2012). This is the average squared deviation from the mean

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{2.1}$$

where mean is defined as

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.2}$$

The image pixels can be considered as a data set on which variance can be calculated.

2.2 Rigid body kinematics

2.2.1 Rotations

Mathematical representation of orientation is an important task in modelling of dynamic systems. A method for representing three-dimensional orientation of frame a with respect to frame b, is by Euler angles. Euler angles parameterize orientation with the three angles $\Theta_{ba} = [\phi\, \theta\, \psi]^T$ around the x_a, y_a, and z_a axes respectively (Fossen, 2011). In order to rotate vectors in frame a to frame b, the notion of rotation is introduced. Rotation can be represented by a rotation matrix $\mathtt{R}_b^a$ which transforms points in frame b into frame a; $\mathbf{v}^a = \mathtt{R}_b^a \mathbf{v}^b$. Note that the inverse, reverses the rotation source and destination frames $(\mathtt{R}_b^a)^{-1} = (\mathtt{R}_b^a)^T = \mathtt{R}_a^b$ (Egeland and Gravdahl, 2002).

Rotation around x, y and z are expressed by the following rotation matrices respectively

$$\mathtt{R}_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \tag{2.3}$$

$$\mathtt{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.4}$$

$$\mathtt{R}_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

and are referred to as the principal rotation matrices (Fossen, 2011). Composite rotations are performed by simply ordering the rotation matrices in such a way that rotation carried out first is applied first e.g. $\mathbf{v}^d = \mathtt{R}_c^d \mathtt{R}_b^c \mathtt{R}_a^b \mathbf{v}^a$ (Fossen, 2011).

A matrix is a rotation matrix if an only if $\mathtt{R} \in \mathbb{SO}_3$ (Fossen, 2011) where $\mathbb{SO}_3$ is the third order special orthogonal group

$$\mathbb{SO}_3 = \{\mathtt{R} | \mathtt{R} \in \mathbb{R}^{3\times 3}, \mathtt{R} \text{ is orthogonal and } \det \mathtt{R} = 1\}$$

2.2.2 Homogeneous transformations

When reconstructing a three-dimensional space with a collection of two-dimensional depth images, the definition of several coordinate frames is necessary. An essential operation in this study is the transformation of points in one frame into another. This is carried out by multiplication with a homogeneous transformation matrix

$$\mathtt{T}_b^a = \begin{bmatrix} \mathtt{R}_b^a & \mathbf{t}_b^a \\ \mathbf{0}^T & 1 \end{bmatrix} \in \mathbb{SE}_3 \tag{2.6}$$

where $\mathbb{SE}_3$ is the Special Euclidean three-dimensional group (Egeland and Gravdahl, 2002)

$$\mathbb{SE}_3 = \{\mathtt{R}, \mathbf{t} | \mathtt{R} \in \mathbb{SO}_3, \mathbf{t} \in \mathbb{R}^3\} \tag{2.7}$$

$\mathtt{R}_b^a$ is the rotation matrix from frame b to frame a and $\mathbf{t}_b^a$ is the position of the origin of frame b expressed in frame a. A position vector in frame b $\mathbf{p}^b$ is then transformed to a position in frame a $\mathbf{p}^a$ by $\dot{\mathbf{p}}^a = \mathtt{T}_b^a \dot{\mathbf{p}}^b$ where $\dot{\mathbf{p}}$ denotes the homogeneous vector $\dot{\mathbf{p}} = [\mathbf{p}^T | 1]^T$ (Newcombe et al., 2011).

2.2.3 Robotic manipulator modelling

In applications involving robotic manipulators, it is essential to determine the pose of the end-effector relative to some reference frame. Because the links of the robotic manipulator is assumed to be rigidly attached to each other, the pose of any given link can be computed in terms of the poses of the previous links. That is, given the link numbering where link 0 is the base frame and increasing numbers are subsequent links, the pose of link n in the base frame 0 is expressed as

$$\mathtt{T}_n^0 = \mathtt{A}_1(q_1) \cdots \mathtt{A}_n(q_n) \tag{2.8}$$

$\mathtt{A}_i(q_i)$ represents the joint homogeneous transformation matrix of joint i with respect to the previous link, with the joint variable q_i of the link which is

$$q_i = \begin{cases} \theta_i, \text{for revolute joints,} \\ d_i, \text{for prismatic joints} \end{cases} \tag{2.9}$$

θ_i is the rotation around the joint axis of the revolute joint and d_i is the translation distance along the joint axis of the prismatic joint (Spong et al., 2006).

2.3 Camera model

The measurements used in this study are expected to be in the form of a depth image. The depth to each image pixel $\mathbf{u} \in \mathbb{N}_0^2$ is defined as the distance along the $\mathbf{z}_k$ axis and is encoded in each image pixel $D(\mathbf{u})$.
In order to transform a camera pixel together with it's corresponding depth measurement to a three-dimensional point $\mathbf{p}_k$ in the camera frame, the camera matrix of intrinsic parameters also referred to as the camera calibration matrix is used.

$$\mathbf{K} = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

(Hartley and Zisserman, 2004) The intrinsic parameters are sensor-specific parameters and the meaning of these can be better understood when explained in the context of the camera pinhole model (Gonzalez and Woods, 2010). Consider the camera pinhole model in figure 2.1. Light is captured on the film on the image plane through the pinhole of the camera.

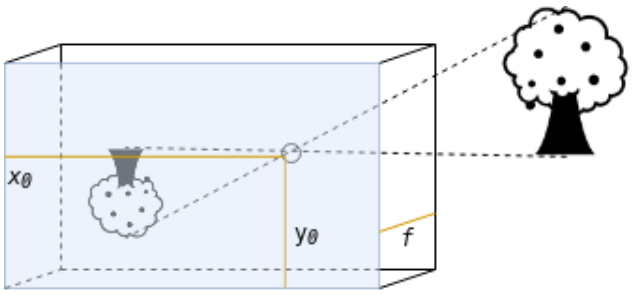

Figure 2.1: Camera pinhole model

The focal length f is the distance from the image plane to the pinhole and is represented by the intrinsic parameters f_x and f_y measured in pixels. In a true pinhole camera $f_x = f_y$ is true. However, due to factors such as flaws in the sensor or lens distortion effects these may differ.

The skew coefficient s represents the skew between the image plane x and y axis. Lastly, the principal point (x_0, y_0) represent the xy position of the pinhole in the camera frame.

A an image pixel $\mathbf{u}$ together with it's corresponding depth measurement $D(\mathbf{u})$ is then transformed to a point in the camera coordinate frame by

$$\mathbf{p}_k = D(\mathbf{u})\mathbf{K}^{-1}\dot{\mathbf{u}} \tag{2.11}$$

2.3.1 Extrinsic camera calibration

Extrinsic camera calibration refers to the task of determining the camera's extrinsic parameters, namely the pose (position and orientation) relative to the global coordinate frame g denoted as the homogeneous tranformation matrix $\mathtt{T}_k^g$. In spite of the camera being mounted in eye-in-hand configuration the pose of the camera is *not* the same as the pose of the end-effector due to offset of the mounting point of the camera from the end-effector position. Since the links in the robotic manipulator are considered to be rigid, the camera pose relative to the end-effector $\mathtt{T}_k^e$ can be used to calculate the pose of the camera relative to the global frame. This can be calculated as the composite homogeneous transformation matrix

$$\mathtt{T}_k^g = \mathtt{T}_e^g \mathtt{T}_k^e \tag{2.12}$$

where $\mathtt{T}_e^g$ is the pose of the end-effector of the robotic manipulator which is obtained from the manipulator controller. So the task of performing extrinsic camera calibration thus becomes the task of estimating the camera pose relative to the manipulator end-effector $\mathtt{T}_k^e$. A detailed description of this task is beyond the scope of this report, but further information may be found in Tsai and Lenz (1989).

Chapter 3

Method

3.1 Overview

In this chapter, a detailed description of the methods employed to solve the task at hand will be given. The description given in this chapter is concerned with the conceptual approach to the solution of the problem, while implementation details are reserved for the subsequent chapter. Essentially, this means that the discussion of different available methods for solving sub-problems or mathematical analysis of these - where necessary - will be carried out in this chapter.

To begin with, the planned hardware setup of the application discussed in this study will be described. Then, a closer look will be taken at different options for how 3-D scene measurement can be carried out as well as sensors available to realize these. Furthermore, methods for representing the 3-D scene and how new measurements are to be integrated into this reconstruction will be presented. Novelty detection being one of the core topics in this study will be discussed extensively in the subsequent sections with the most promising alternatives for performing this being evaluated. Another core topic of this study is the grouping of novel regions in the 3-D reconstruction of novelties. This motivates the introduction of cluster analysis in this study and thus several appropriate methods for performing this analysis will be covered.

3.2 Measurement

3.2.1 Hardware setup

In order to better understand the context of this study, an overview of the hardware setup will be given in this section. This is to aid the understanding of the Neodroid application itself as well as the options available for solving several of the sub-problems encountered in this study.

The hardware setup of the Neodroid application consists most fundamentally of two robotic manipulators, each mounted with a 3-D scene measurement device in eye-in-hand configuration with small field-of-view. Eye-in-hand configuration refers to vision sensors mounted at the end of the robotic manipulator close to the manipulator end-effector (Lippiello et al., 2005). This is to allow the robot to non-intrusively view objects from different angles since the alternative involves to manipulate the objects themselves. The base of the robotic manipulators are mounted outside of the $1\,\mathrm{m} \times 1\,\mathrm{m} \times 1\,\mathrm{m}$ workspace volume on either side as shown in the figure 3.1

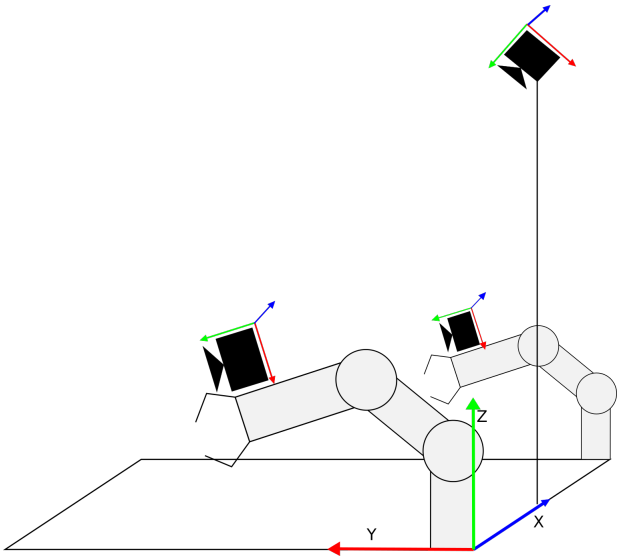

Figure 3.1: Hardware setup of the Neodroid consisting of two robotic manipulators with depth sensors eye-in-hand and an overview sensor.

The manipulators are of the type Franka Emika and are shipped with an open source API (Application Programming Interface) to control these. The task of operating the manipulators are beyond the scope of this study, but the reader may consult the vendor website [1] for more information.

[1] `https://franka.de/`

The robot is also equipped with what will be referred to as an overview 3-D scene measurement device with a large field-of-view which is intended to be mounted at a fixed location above the workspace volume. This is - as the name implies - to provide the robot with an overview of the contents in workspace volume. The device is to be mounted in a top-down fashion allowing for maximum awareness in the scene as objects are less likely to be occluded by other parts of the scene as for example is the case if it was mounted to view the volume from the side. Note that this does not restrict the overview camera to a 90° viewing angle downwards from the horizontal plane.

3.2.2 Scene measurement

Central to the task of reconstructing a 3-D scene is the task of measuring 3-D space. Measurement here refers to the acquisition of information on occupancy of the volume as well as color and texture of objects in the scene. Vital to the task at hand is the acquisition of occupancy information of a fixed-sized volume and therefore appropriate measurement techniques needs to be determined.

There are several approaches available to obtain depth measurements form a 3-D scene (including ultrasound, RADAR, LIDAR, etc.) but this study calls for these to be commercially available and mountable on a robotic manipulator. Even with the set of relevant sensors limited in this way, there are still several viable techniques available to obtain the needed information from a scene.

Color-sensitive cameras have been around for a long time, but a single such sensor alone does not produce any information on the depth to the pixels in the frame. It is only when multiple such sensors are combined that information on the depth to each pixel is made available by doing stereo-correspondence (Lazaros et al., 2008). This is much like the human vision system works, where the two eyes act as individual sensors and the brain performs the stereo-correspondence (Marr and Poggio, 1979). This is referred to as stereo vision and is one of several approaches available to obtain depth measurement of a 3-D scene.

The introduction of Kinect by Microsoft Corporation in late 2010, signalled the dawn of a new age of commercially available depth sensing devices (Han et al., 2013). By commercially available it is meant that the price is bearable on an average private con-

sumer budget. These sensors employ the infrared (IR) spectrum to project a know pattern known as structured light (Scharstein and Szeliski, 2003), onto the scene directly in front of the sensor with an IR projector. The depth to the scene is determined by interpreting the reflection of the projected pattern and the range to each pixel is encoded as a scalar value at the pixel location in the image frame. This eliminates the necessity of multiple color cameras to determine the distance from the sensor and thus provide a more compact solution to the scene measurement problem.

One drawback with using only pure range sensors is that color measurement of the scene is absent. Consequently, perception of a vast amount of textures that have a weak signature in the IR spectrum are unobservable.

The importance of having a perceptive ability that includes color in this application can be questioned. This doubt is raised as a result of the assumption that the ability to percieve the relative location of surfaces in the scene takes presedence over the ability to percieve color and texture. The reasoning behind this is that the vital piece of information needed for the robotic manipulators to avoid collision with the scene and being able to grip the intended objects, is the location of the scene surface.

Given the above discussion on different sensor types, this study proceeds to utilize the 3-D scene measurement technique involving active range sensors. These sensors are available in a variety of differing sizes suitable for the Neodroid application. Some of the devices containing such sensors are even produced with a color camera in the same device which allows for doing stereo-correspondence with the IR image frame to obtain the corresponding color information to each pixel in the IR frame.

3.2.3 Sensors

With the decision made to use active range sensors, there is still a variety different sensors on the market. The most relevant attributes in a camera product for use in this application are size, pricing, quality, and ease of use. The device suitable for use in this application is found by balancing these four attributes in such a way that the size allows for mounting on a robotic manipulator, the price of the device is as low as possible to allow for reasonably good depth measurements, and that the API provides a set of low-level basic functionality related to image acquisition in different streams to reduce setup time.

Some relevant sensors on the market at the time of this study include

- **Stereolabs ZED**[2] is a stereo depth camera with high resolution, wide field of view and integrated motion tracking. It comes with wide support for third-party software and is priced at around $450 .

- **Intel RealSense depth cameras**[3] are a series of modern, easy to use depth cameras that come with a multi-platform SDK and are sold at a price of around $140. The Intel RealSense SDK include a wide range of tools related to sensor control and image processing.

- **Zivid**[4] is a high-end, high precision stereo depth and color camera that focuses on quality of measurement and provides an easy and intuitive API for major vision programming tasks. The price is estimated to be at least one order of magnitude higher than the two cameras mentioned above.

The sensors used in this study belong to a class of active range sensors produced by Intel® Corporation, namely the Intel® RealSense™ SR300 and D415 models. These are chosen based on how they trade-off the attributes mentioned above; size, pricing, quality, and ease of use. Even though there are sensors on the market that provide higher quality measurements, the Intel® RealSense™ cameras are at their most attractive in their quality-to-price ratio.

On the Neodroid, two SR300 will be used, one mounted on each arm. This is to provide a high-resolution reconstruction of regions of interest (ROI). By moving closer and thus covering a smaller surface area, the surface area to number-of-pixels ratio is maximized, and the resolution of the reconstruction is maximized.

3.2.4 Intel RealSense API

The Intel® RealSense™ API known as librealsense, is a collection of software that is used to interact with the camera sensors in the D400 series and the SR300 model. It is written in the C++ programming language but provides wrapper functions and classes for Python, C, .NET languages, Node.js and integration with third-party software such as MATALB,

[2] `https://www.stereolabs.com/`

[3] `https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html`

[4] `https://zividlabs.com`

Point Cloud Library, OpenNI, OpenCV, ROS, LabVIEW, and Unity.

Essential functionality in the Intel RealSense API include the extraction of scene measurement in the for of an image frame from different streams available in the device. The streams available in both of the camera models used in this study include infrared (IR), depth, and color. The depth stream refers to a stream of IR images that have been processed internally by the device to provide frames that have the depth to the nearest object along the axis perpendicular to the camera face encoded into each pixel (as shown in figure 3.1). This differs from the IR frame in the sense that only the intensity in the IR spectrum is encoded into each pixel in the IR frame.

Caution is advised when using both the IR sensor and the RGB color sensor. Given the same resolution in the IR frame and the depth frame, a pixel in either frame overlaps with the same 3-D scene position as the other since both frames are captured with the same sensor. However, with the color frame, this pixel correspondence is not the same since the RGB color frame is not obtained with the same sensor and thus there is an offset between the IR sensor and the RGB color sensor in the device.

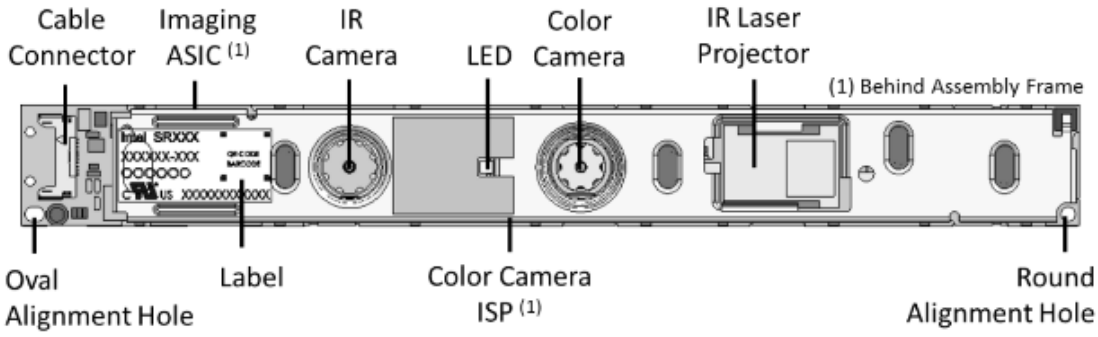

Figure 3.2: Intel® RealSense™ SR300 component layout (front view) (Intel®Corporation, 2016)

3.3 Workspace reconstruction

As previously mentioned, the workspace volume is expected by other parts of the system to be represented in terms of a voxels indicating occupancy. Given this information, two representation alternatives are chosen for further investigation, namely the regular 3-D grid of voxels and the octree representation.

3.3.1 Regular 3-D voxel grid

The regular 3-D voxel grid is best described by the illustration shown in figure 3.3.

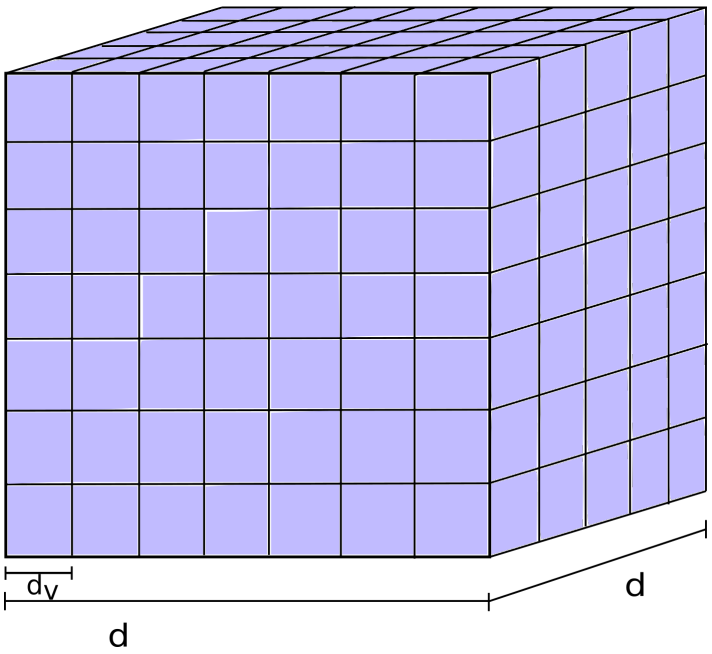

Figure 3.3: Regular 3-D voxel grid

The computer representation of this data structure is in the form of a 3-D (triply subscripted) space array of indivisible unit cubes (voxels). The edges of each cube are equal in size, and all cubes within the grid are of equal size and are referred to as the cube diameter d_v. Each voxel contains information indicating occupancy in the form of a binary flag. The defining properties of this representation data structure is the number of voxels per any one side of the grid n and the resolution, defined as the diameter d_v of the voxels.

The main attriactive quality with this representation method of 3-D space is simplicity. It is extremely easy to index into such a data structure to obtain the voxel containing any given point within the boundary of the workspace. The position of the voxel is inferred from its index within the grid $\mathbf{p} = d_v \cdot [i, j, k]^T$ where i, j, k are non-negative integers $0 \leq i, j, k < n$.

However, with a total of n^3 number of voxels within a grid, the regular 3-D grid of voxels consumes a rather large amount of memory. This is because the grid creates an equally high-resolution representation of the entire volume regardless of the contents. Given that most likely the volume to be reconstructed will contain large amounts (70-

90%) empty space (transparent voxels) (Lacroute and Levoy, 1994), this can lead to sub-optimal usage of memory. Therefore, techniques to reduce the memory footprint of a space array representation will be investigated.

3.3.2 Oct-Trees as a 3-D modelling technique

Proposed by Jackins and Tanimoto (1980) is an alternative modeling approach for three-dimensional objects called *oct-tree* (commonly referred to as *octree*). This approach aims to reduce the memory space footprint of the regular voxel grid data structure and operations performed on these. The proposed method for representing this data structure as octrees consists of partitioning the volume into eight equally sized octants. Much like the voxels in the 3-D grid, each of these octants holds a label indicating the content of voxels in the octant:

- `FULL` indicating that all unit cubes contained within the octant are occupied.
- `VOID` indicating that all unit cubes contained within the octant are empty.
- `MIXED` indicating that the octant contains unit cubes that are labelled `FULL` and `VOID`

The octree is a tree structure where the root node represents the entire volume which is a cube of diameter d. This node may be divided into 8 octants, each with diameter $d/2$. The subdivision process may be continued successively for each octant until either each octant is a unit cube, or all unit cubes within an octant have the same label (either FULL or VOID). The result of such a process is illustrated in figure 3.4.

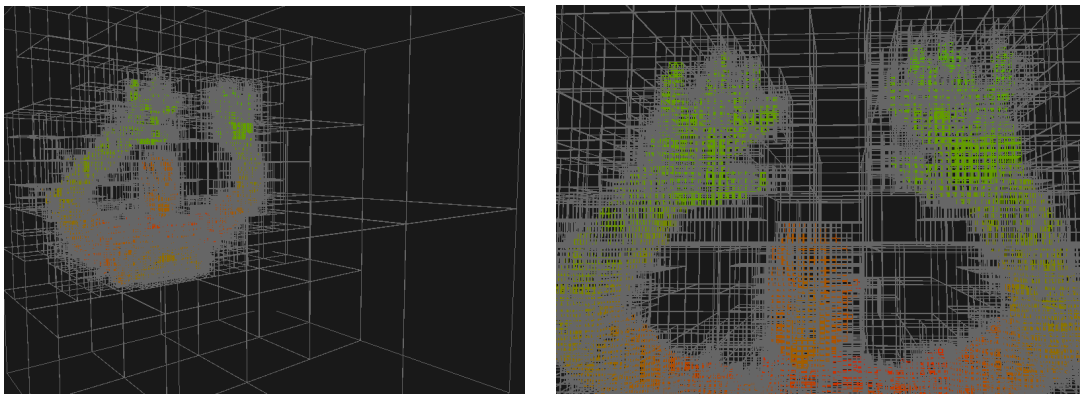

Figure 3.4: 3-D scene representation as octree. Note that ”free space” regions are represented with larger cubes thus limiting the number of leaf nodes.

The root is said to be at level 0 and the unit cubes at level n. To generalize the notion of levels in the octree a node is said to be at level i if it contains $2^i \times 2^i \times 2^i$ unit cubes (Jackins and Tanimoto, 1980).

The voxels in the octree are indexed by a 3-D vector of non-negative integers $\iota = [i, j, k]$. The elements of ι represent the number of unit cubes along each dimension of the reconstruction, exactly similar to how indexing is done in the regular voxel grid. Since the octree creates leaf nodes (unit cubes) only if space is occupied, there may not exist a leaf node at the given index, and thus a null value is returned. Further implementation details are described in chapter 4.

One useful operation on the octree to reduce the memory space usage, is the `Condense()` operation. This procedure traverses the octree and removes all cases where eight siblings have identical labels and sets the label of the parent equal to the child labels.

The octree data structure trades off memory consumption with complexity. That is, the octree may produce less nodes to represent the same scene compared to a regular 3-D voxel grid, at the cost of keeping track of the relation between the nodes in the tree. Experiments were carried out to determine the impact of the more complex octree structure versus the regular 3-D voxel grid on memory (see chapter 5). Another drawback with the octree is that the resolution depends on the volume diameter and the number of levels in the tree

$$d_v = \frac{d}{2^n}$$

The fact that the number of levels n is required to be a non-negative integer, results in the user not necessarily being able to realize the desired combination of resolution d_v and volume diameter d. Being able to specify the exact resolution of the reconstruction is an essential feature of the reconstruction method as this is needed by the decision mechanism of the robot and thus the regular 3-D voxel grid is selected as the representation method.

3.4 Integrating new measurements

After obtaining a way to represent the reconstruction of the workspace volume, the next task that presents itself is integrating new measurements from different viewpoints into this reconstruction.

3.4.1 Direct depth map projection

The simplest way to integrate a scene measurement in the form of a depth image into the reconstruction is by directly transforming image points to voxel indices, and marking these as occupied. This is done by using the location of the image pixel in the image frame $\mathbf{u}$ together with the depth in meters encoded into the image at this location $D_i(\mathbf{u})$. The corresponding camera frame position of this point is expressed as $\mathbf{p}^k = D_i(\mathbf{u})\mathbf{K}^{-1}\dot{\mathbf{u}}$. The point $\mathbf{p}^k$ is then transformed to the global coordinate frame g by multiplication with the homogeneous transformation matrix $\mathtt{T}^g_{k,i}$ which represents the camera pose for the i-th depth observation $\dot{\mathbf{p}^g} = \mathtt{T}^g_{k,i}\dot{\mathbf{p}}^k$. The point $\mathbf{p}^g$ is then used to generate a 3-D space array index $\iota \in \mathbb{N}^3_0$ that corresponds to the voxel in which the depth map observation is contained. ι is computed by simply dividing $\mathbf{p}^g$ by the length of the unit cube edges d_v and flooring the result $\iota = \lfloor \mathbf{p}^g / d_v \rfloor$. ι is then used to index into the representation and mark the voxel as occupied. If $\mathbf{p}^g$ is not a point within the bounds of the volume, the observation is dropped.

Note that this method implements no noise reduction as measurements are directly projected onto the 3-D representation. Therefore, the direct projection approach to measurement integration can be improved by applying a filtering of voxels based on the number of observations. This is done simply by letting each voxel in the reconstruction hold a counter that is incremented each time a surface is observed within it.

3.4.2 Truncated Signed Distance Function

Another method for measurement integration is based on the truncated signed distance function (TSDF). An analysis of this approach is presented by Werner et al. (2014) which provides an overview of how the TSDF can be used to extract object surfaces from a volumetric scene, and how multiple depth images from different viewpoints can be integrated into one single reconstruction. This is done through the TSDF update scheme

$$TSDF_i(\mathbf{x}) = \frac{W_{i-1}(\mathbf{x})TSDF_{i-1}(\mathbf{x}) + w_i(\mathbf{x})tsdf_i(\mathbf{x})}{W_{i-1}(\mathbf{x}) + w_i(\mathbf{x})} \tag{3.1}$$

$$W_i(\mathbf{x}) = W_{i-1}(\mathbf{x}) + w_i(\mathbf{x}) \tag{3.2}$$

for all voxel positions $\mathbf{x}$ in the reconstruction where $TSDF_i$ and W_i denote the total TSDF update and weight after observation i respectively. $tsdf_i$ and w_i denote the TSDF

value and weight for the new observation. Furthermore,

$$sdf_i(\mathbf{x}) = d_s - d_x \tag{3.3}$$

$$tsdf_i(\mathbf{x}) = max(-1, min(1, \frac{sdf_i(\mathbf{x})}{t})) \tag{3.4}$$

$$w_i(\mathbf{x}) = 1 \tag{3.5}$$

where d_s and d_x denote the depth along the $\mathbf{z}_k$ axis to the surface point and the voxel $\mathbf{x}$ respectively.

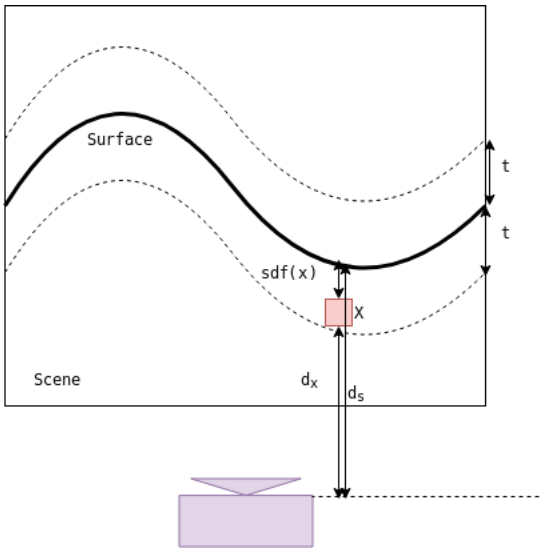

Figure 3.5: The signed distance to voxel x and truncation distance relative to the surface

To estimate the surface of the scene, the zero-level set of the reconstruction is extracted. This consists of searching for voxels in the reconstruction that have a TSDF value that evaluates to zero. This surface estimation procedure also acts as noise reduction. Since the TSDF value of a voxel may be the weighted average of several measurements, the effect of bad measurements may be reduced with increasing number of depth measurements.

3.5 Novelty Detection

To be able to keep an accurate reconstruction of the workspace volume, even after changes have occurred, the system is required to update the current reconstruction in some way. When we humans intend to further visually inspect an object of interest, we might want to get closer to the object and/or view it from a different vantage point or orientation. Our muscular system makes this possible by movement of the neck, our entire body, or

by manipulating the object with our limbs e.g. picking it up or turning it over so that different sides of the object is exposed to our vision system i.e. our eyes. For a humanoid robot such as the Neodroid, most of these motions are in violation of the premise stating that the robot is stationary. In addition, movement of the overview camera sensor acting as the Neodroid's eyes, is eliminated. This translates to neck motion in humans. Continuing on the human analogy, the only remaining options in order to generate a different vantage point of an object of interest, is to use the robotic manipulators to grip the object and bring it closer to the robot's overview camera. This option has several drawbacks. It limits the objects of which new vantage points can successfully be realized to those that can be picked up by the manipulators. This has potential to severely impair the robot's awareness within it's own workspace volume. Consider the scenario when a large object is placed in front of several other smaller objects as seen from the camera. The smaller objects will stay occluded since the robot is unable to realize new vantage points. This is the main motivation behind the mounting of a camera on each of the robotic manipulators in what is known as eye-in-hand configuration. This enables the Neodroid to move it's eyes, allowing for more adaptive awareness schemes in the workspace volume. The task of staying aware of the contents of the workspace thus becomes finding out where to move the eyes. Here is where novelty detection becomes relevant. Objects that are of interest to the vision system are objects that are not already incorporated into the workspace reconstruction i.e. novel objects. If the system can detect where novel objects are, it can integrate them into the workspace reconstruction by moving the eyes-in-hand to scan these regions of interest (ROI), instead of scanning the entire workspace. It is desirable that both placing new objects in the workspace as well as removing them should require a close-up scan by the eye-in-hand cameras of the region where an object was placed or removed. This is because other objects may be covered/uncovered upon placement/removal of objects. By updating the reconstruction from a vantage point that is closer to some surface than the overview camera, a more detailed perception of that surface may be obtained. The reconstruction of the scene is in the form of a regular 3-D voxel grid where each voxel has a binary label indicating whether or not that voxel is occupied by some solid. The voxel grid and voxels are constant in size and so are the images and pixels used to measure the scene. Due to perspective distortion, the closer to a surface an image is captured, the fewer voxels are overlapped by each image pixel. Therefore, the need for an accurate novelty detection scheme arises. By having a good bearing on where the novel sub-volumes within the workspace are, new vantage points may be realized by the eye-in-hand cameras.

3.5.1 Available streams

For the sake of simplicity, novelty detection is done using the overview camera by avoiding the more complex task of doing novelty detection on image streams from cameras that are not stationary i.e. the eye-in-hand cameras.

From the overview camera used in this study, the Intel® RealSense™ D415, there are three available streams, namely the infrared stream, the depth stream, and the RGB color stream. For reasons previously discussed, the need for depth information from each image frame is decided to be of grater importance than color features. Therefore the relevant streams for novelty detection are reduced to the depth stream and the infrared stream, since these streams are obtained from the same sensor within the device, while the color sensor is in a separate sensor as shown in figure 3.2. An example of how performing novelty detection in the IR frame is not desirable is shown in figure 3.6.

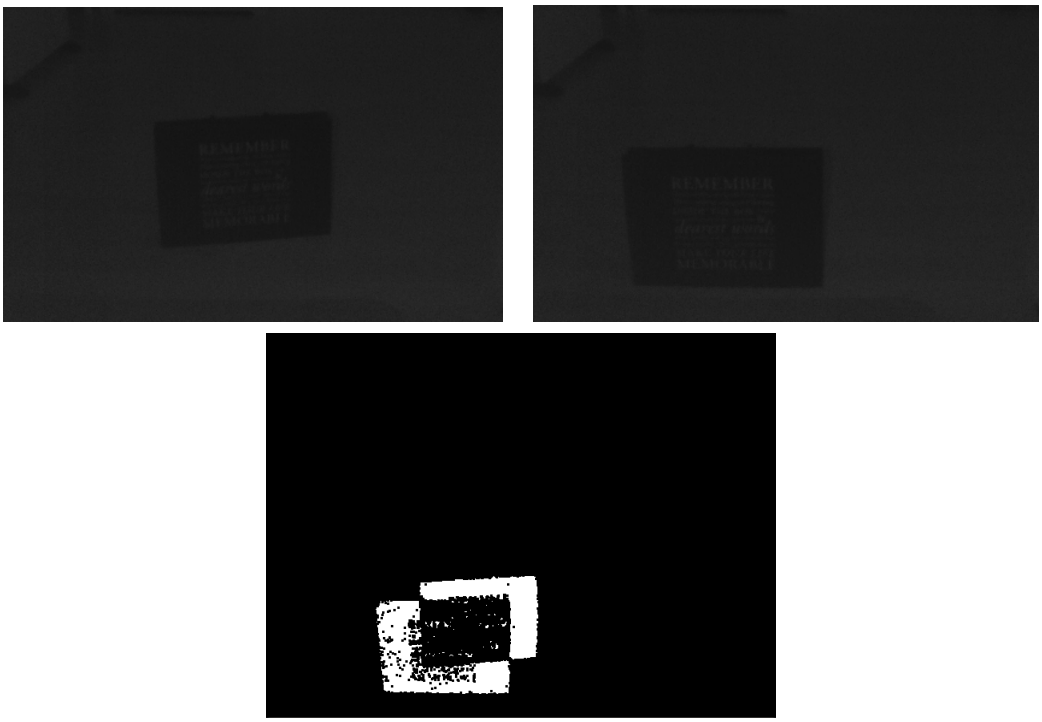

Figure 3.6: Novelty detection on a stream of IR images. In spite of the distance from the surface to the camera sensor being different in the two frames, there is not a considerable change in the IR intensity values to flag the entire surface as novelty. This results in important information loss.

3.5.2 Technique overview

Since novelty detection will be done on the image streams obtained from the overview camera, probabilistic novelty detection schemes using background subtraction can be used since the background is stationary. Such techniques have shown great promise and are suitable under the conditions of this application as recorded by Sobral and Vacavant (2014).

These techniques are used to extract the foreground containing objects of interest in images by subtracting the background. In order to do this, a decision has to be made on what is background and what is foreground. The way this is done is by keeping a model of the background (Sobral and Vacavant, 2014). For every new frame, the background model is subtracted to extract the foreground. The background model is then updated with the new frame to adapt to changes in the image.

Recursive Density Estimation

Probabilistic novelty detection using background subtraction methods are used in this study based on the widespread success that has been recorded with such methods. One such approach addresses the need for real-time novelty detection with constant memory footprint and is known as Recursive Density Estimation (RDE) (Morris and Angelov, 2014). RDE estimates the probability density for each image pixel recursively. That is, each new frame is incorporated into the background model before it is discarded. The background model is represented as an image of probability densities. The intensity in each pixel is computed pixel-wise based on previous frames and thus RDE is a mono-modal technique (Morris and Angelov, 2014). The densities for each pixel are computed by recursively updating the mean and the average squared intensity of each data sample for every frame, which raises the need for a recursive update scheme.

Formula (2.2) can be adapted to fit a recursive update of the new mean when a new sample arrives by writing

$$\mu_k = \frac{k-1}{k}\mu_{k-1} + \frac{1}{k}x_k \tag{3.6}$$

$$\mu_1 = x_1 \tag{3.7}$$

where x_k is the image pixel intensity at frame k.

For the average squared pixel intensities

$$s_k = \frac{k-1}{k} s_{k-1} + \frac{1}{k} x_k^2 \tag{3.8}$$

$$s_1 = x_1^2 \tag{3.9}$$

These parameters can be used to estimate the probability density of the pixel with the Cauchy type kernel function

$$D = \frac{1}{1 + \|x_k - \mu_k\| + s_k^2 - \|\mu_k\|^2} \tag{3.10}$$

$$\tag{3.11}$$

The reason for using the Cauchy type kernel is described by Morris and Angelov (2014) as to provide a probability density estimation scheme that does not make any assumption on the probability distribution of the pixel intensities. This allows for more flexibility in what types of footage that may be used. When there is no change in density between the background model and the current frame, the pixel is considered to belong to the background. If the difference is larger than some threshold the pixel is considered to belong to the foreground.

if $\|D_k - D_{k-1}\| > c\sigma_k$ **then**
  pixel is foreground;
else
  pixel is background;
end

where c is a user specified constant.

The threshold proposed by Morris and Angelov (2014) is chosen to be a multiple of the standard deviation of all previously recorded frames, usually $2\sigma_k$ or $3\sigma_k$. A higher multiple of the standard deviation results in a loss of sensitivity to change, whereas a smaller threshold results in increased sensitivity.

One of the major strengths of the RDE novelty detection approach is that it does not need to keep any of the images in the stream in memory after the image has been processed. This is tremendously advantageous for applications where memory is scarce and for applications using large datasets e.g. surveillance videos. Frames can thus be processed on-the-fly and discarded. The RDE algorithm requires relatively little computation

with its only pixel-wise calculation being the recursive background model update and the Cauchy kernel evaluation. Since the image traversal is a task that is highly parellelizable, real-time implementations are well within reach without quality-sacrificing optimizations. This allows for implementation on hardware suitable for unmanned vehicles or other embedded applications. Additionally, RDE is a pixel-wise novelty detection scheme and thus it is able to adapt certain regions subject to persistent change, by adjusting the pixel-wise probability of novelty in the given region. Experiments have been performed in this study to illustrate this (see section 5.5). This differs from the more simplistic difference image detection where the previous frame in the stream is subtracted from the current frame in order to detect novelties. Other benefits of the RDE approach is that it is a fully autonomous technique that does not require any user input which can prove to be useful when used on a large data set.

Aspects of the RDE algorithm that are not desirable include luminescence having a very large impact on the output. Areas of an image that are subject to change in lighting conditions are picked up as novelties. This makes RDE error-prone when used in applications where sudden changes in lighting conditions may occur.

In order to investigate the effects of different settings of the novelty detection scheme, a series of modifications are introduced. This is done to find out if other options may perform better in the given application. The first modification introduced is to change the kernel function to estimate the pixel-wise intensity probability. On advice from the co-supervisor of this study, the assumption that the intensities are approximately distributed according to the Gaussian distribution. The reasoning behind this assumption lies in the central limit theorem stating that the normalized sum of independent random variables tend towards a Gaussian distribution even if the random variables themselves are not Gaussian (DasGupta, 2010). This leads to replacement of the formula (3.10) with the Gaussian function

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma}} \tag{3.12}$$

In order to enable a larger degree of freedom when it comes to tuning of the novelty detection scheme, the ability to adjust the learning rate of the background model is introduced. That is, adjusting how fast new raw scene measurements in the form of IR or depth stream images are integrated into the mean and variance estimation used in RDE. This is done using a scalar weight $\alpha \in [0, 1]$ that is introduced into the background model update scheme in the following way

$$\mu_k = (1-\alpha)\mu_{k-1} + \alpha x_k \tag{3.13}$$

$$\sigma_k^2 = (1-\alpha)(s_{k-1} - \mu_{k-1}) + \alpha(x_k^2 - x_k) \tag{3.14}$$

The new thresholding scheme to determine if pixels are background or foreground was modified to

if $f(x_k|\mu_k, \sigma_k)/f(\mu_k|\mu_k, \sigma_k) > c$ **then**
  pixel is foreground;
else
  pixel is background;
end

The division by $f(\mu_k|\mu_k, \sigma_k)$ is done for the purpose of normalization relative to σ_k.

3.6 Post-processing and novelty segmentation

Given the results of the novelty detection stage in the form a binary image seen in figures 5.9 and 5.10, it is seen that the presence of noise is significant. Integrating this directly into a 3-D model of the robot's workspace results in the appearance of novelties in areas where there in reality are none. Therefore, some form of processing of the output of the novelty detection needs to be applied if the results are not to be rendered completely useless.

One approach to reduce these noise artifacts may be the simple approach deduced from intuition involving finding the contour around the of the artifacts an then thresholding on the areas of the contours. That is, discaring all contours that have an area less than some given threshold defined by the user.

Another available approach to use the morphological operation erosion (Haralick et al., 1987).

3.7 3-D reconstruction of novelties

As previously mentioned, the goal of this study is to determine what sub-volumes of the workspace need further inspection in order to generate an accurate reconstruction of the entire volume. The logical next step after obtaining novel regions in the image plane is to convert these 2-D regions to 3-D regions before integrating them into the workspace

reconstruction. In this section, a closer look will be taken at the task of converting 2-D into 3-D regions.

In order to convert 2-D regions into 3-D, a pixel-wise transformation can be performed on all pixels contained in the region. This pixel-wise transformation takes as input the 2-D image space location of the pixel $\mathbf{u}$ together with the depth in meters encoded in that pixel $D(\mathbf{u})$ and outputs the 3-D position of the corresponding point in the camera frame as shown in equation (2.11).

The depth encoded in each pixel is represented as an integer value in the image frames obtained through the librealsense API. This value can be scaled to a metric value through a sensor-specific scaling factor determined by the intrinsic parameters of the sensor.

Pixels that do not represent any surface observation e.g. the surface is positioned closer to the sensor than the near clipping plane or further away than the far clipping plane, are represented as $D(\mathbf{u}) = 0$. This gives rise to a problem whenever two-dimensional regions are to be transfered to their three-dimensional representation. Since the novelties in the scene are represented as white regions in a binary black-and-white image, there may not always be a non-zero depth value encoded in the corresponding pixel in the depth frame. To solve this problem, novel pixels that have a pixel depth $D(\mathbf{u}) = 0$ are simply omitted by not calculating the corresponding three-dimensional points.

The points obtained from the transformation described in (2.11) are defined in the overview camera frame. These points are transferred to points in the global frame by a multiplication with the homogeneous transformation matrix

$$\dot{\mathbf{p}}^g = \mathrm{T}_k^g \dot{\mathbf{p}}^k \tag{3.15}$$

3.7.1 Determining three-dimensional regions

The desire to identify separate novel regions and create bounding boxes around them, warrants some form of detection of where these regions are. The options for performing detection of novel regions include doing so both before and after integrating the novel regions into the 3-D reconstruction of the workspace. However, since the regions are to be expressed as bounding boxes around novel sub-volumes of the workspace, techniques performing this detection in the 3-D workspace reconstruction are considered more suitable.

One such approach include the use of cluster analysis (clustering), referring to the task of grouping similar objects based on some attribute (Willett, 1988). In the context of the task at hand, clustering in this application is done based on the positions of all points in the reconstruction that represents a novel surface observation. Two common algorithms for performing clustering on a data set include agglomerative hierarchical clustering (AHC) and k-means (Steinbach et al., 2000).

Agglomerative hierarchical clustering refers to a clustering approach which is initialized with each data point in the data set constituting its own cluster. Clusters are then merged into one another based on similarity of some chosen attribute. While the merging of clusters is done, a log is kept of every merge in the form of a tree data structure referred to as *dendrogram*. An example dendrogram of the agglomerative clustering of a data set of two-dimensional data objects is shown in 3.7. First, the data points p_3 is merged with p_5 to form the first cluster C_1. Then, p_1 is merged with p_2 to form cluster C_2. C_2 and the point p_4 are merged into C_3 before merging the entire data set into a single cluster C_4.

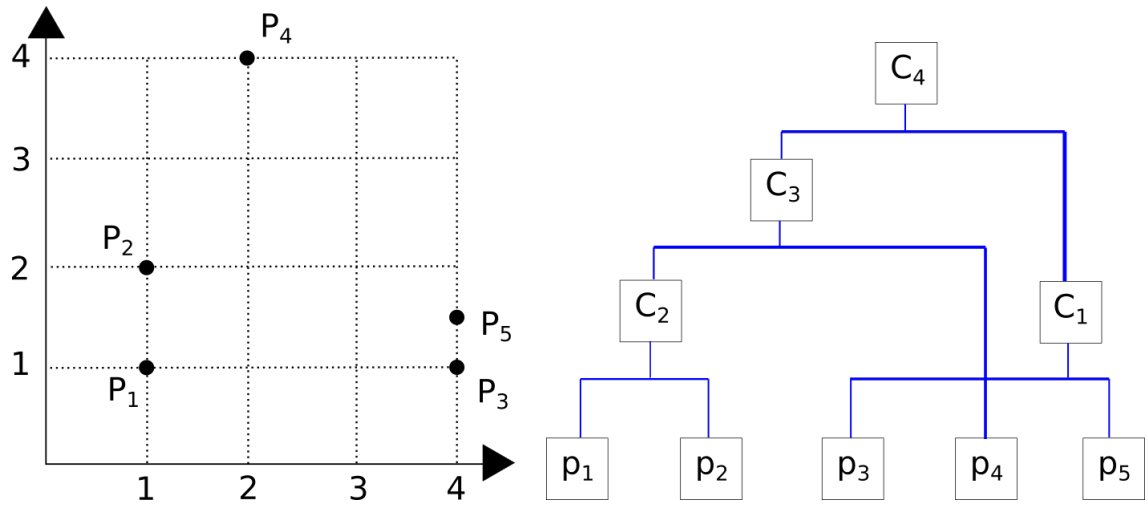

Figure 3.7: Clustering example. AHC produces the output dendrogram (left) from the input data set (right)

The algorithm terminates when there is only one cluster (Steinbach et al., 2000).

The k-means clustering algorithm on the other hand, takes as input the data set as well as the desired number of clusters in which the set is to be divided. The algorithm then proceeds to assign data points to the cluster in which the mean of the data points is closest.

A clustering technique found to be suitable in this study is agglomerative hierarchical clustering. This choice is made based on the the fact that it is one of the main clustering approaches (Steinbach et al., 2000) while maintaining simplicity.

Determining the number of clusters

A problem that arises with either of these clustering algorithms is that the "optimal" number of clusters - referring to the cluster configuration that is as close as possible to the grouping of the original data set (Halkidi and Vazirgiannis, 2001)- is not an output. This is obvious for the k-means algorithm since it takes k number of desired clusters as an input. For the AHC algorithm, the dendrogram produced allows the user to select the l top clusters for further analysis, requiring the user to have a level of a priori insight into the data set that may not be available.

One way to determine the number of clusters in a dataset could be to obtain a measure of how spread out or varied the data points are from the centroid. A common way to measure this in statistical analysis is with the use of population variance (2.1). The intention is that the variance of the data set around its centroid can be used to slice the dendrogram returned by AHC to obtain the l top clusters. To perform this estimation of number of clusters the centroid and variance of the input dataset is computed as

$$\mathbf{p}_c = \frac{1}{n}\sum_{i=1}^{n}\mathbf{p}_i \tag{3.16}$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}[\mathbf{p}_i^2] - \mathbf{p}_c^2 \tag{3.17}$$

respectively for n number of data points in the data set. The variance is represented as a vector $\sigma^2 \in \mathbb{R}^3$ where each entry represent the variance in each respective dimension. A positive scalar measure of the variation in the data set is obtained by taking the scalar product with a user-specified scaling factor $\mathbf{c} \in \mathbb{R}^3$ weighting each dimension as desired. The result is floored to obtain a positive integer number l to represent the number of clusters in the data set

$$l = \lfloor \mathbf{c} \cdot \sigma^2 \rfloor \tag{3.18}$$

In this application this was set to $c = \mathbf{0.002}$. l is then used to select the top clusters in the dendrogram. This approach has severe limitations in the sense that finding the number of clusters in a dataset based on variance is *not* a suitable approach. This is because variance

only takes into account the deviation from the centroid with no regard to the number of points included in the calculation. That is, a small number of points can result in high variance of the data set and thus a large number of clusters. Even if the formula (3.18) were to be adapted to take into account the number of points in the data set, it would still be a rough approximation of the clustering in the original data set.

However, methods have been proposed to determine the optimal number of clusters in a data set including ISODATA (Ball and Hall, 1967), DYNOC (Tou, 1979), and several more as summarized by (Omran et al., 2007). These methods take an interative approach to minimize the inter-cluster distance and maximize intra-cluster distance involving several executions of a clustering algorithm such as k-means or AHC.

Due to the complexity of these methods, an alternative approach to perform clustering is desired. A method deemed more suitable for this application based primarily on its more intuitive approach is described in (Jung et al., 2003). The approach in question is built around the notion that a scalar number referred to as *clustering gain* is sufficient in measuring the optimality of a given cluster configuration. The clustering configuration that yields the highest clustering gain, is the optimal clustering configuration. This approach can be integrated into AHC by calculating the clustering gain after each cluster merge is performed and storing the configuration that has the maximum gain.

As previously mentioned, the clustering gain is used as a measure for optimality of a given clustering configuration. This measure is therefore supposed to represent any given configuration's similarity to the underlying grouping in the input data set. The similarity of some given cluster configuration is defined as the sum of the gains for each individual cluster

$$\Delta = \sum_{j=1}^{k} \Delta_j \tag{3.19}$$

for k number of clusters in the configuration. This quantity is expressed in terms of the input data set and will be defined here to maintain clarity through the following definitions. Let p_i represent data point $i \in \{1, 2, 3, ..., n\}$ in the data set containing n data points. The global centroid p_0 of the entire data set is defined as

$$p_0 = \frac{1}{n} \sum_{i=1}^{n} p_i \tag{3.20}$$

Let C_j represent cluster $j \in \{1, 2, 3, ..., k\}$ where $k \in \{1, 2, 3, ..., n\}$ is the number of

clusters in the data set according to some clustering method. The data points in cluster C_j are denoted $C_j = \{p_1^{(j)}, p_2^{(j)}, p_3^{(j)}, ..., p_{n_j}^{(j)}\}$ and the centroid of cluster C_j is defined as

$$p_0^{(j)} = \frac{1}{n_j} \sum_{i=1}^{n_j} p_i^{(j)} \tag{3.21}$$

Δ_j is defined as the difference between the inter-cluster error sum γ_j compared to the initial configuration of the input data set, and the intra-cluster error sum λ_j.

$$\gamma_j = \sum_{i=1}^{n_j} \|p_i^{(j)} - p_0\|_2^2 - \|p_0^{(j)} - p_0\|_2^2 \tag{3.22}$$

$$\lambda_j = \sum_{i=1}^{n_j} \|p_i^{(j)} - p_0^{(j)}\|_2^2 \tag{3.23}$$

respectively and thus Δ_j becomes

$$\Delta_j = \gamma_j - \lambda_j \tag{3.24}$$

$$= (n_j - 1)\|p_0 - p_0^{(j)}\|_2^2 \tag{3.25}$$

The total cluster gain defined in (3.19) thus becomes

$$\Delta = \sum_{j=1}^{k} (n_j - 1)\|p_0 - p_0^{(j)}\|_2^2 \tag{3.26}$$

Note that for singleton clusters ($n_j = 1$) the contribution to the total gain is $\Delta_j = 0$ and thus the initial configuration of AHC (containing only singleton clusters) results in $\Delta = 0$. This leads to the conclusion that $\Delta > 0$ given that the initial configuration is not the optimal configuration.

A problem arises due to the search for the optimal cluster configuration is intended to be integrated into the clustering algorithm itself, thus raising the need to implement the clustering algorithm itself with this feature integrated. This presents a severe drawback for this application in the sense that it may not be possible to take advantage of existing optimized implementations of agglomerative hierachical clustering algorithms (such as ALGLIB[5]. It is very desirable to use an existing implementation since clustering is considered a computationally intensive task (Jung et al., 2003) and thus one stand to gain

[5] `http://www.alglib.net/dataanalysis/clustering.php` (last visited 21.05.18)

much from such optimized implementations). However, by inspection of the dendrogram returned by AHC, it is possible to devise an algorithm that outputs the optimal cluster configuration based on the clustering gain described in (Jung et al., 2003). This is done by reconstructing every single merge performed by AHC which is documented in the dendrogram, calculating the clustering gain for the cluster configuration *after* the merge is applied, and finally storing the cluster configuration that has the maximum clustering gain as well as the gain itself for further comparison. The initial clustering gain is set to an arbitrary negative number. This is to guarantee that any cluster configuration will be chosen since the minimum value of the clustering gain is $\Delta \geq 0$. Consider the dendrogram returned by AHC shown in 3.7 together with its representation in ALGLIB

Listing 3.1: Dendrogram representation

```
1 ahc_output = "[[2,4],[0,1],[3,6],[5,7]]";
```

By starting at the bottom level of the dendrogram, the first merges applied by the AHC algorithm are applied sequentially and the corresponding clustering gain is computed based on the resulting cluster configuration. If the clustering gain is greater than the current maximum recorded gain, the current maximum gain is updated and the clustering configuration is stored. This processed is repeated until the top level of the dendrogram is reached (every data point is merged into one single cluster).

As it turns out, this approach has one deal-breaking drawback; the user has no control over how large the clusters in the optimal cluster configuration are or how far the clusters are from each other. The cluster configuration is based solely upon optimality represented as the clustering gain that gives a measure of how close the clustering configuration fits the clustering in the original data set. To point out exactly how this is troublesome for this application, consider figure 3.8.

As specified in the problem description and introduction, each novelty is to be closely investigated by the eye-in-hand cameras. Considering the eye-in-hand cameras' range an field of view together with the size and and inter-cluster proximity, a lot of redundant camera poses will potentially be generated if each of the clusters in figure 3.8 are to be inspected. Redundant in this context, it is referred to the measurement of the same subvolume of the workspace more than once. A solution to this problem could be to merge smaller clusters into each other based on proximity. This is in fact exactly what the AHC clustering scheme already does and thus it is unreasonable to implement yet another layer of post-processing on top of the output of the AHC output to achieve this.

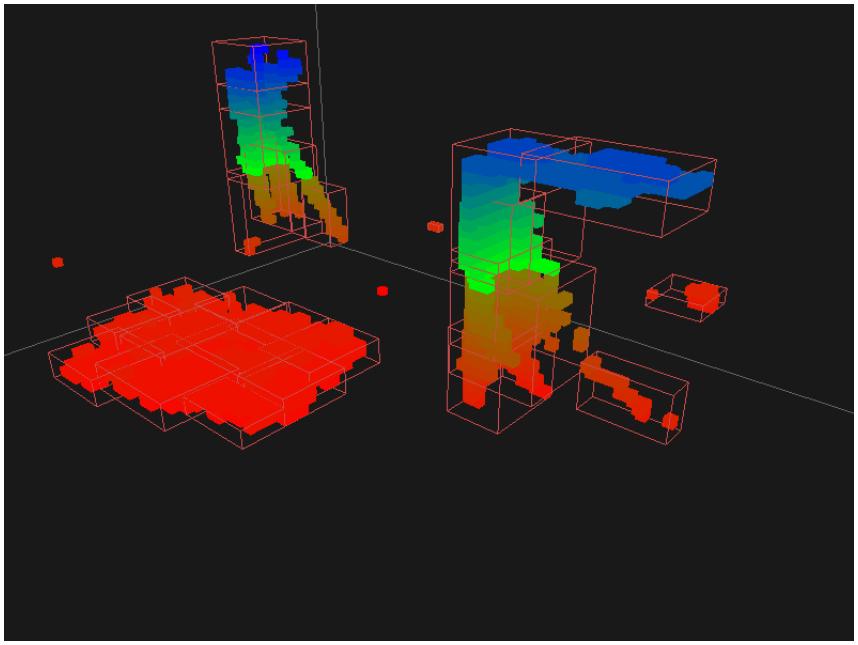

Figure 3.8: Optimal clustering configuration using the approach by Jung et al. (2003)

The desired outcome is that the bounding boxes around each clusters are to be as close to a specified box size as possible but not larger. The box size is chosen so that one box can easily fit inside the camera view frustum. An alternative to the approach described above, is to use the AHC output dendrogram to extract the first cluster from the top that fits within the specified box size. This procedure consists of recursively traversing the dendrogram from the top, generating bounding boxes that fits tightly around each cluster and checking whether the box dimensions are smaller than the specified box threshold. If so, the search along the current dendrogram branch terminates and that cluster is chosen to represent the novelty. Through experimentation, this approach was concluded to be suitable for this application (see section 5.5).

3.8 Generating camera poses

With novelties approximated as bounding boxes around a collection of novel pixels, the final task in this study presents itself. The task is to generate poses to realize with the eye-in-hand cameras to obtain a close-up measurement of the novelties. As specified in the problem description, these poses are to be generated in a fixed pattern around each novelty so that these can be observed from multiple different angles. Note that there exists some techniques for solving the problem of finding the best possible camera poses to maximize exposure of unseen surfaces to the camera. This is known as the *Next Best View*

(NBV) problem (Pito, 1997). One such technique is the solution algorithm to the NBV problem proposed by (Banta et al., 1995) which autonomously calculates the viewpoint that maximizes the information gain of the surface of some object. This technique is not implemented as part of this study due to the fact that this is currently a separate research project at SINTEF Ocean and thus resources are better spent elsewhere in this study.

In order to better explain the procedure developed for camera pose generation, consider figure 3.9

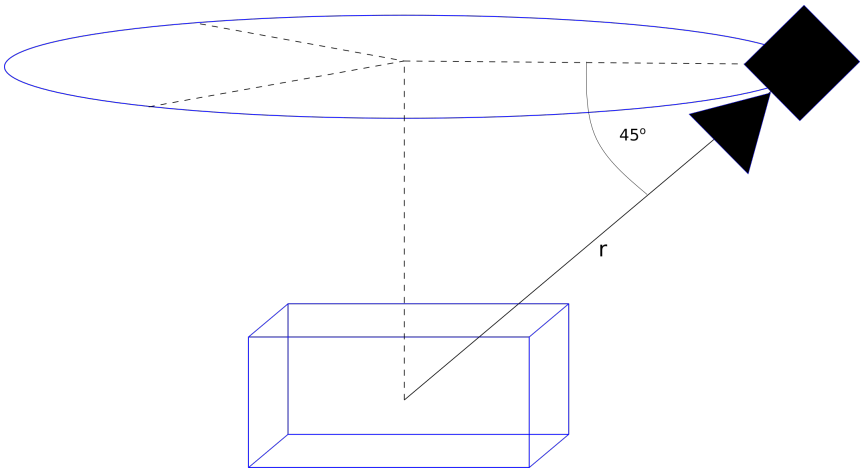

Figure 3.9: Camera poses are placed at a distance r from the centroid of the novelty bounding box on a circular path

The idea is to place viewing frustums on a circular path centered at the centroid of the novelty bounding boxes, pointing 45° downwards from the horizontal plane as shown in figure 3.9. The bounding box centroid is computed using forumla (3.20) with the 8 corners of the bounding box as the input data set. The distance r in figure 3.9 desired to be as small as possible while still allowing all the corners of the bounding box to be contained within the frustum. This is done in order to get as close as possible to the novelties and thus obtaining the most accurate representation. In order to achieve this, the eye-in-hand cameras' frustum need to be defined. Sufficient information to reconstruct this frustum is the near z_{near} and far z_{far} clipping planes, the vertical field-of-view angle ψ, and the aspect ratio a of either of the clipping planes (see figure 3.10). The procedure starts by placing n number of frustums at the centroid of the novelty boxes with the Euler angle orientation $\mathbf{\Theta} = (0, \pi/2 + k\delta, \pi/4)^T$ with $\delta = 2\pi/n$ for the $k \in \{0, 1, ..., n-1\}$ frustums using the ZYX rotation order. The reason for temporarily placing all the frustums at the centroid of the bounding box is to evaluate the location of the xy-position of the corners of the bounding box in the frustum frame defined in figure 3.10.

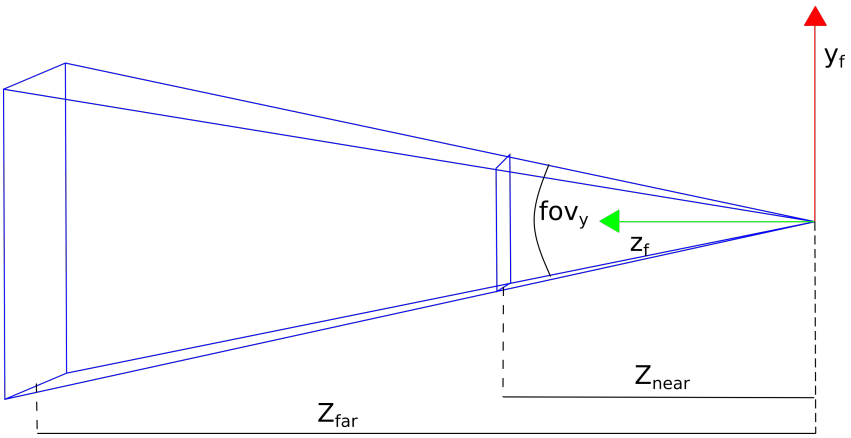

Figure 3.10: Frustum geometry

This is in order to calculate the exact z-displacement of the frustum in the frustum frame. Furthermore, for each frustum, the corners of the bounding box in the global frame $\mathbf{p}_{b,i}^{g}$ where $i \in \{0, 1, ..., 7\}$ are the enumerated corners of the bounding boxes. These are transformed to the frustum frame (see figure 3.10) by multiplication with the inverse frustum pose $\dot{\mathbf{p}}_{b,i}^{f_k} = \mathrm{T}_{f_k}^{-1}\dot{\mathbf{p}}_{b,i}^{g}$. Note that the distance r is measured along the frustum frame z axis. The corner points of the bounding box in the frustum frame $\dot{\mathbf{p}}_{b,i}^{f_k}$ can then be used to determine the maximum height and width to fit the entire box inside the frustum. The relationship between the height h and the width w of the frustum sliced at depth z is derived from the frustum geometry

$$h(z) = 2z \tan \frac{\psi}{2} \tag{3.27}$$

$$w(z) = ah(z) \tag{3.28}$$

The maximum height h_{max} and width w_{max} are found by iterating through all the $\dot{\mathbf{p}}_{b,i}^{f_k}$ for the given frustum k and the maximum x and y values are stored in x_{max} and y_{max}. The relationship (3.27) can be rearranged so that z is obtained from h_{max} and w_{max}

$$z(h) = \frac{h}{2 \tan \frac{\psi}{2}} \tag{3.29}$$

The translation distance along the negative z axis of the frustum is then calculated as

$$r = \max(z(h_{\text{max}}), z(\frac{w_{\text{max}}}{a})) \tag{3.30}$$

Chapter 4

Implementation

The result of this study is largely in the form of software implementation of the solution to the task at hand. Therefore, an overview of the software will be presented in this chapter. First, an overview of the software architecture will be presented before implementation-specific details of the steps described in chapter 3 will be given. Finally, a description of the 3-D visualization software developed for this study will be given.

4.1 Software architecture

The software developed in this study is divided into modules according to the object-oriented design approach where each module has a specific area of responsibility. Note that object-oriented design does not imply object-oriented programming. An overview of the software modules and the interaction between these are shown in figure 4.1 A brief description of each module is given below.

- **CameraSensor** acts as the rest of the system's interface to the depth sensor. An interface to the camera hardware is provided by the *librealsense* (Intel® RealSense™ API) library but this interface provides access to a large amount of functions that are not needed to solve the task at hand. This module is built on top of librealsense in order to extract only the relevant data from the sensor and in the desired format. Another purpose of introducing this extra layer between librealsense and the developed software in this study is the minimization of dependency. librealsense is still in development and has changed dramatically several times during this study and will likely be subject to some change in the future. Having a buffer layer between the vendor API and the rest of the system ensures that only this module needs to be accomodated to changes in librealsense.

- **SurfaceReconstructor** handles the integration of depth measurements into the desired form of reconstruction. This module implements only independent operations on the data supplied and stores no internal state. These operations include updating a reconstruction method of choice with a given measurement.
- **VoxelGrid** is a data structure that represents the triply indexed space array of voxels. It keeps much information on the dimensions of the represented volume and resolution and a reference to the start of the memory area in which the voxel array is stored.
- **Visualization** implements all functionality related to visualizing the relevant data structure by executing the specified draw calls and applying the correct view- and perspective transformation and lighting.
- **Viewpoint** implements the necessary functions to generate camera poses around a given bounding box around a novel region given the camera viewing frustum.
- **Frustum** is a data structure that defines a viewing frustum.
- **Detection** implements the RDE novelty detection scheme given an input image stream and produces a black-and-white output novelty image.

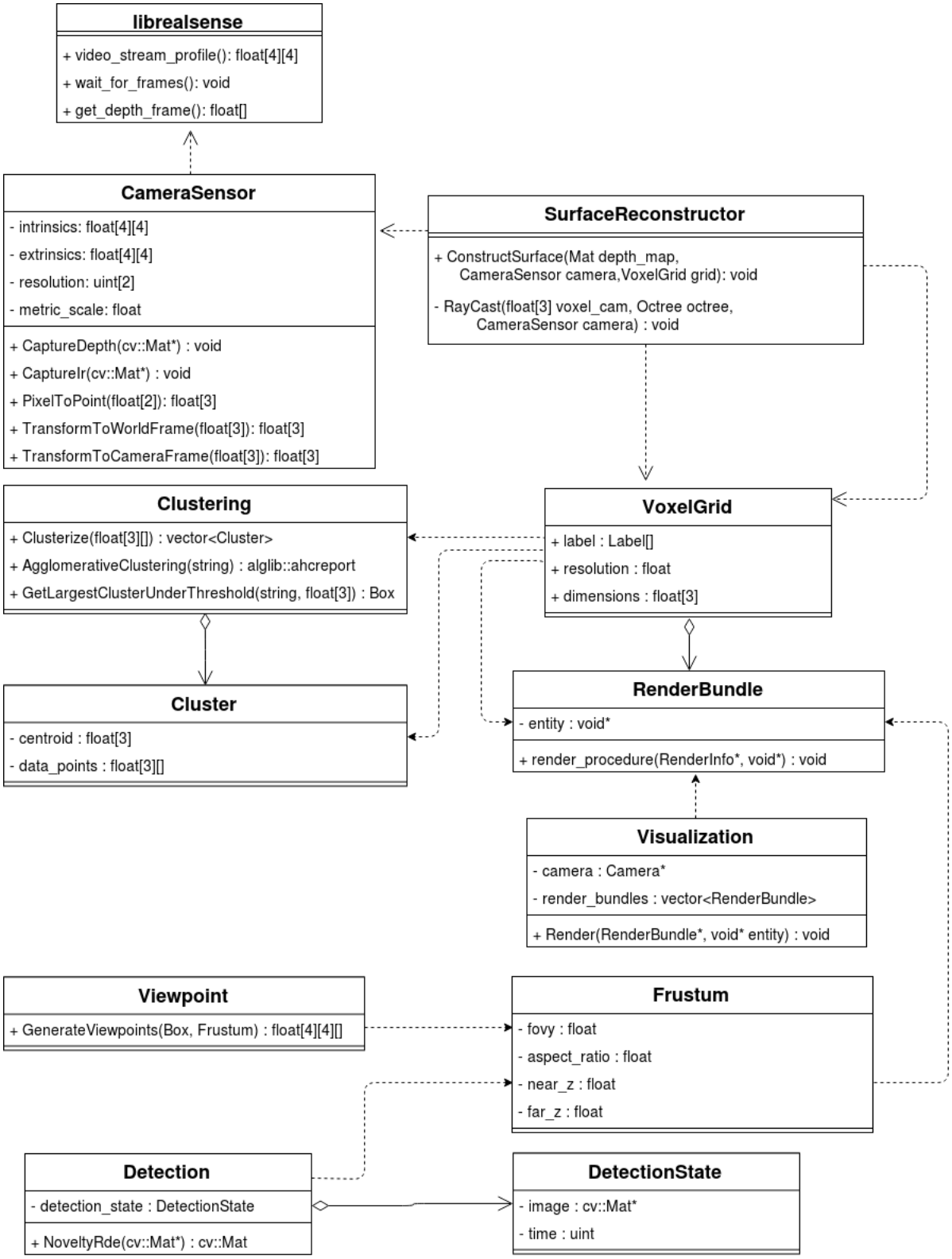

Figure 4.1: Class diagram of the software developed in this study. Dotted directional arrows shows dependency by source class on target class. Solid directional lines with diamond at the source class indicate that the target class is aggregated by the source. Core operations and data types are shown as contents of the class

4.2 Third-party software

In order to avoid time consuming implementation of basic functions related to mathematical operations and representation, low-level graphics hardware interaction, low-level system I/O hardware interaction, and operations and representations related to image processing and storage, a series of third-party software has been utilized. An overview of these and a brief summary of their use in this study i given below.

- **Intel RealSense SDK 2.0** is - as mentioned above - used to generate measurements from the camera sensor.
- **OpenCV 2.4** provides a practical way to handle a variety of different image formats and display these.
- **Eigen 3.3** is a header-only mathematics library that is used to carry out linear algebra operations.
- **OpenGL 3.0** is a graphics library that provides an API that is used to interact with the GPU. This is used to achieve fast and customizable hardware accelerated rendering.
- **OpenGL Mathematics 0.9** is a mathematic library that is based on the OpenGL Shader Language and thus provides a practical representation of vertex data in the context of rendering.
- **SDL 2.0** (Simple Direct Media Layer) is used for OpenGL context creation, window creation, and as an interface to the computer's keyboard and mouse.
- **ALGLIB**[1] is a numeric analysis and data processing library used in this study to perform clustering analysis.
- **`tup`**[2] is the build system used to compile the source code. This is a fast build system that emphasizes speed and correctness over brevity.

4.3 Sensor interface

An interface to the camera hardware is provided by the *librealsense* (Intel® RealSense™ API) library but this interface provides access to a large amount of functions that are not

[1] `www.alglib.net`
[2] `www.gittup.org`

needed to solve the task at hand. The interface is built on top of librealsense in order to extract only the relevant data from the sensor and in the desired format. The most central task of this interface is to issue commands to the sensor to start streaming IR and depth and provide easy access to the frames streamed by these. Code for activation of these streams and acquisition of frames from these are shown in listings 6.4 and 6.5. Another purpose of introducing this extra layer between librealsense and the rest of the software in this study is the minimization of dependency. librealsense is still in development and has changed dramatically several times during this study and will likely be subject to some change in the future. Having a buffer layer between the vendor API and the rest of the system ensures that only this module needs to be accommodated to changes in librealsense.

4.4 Workspace reconstruction

4.4.1 Regular 3-D voxel grid

The regular 3-D voxel grid is represented as a C-style structure as shown in listing 6.6. Upon instantiation of the VoxelGrid structure, a simple allocation is made to reserve the considerably large amount of memory needed to store the voxel array.

As seen from the VoxelGrid data structure, each voxel is stored in a flat (1-D) array and thus the effect of 3-D indexing needs to be simulated. That is, in order to obtain the voxel at index $\iota = [x, y, z]$ where each entry represent the integer position along the respective dimension, the transformation $i = x + n(y + nz)$ needs to be applied where n is the number of voxels per side in the grid. i is then used directly to index into the 1-D array that holds the voxels in the VoxelGrid structure.

4.4.2 Octree

With the octree representation of the volume, things are somewhat different. The class definitions of the octree and octree node are shown in listing 6.7. Upon instantiation of the octree only the root node is created. Each leaf node is lazily-instantiated under the root node as desired. This is done by recursively subdividing each node into 8 octants until the predefined level of division is reached. This procedure is shown in listing 6.8.

4.5 Measurement integration

As discussed in section 3.4, integration of new measurements in the form of depth images is done by directly projecting the depth image onto the 3-D volume. This is done by transformation of each pixel depth $D(\mathbf{u})$ along with its location in the image plane $\mathbf{u}$, to a voxel index on the form $\iota = [x, y, z]$. Each time a depth image pixel is mapped to a voxel, a counter within that voxel is incremented. All voxels that have a counter value grater than some threshold are then labeled as occupied. The implementation of this functionality is shown in listing 6.9. The transformation functions shown in this procedure are simple matrix multiplications with homogeneous transformation matrices.

4.6 Novelty detection

Being one of the most essential tasks in this study, novelty detection with the Cauchy type kernel is implemented as shown in listing 6.10. Novelty detection with the assumption that the pixel intensity distributions are Gaussian, have also been implemented. Since novelty detection may be done on several different images, a list of DetectionState is kept for each new image, storing the state of the detection scheme on that specific image (which is referred to by its memory address). This allows for more flexibility when testing novelty detection performance on different image streams such as the IR stream, and comparing it to depth stream novelty detection at run time.

The density estimate obtained by evaluation of the Cauchy kernel at the pixel in the previous frame is subtracted from the density estimate at the current frame and compared to a user defined threshold (here 0.1σ). If the change in density is larger than this threshold, the pixel is considered to be foreground, and is marked in white in the novelty image.

4.7 Post-Processing and novelty segmentation

Post-processing of the novelty images remains quite simple in this study. Only the mathematical morphology operation erosion being used with a quadratic structuring element. This is motivated by the experimental success that was recorded with this approach (see 5.4). The implementation is handled by the OpenCV library that provides simple function signatures to perform this.

Listing 4.1: Function signature provided by OpenCV for performing erosion

```
1 void erode(InputArray src, OutputArray dst, InputArray kernel);
```

4.8 Novelty reconstruction

The first stage of the novelty reconstruction implementation is simply achieved by putting together the results of the novelty detection (black and white novelty image) and the measurement integration. That is, the black and white novelty image is used as a mask for incrementing the voxel counters in listing 6.9. White pixels in the novelty image are transformed to voxels and the counters incremented. Black pixels are ignored.

The next stage is performing cluster analysis to approximate novel ROIs. First, clustering is performed with the third-party software named ALGLIB. The output of the clustering algorithm of choice (AHC) is a dendrogram in the shape of a 2-D array of size $n \times 2$ where n is the number of merges performed by AHC. In either of the two columns in the output dendrogram representation are indices of the clusters merged. The merges (rows) appear chronologically in the 2-D array, that is, the first row represents the first two clusters merged and the last row represents the last two clusters merged.

The source code for extraction of the largest cluster from the root node of the dendrogram as described in section 3.7.1, is shown in listing 6.11. This procedure starts by generating a list of all voxel positions defined as the least corner (closest to the volume origin) of the voxel. The bounding boxes around the clusters are defined by their least corner and the lengths in each direction (x, y, and z) in the form of a 3-D vector, and thus the output of the clustering algorithm is an array of box positions and lengths. These arrays along with an array of occupied voxel positions, and the desired cluster size is passed to the clustering algorithm and the bounding box positions and lengths are returned. The core of the clustering algorithm is shown in listing 6.12. This procedure traverses the dendrogram from the root node, and terminates the search along each branch once the cluster bounding box is less than the threshold.

4.9 Camera pose generation

Implementation details for generating a set of camera poses around a single bounding box, mirror the description given in section 3.8. The source code for doing this is shown in list-

ing 6.13. No external third-party software is used in this module. This is largely because the approach is derived from intuition and requires only simple geometric calculations.

4.10 Visualization software

Essential to this study has been the task of visualizing the concepts related to 3-D reconstruction and representation, clustering, camera pose generation, and other spatial features. In order to provide a versatile and flexible visualization tool that is easily extendable to accommodate future visualization needs, the software used in this study was developed from the ground up solely for this purpose.

Several existing software packages were considered as alternatives. A criterion for these in addition to the ones mentioned above is that these tools have to be free. Alternatives that satisfy these criteria are listed below

- **Unity 3D**[3] is a 3-D game engine that offers a wide range of functionality related to graphics rendering and 3-D digital art creation. Unity is however an extremely large software collection that includes so much functionality that is largely irrelevant to the 3-D visualization needs of this study. In addition, unity does not support the target platform which is the GNU/Linux distribution Ubuntu 17.10.

- **VTK**[4] (The Visualization Toolkit) is an advanced open-source software package for 3-D computer graphics and image processing. Although VTK seems suitable for this application it is not used due to it being too high-level to offer the necessary flexibility.

The main rendering tasks required by the visualization module is rendering of the regular 3-D voxel grid and octree. Since all the information on the contents of the volume are stored in the data structure representing that reconstruction (voxel grid or octree), the respective representation is fully responsible for generating and placing the vertices to be rendered into vertex buffer objects. Additionally, the draw call to execute rendering of the reconstruction is defined in the same module as the reconstruction. Following the object-oriented design principle, all the information required by visualization module is then packed into a data structure referred to as a render bundle. The prototype for this data structure is shown in listing 6.1. The variable *entity* is a pointer to the instance of the

[3] `https://www.unity3d.com` (last visited 12.05.2018)
[4] `https://www.vtk.org` (last visited 12.05.2018)

entity to be rendered e.g. VoxelGrid or Octree. The member *render_procedure* is a function pointer to the function that executes the draw call for the given entity. This function takes as argument a pointer to the entity itself, and a pointer to a structure containing all the necessary information on how the entity is viewed. The content of this structure is shown in listing 6.2. The variable *view_proj* is the 4×4 matrix product of the view matrix and the projection matrix $M_p \cdot M_v$ where $M_v \in \mathbb{SE}_3$ is the virtual camera view matrix and $M_p \in \mathbb{R}^{4\times4}$ is the projection matrix, in this application is the perspective projection. The remaining variables of the RenderInfo structure are the light's position and the camera's position respectively. The variable *destroy_procedure* of the Render Bundle structure is the a function pointer to the function that deallocates the given entity (passed in as a void pointer). Each entity that wants to be rendered is responsible for filling out all the information in the RenderBundle. An example of creation of the RenderBundle for the regular 3-D voxel grid is shown in listing 6.3.

Rendering using the approach described above ensures that the visualization module and the objects to be rendered are decoupled. The benefit of this is that as the objects to be rendered changes, the change is completely irrelevant to the visualization module as long as a complete RenderBundle is produced.

4.11 Documentation

Documentation for software developed in this study is created using the automatic source documentation generation tool doxygen[5]. Doxygen automatically generates documentation in the form of PDF documents, HTML documents and contains graphical dependency trees etc.

Instruction for compiling the source code can be found in the attached software repository in the README.md file.

[5] `http://www.stack.nl/~dimitri/doxygen/`

Chapter 5

Experimental Results

To evaluate the performance of the methods employed to solve the task at hand, experiments were performed on the solutions to the sub-problems. The experimental results of these methods are described in this chapter and appear chronologically. The most suitable techniques for solving each sub-problem is used when considering the next sub-problem. This is done whenever it is reasonable to assume that the subsequent sub-problem is unaffected by the result of the previous. For example, the first task is to decide a volumetric reconstruction representation. The two representation methods are evaluated before taking a closer look at the different approaches to integrating depth measurements. When a representation method is chosen, this method is assumed to be used in the subsequent experiments on integration schemes, since the integration schemes do not depend on the representation method used.

5.1 Workspace reconstruction

To obtain a performance measure of the two volumetric representation methods discussed in this study, memory consumption is investigated. To clarify, memory consumption refers to the amount of bytes needed to store the reconstruction of the workspace volume expressed in terms of the byte size of each voxel s_v. Each representation will reconstruct a volume of dimension $1m \times 1m \times 1m$ with the number of voxels per side $n = 128$ and resolution $r = 0.01m$. The volumes that are to be reconstructed in these experiments are the ones measured by the images in figure 5.1 and 5.2

These purposely contain variable amount of objects to test how the reconstruction handles different workspace contents. These images were captured by the the Intel® RealSense™ D415 mounted on a tripod $1.4\,\mathrm{m}$ above the ground, facing $60°$ downwards

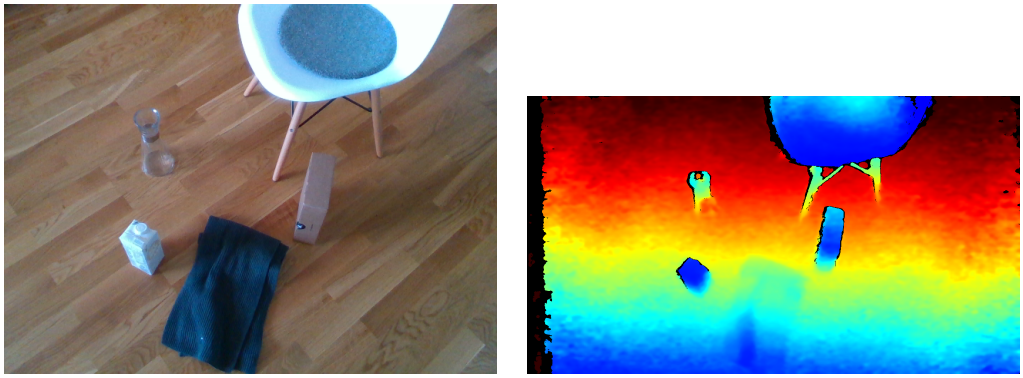

Figure 5.1: Color and depth image of scene 1

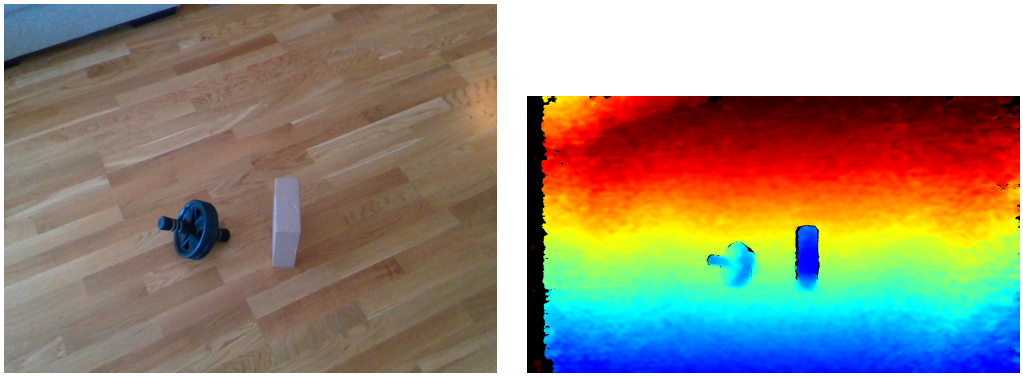

Figure 5.2: Color and depth image of scene 2

from the horizontal plane as shown in figure 5.3

The pose of the camera for this experiment was set manually in software as

$$\mathrm{T}_k^g = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & \cos(2.1) & -\sin(2.1) & 1.4 \\ 0 & \sin(2.1) & \cos(2.1) & 1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

where $180° - 60° \approx 2.1\,\mathrm{rad}$

Memory

The memory consumed by the regular 3-D voxel grid is constant given n, more specifically $m = s_v n^3 = 2097152 s_v\, B$ where $s_v = 1\, B$ (one byte).

For the octree the memory consumption estimation is somewhat more involved. Since the octree includes some overhead to allow for a more compact volumetric representation,

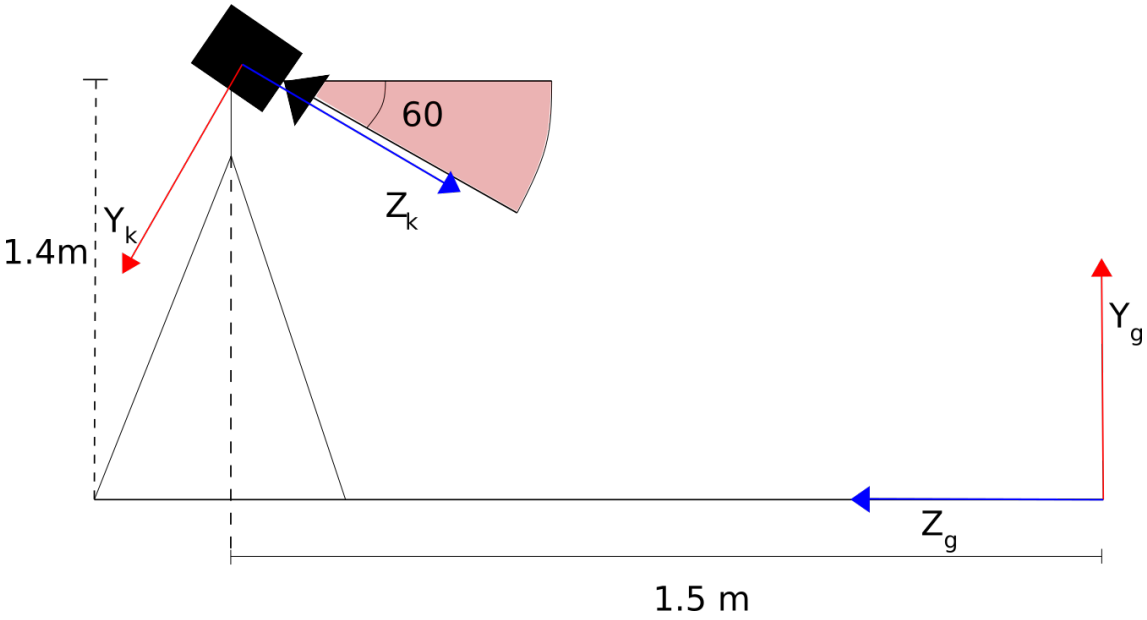

Figure 5.3: Camera setup

the memory consumption in measured in terms of the size in bytes of all the nodes in the tree. That is, also non-leaf nodes are counted. The number of nodes in the octree is obtained by traversal of the octree and incrementing a counter. Since each node in the octree stores references to octants, and contain a label indicating occupancy, the size in bytes of each node $s_n \neq s_v$. For the octree nodes, $8 \cdot 8 = 64\,B$ (on a 64-bit system) are used to store the references to the 8 octants the node can potentially be partitioned into, and $1\,B$ to hold the label VOID, FILL, or MIXED.

For the two scenes considered in this experiment, the memory consumption of the two volumetric representations are as follows.

	Regular 3-D voxel grid	Octree
Scene 1	2097152	$75417 \cdot 65 = 4902040$
Scene 2	2097152	$68985 \cdot 65 = 4484025$

Table 5.1: Memory consumption in bytes of Octree and voxel grid representation of the same scene

This shows that the total number of nodes (including non-leaf nodes) of the octree is considerably less than the number of unit cubes in the voxel grid. However, the extra storage space required to keep track of how these nodes are related to each other result in approximately double the amount of storage space consumed by the regular 3-D voxel grid.

5.2 Integrating new measurements

The evaluation of the quality of the measurement integration is done by visual inspection. That is, the reconstructed scene is visually compared to actual scene. This is done for the direct depth map projection with and without observation count filtering, and with the TSDF integration scheme.

The experiments are done on a series of 9 images of the scene shown in figure 5.4

Figure 5.4: Sample of scene measurement used to in this experiment

taken from 9 different camera poses with a robotic manipulator. The Intel® RealSense™ SR300 is mounted on a robotic manipulator and calibrated so that the pose of the camera sensor is known. The result of reconstruction of the scene is shown below.

From the figures 5.5, 5.6, and 5.7 it is concluded that the most suitable approach to measurement integration is the method involving filtering based on number of surface observations. As shown in figures 5.5 and 5.7, noise artifacts are present in a larger degree. The approach chosen offers a simple and versatile scheme for integration of depth measurements.

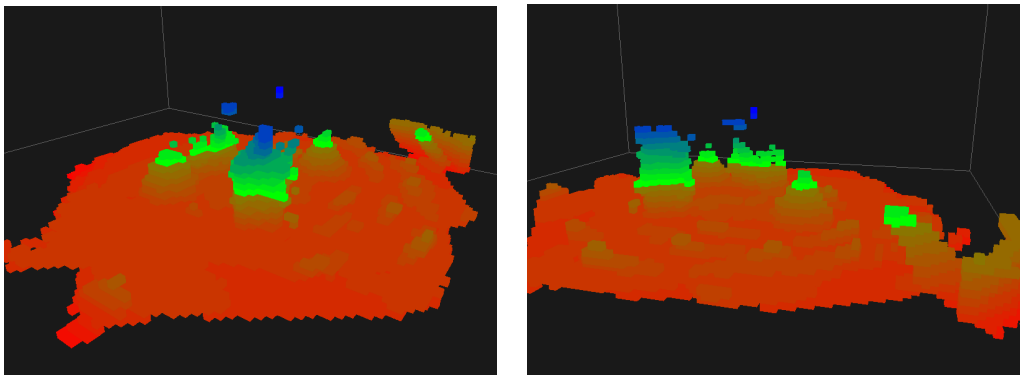

Figure 5.5: Direct projection of a depth measurement onto the volume

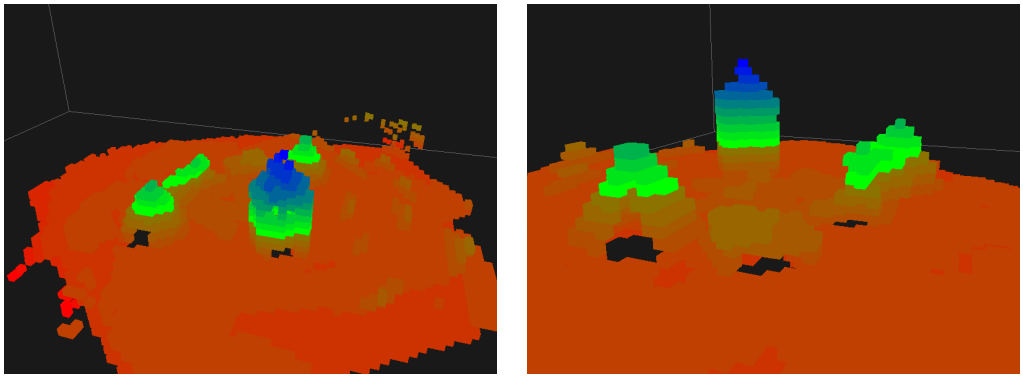

Figure 5.6: Direct projection of a depth measurement with filtering based on number of observations where voxels with less than 50 observations over the 9 images captured, are dropped.

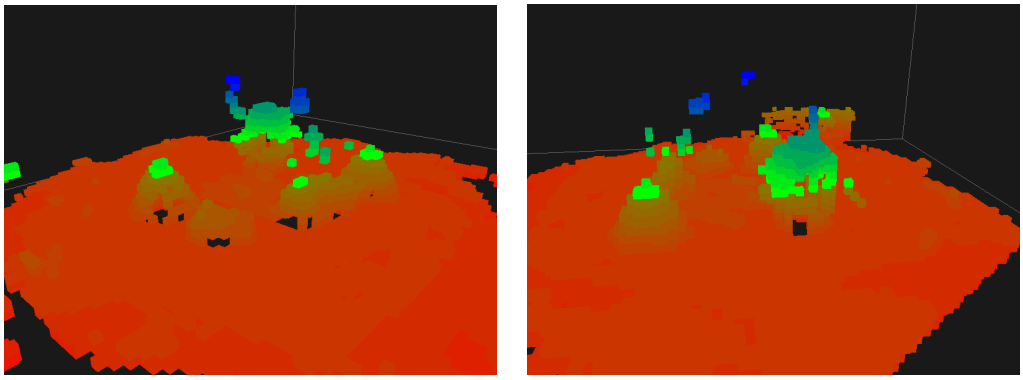

Figure 5.7: Integration of measurements based on the TSDF update scheme

5.3 Novelty Detection

Below are the results of novelty detection with both the Cauchy and Gaussian kernels.

Another experiment is performed to investigate the ability of the novelty detection scheme to pick up small changes in depth alone. This is done by simply raising the object

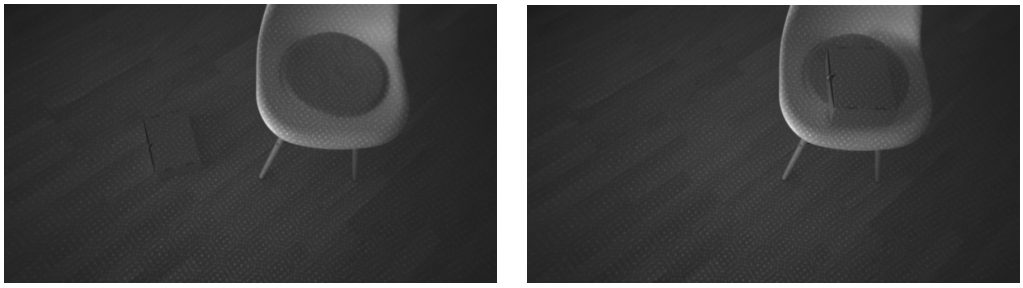

Figure 5.8: The scene change from which the following results are obtained

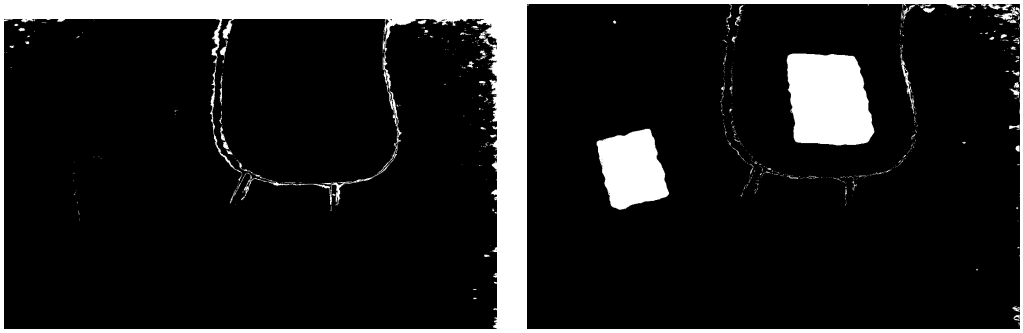

Figure 5.9: Novelty image with the Cauchy kernel and novelty threshold 0.1σ

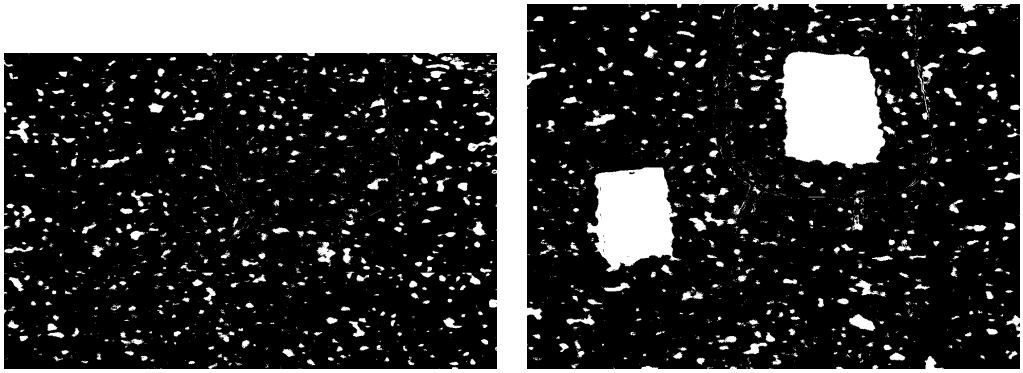

Figure 5.10: Novelty image with the Gaussian kernel and novelty threshold 0.25

by placing another object underneath it. Another box of length $10\,\mathrm{cm}$ was used here.

In all experiments performed under this section, the background model has been trained with 25 frames before starting the novelty detection in order to adapt to the scene. However, when the assumption that the intensities are distributed with the Gaussian distribution, a large amount of noise is present in the novelty image as seen in figures 5.10 and 5.13.

The ability of the novelty detection schemes discussed in this study to adapt to persist-

Figure 5.11: The scene measurement in the IR spectrum from which the following results are obtained. The box is raised $10\,\mathrm{cm}$ in order to isolate performance in the z_k direction

Figure 5.12: Novelty image with the Cauchy kernel and novelty threshold 0.1σ

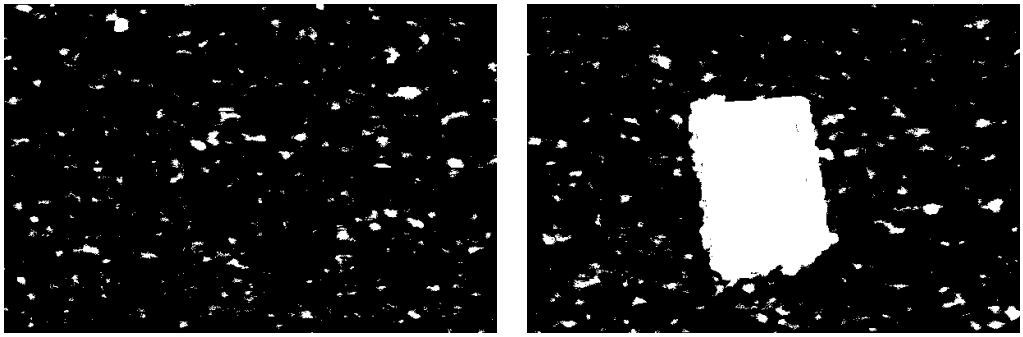

Figure 5.13: Novelty image with the Gaussian kernel and novelty threshold 0.25 and $\alpha = 0.2$

ent change is a feature that increases robustness. Consider the image series in figure 5.14. The cardboard box to the left is rotated each frame, resulting in a strong signature in the novelty image. As time passes, the novelty signature fades for the left box in spite of the constant rotation. However, this does not affect ability of the novelty detection scheme to detect changes elsewhere in the frame. Since the RDE novelty detection scheme is a pixel-wise scheme, the pixel-wise probabilities for novelty are adapted in regions subject to persistent change.

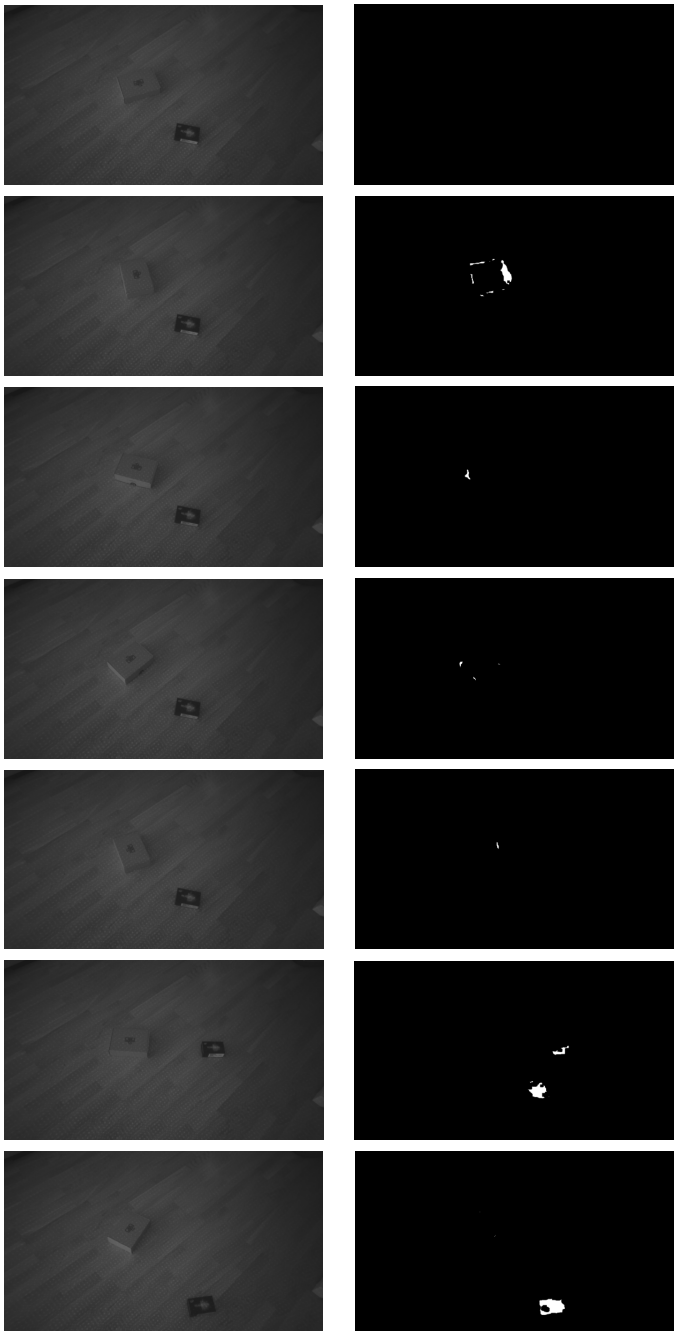

Figure 5.14: Novelty detection of a series of depth images. The correspondng IR frames (left) and novelty image (right). The initial frame is shown at the top and subsequent frames are shown chronologically under. Note how sensitivity to regions with persistent change is reduced.

5.4 Post-Processing and novelty segmentation

From 5.9 it is seen that there is noise present but the areas of these artifacts are relatively small and therefore the mathematical morphology operation *erosion* may effectively sup-

press these as described by Haralick et al. (1987). The result of applying erosion to the raw novelty image are shown in 5.15

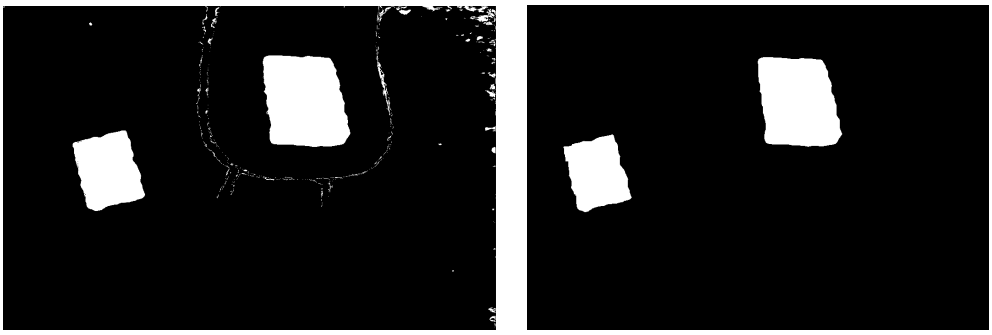

Figure 5.15: Erosion by a square structuring element of width 9 pixels

The result of the erosion operation shown to the right in figure 5.15 completely removes the contour of the chair and noise present in the image to the left. The actual novelties, referring to the movement of the box is successfully isolated. However, caution has to be shown when using erosion as the edges of the actual novelty are eroded away. This is acceptable when the structuring element is small enough.

5.5 Novelty reconstruction

The result of the optimal clustering approach proposed by Jung et al. (2003) is shown in figure 5.17 The reconstruction with optimal clustering of the measurements in figure 5.16 is shown in figure 5.17. As previously mentioned, the clustering using the approach by Jung et al. (2003) provides too small clusters according to what is reasonable to expect to fit within one camera frame.

The result of the bounding box fitting approach based on recursive dendrogram traversal to find first cluster that produces a bounding box that is smaller than the given threshold, on the measurements in figure 5.18 is shown in figure 5.19. The clustering shown in figure 5.18 is deemed satisfactory both due to a reasonable clustering size and the fact that they fit relatively close to the novel voxels. The lower limit for the bounding box lengths used in this experiment is set to $60\,\mathrm{m}$ in all three spatial dimensions.

Note that clusters with a small amount of voxels are dropped from the bounding box fitting as these are likely to be noise artifacts that have made their way into the voxel-based reconstruction.

Compared to the optimal clustering approach in (Jung et al., 2003), the larger clusters are more suitable for this applications regardless if they are optimal. This is because this cluster configuration promotes less redundancy when inspecting the novel sub-volumes.

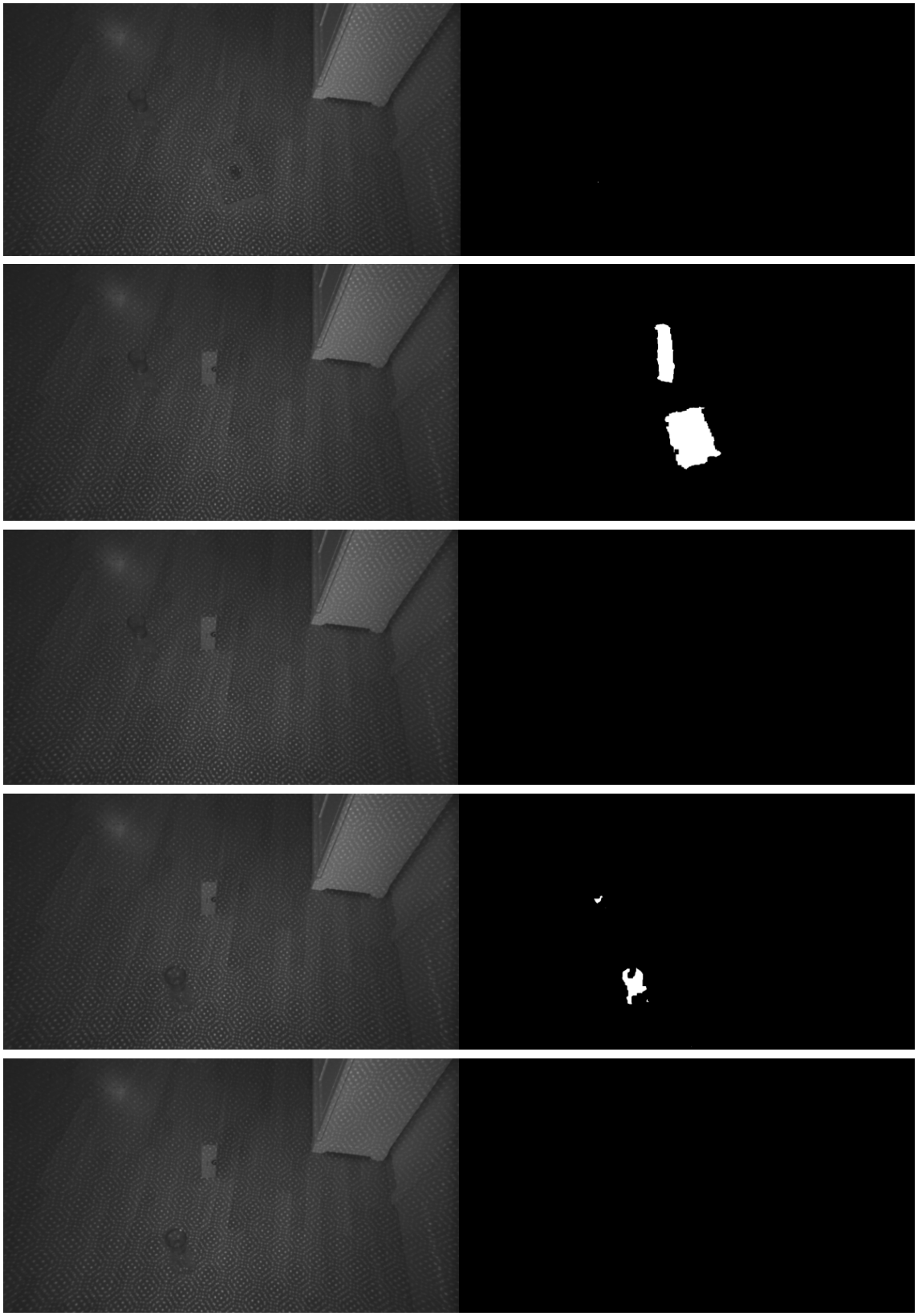

Figure 5.16: The 5 first (from the top) frames of the novelty detection with IR measurement to the right and corresponding novelty image to the left. The objects in the image include a cardboard box and a clear glass vase.

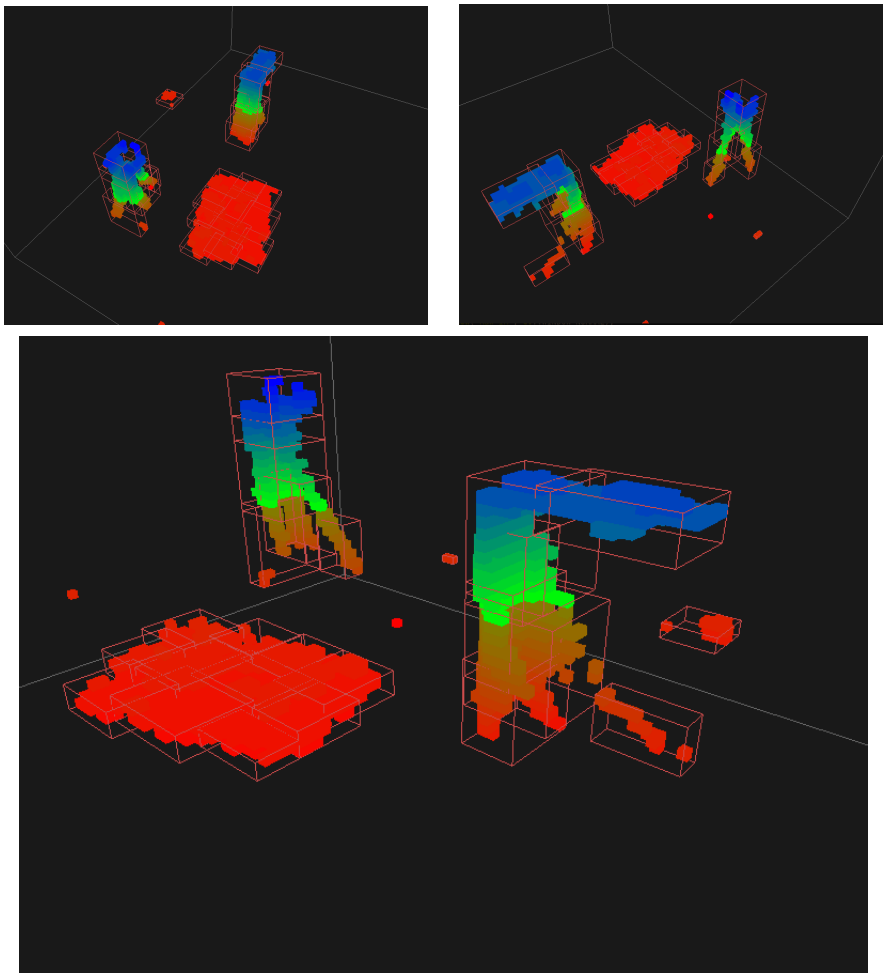

Figure 5.17: Reconstruction with optimal clustering based on the approach by Jung et al. (2003).

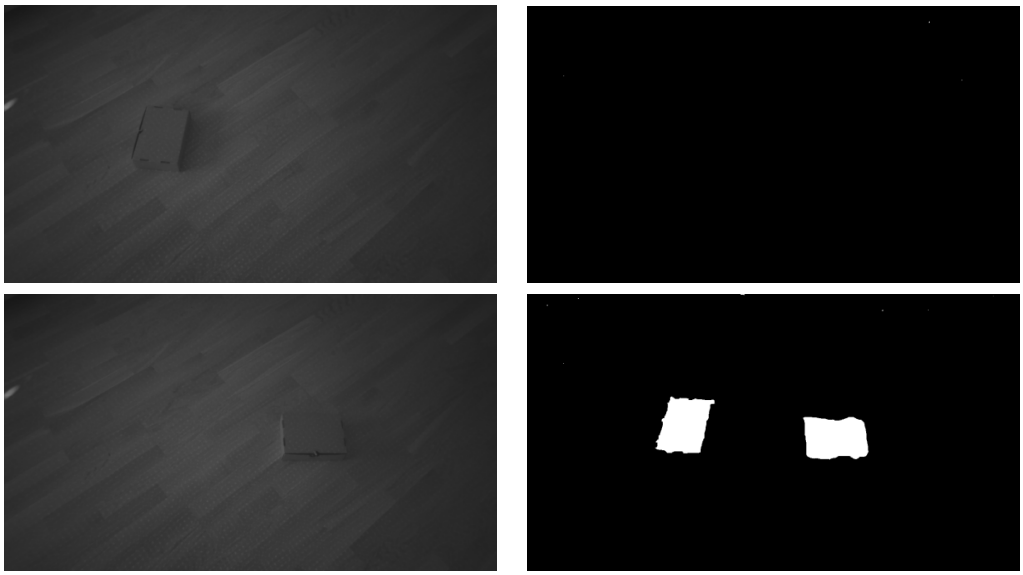

Figure 5.18: The first 2 frames (left) from the top with corresponding novelty image (right)

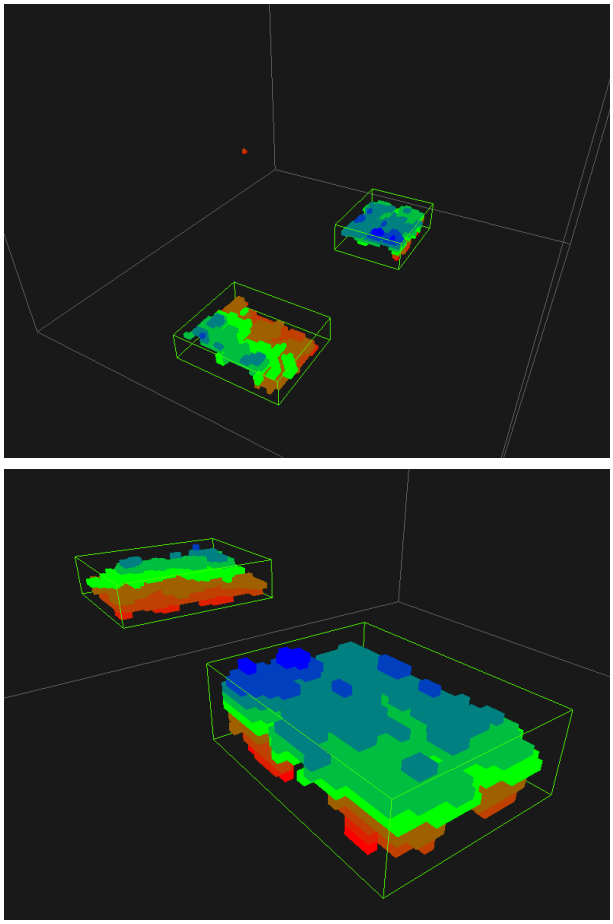

Figure 5.19: Reconstruction of the measurements shown in figure 5.18 with bounding boxes

5.6 Camera pose generation

Application of this camera pose generation scheme is shown in figure 5.20

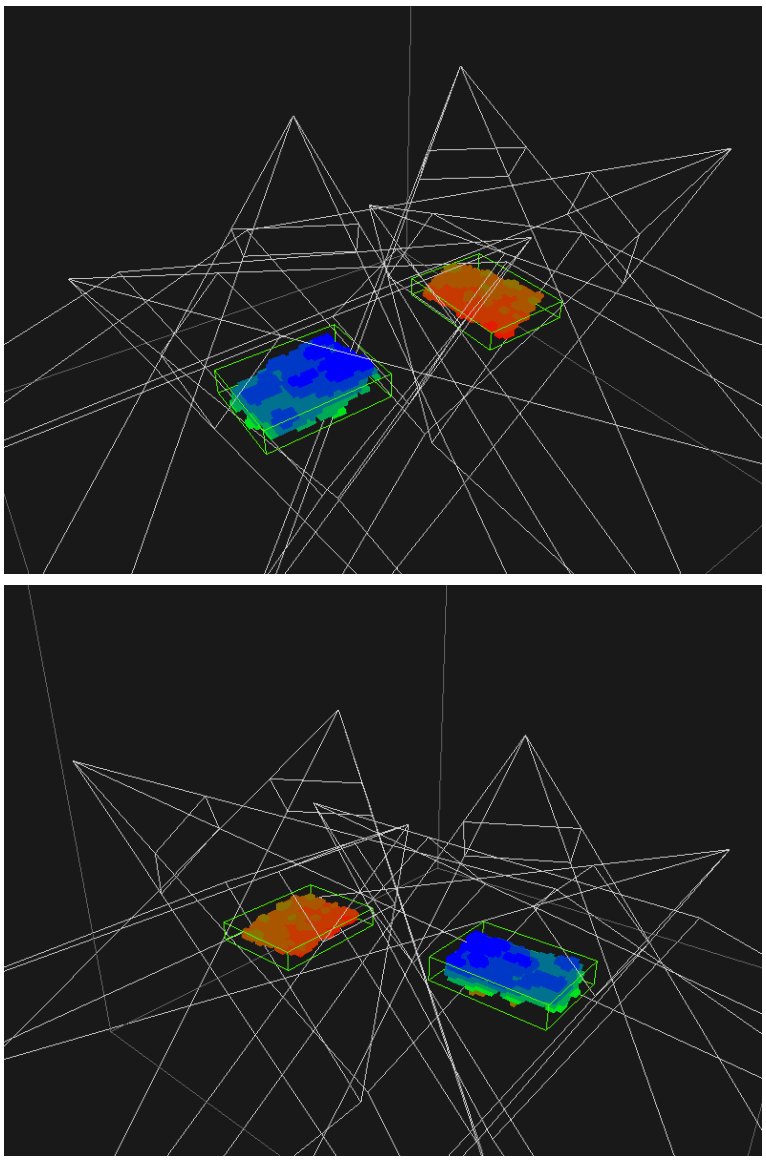

Figure 5.20: 3 frustums (rendered in white) are placed around each novelty bounding box as close as possible

From the result in figure 5.20 it is seen that the reconstruction of the novelties and the bounding box around them are of acceptable accuracy. This evaulated by visual inspection of the real-world scene, the measurements (raw and filtered), and the reconstruction itself. This is because clusters with a small amount of voxels are not considered.

Chapter 6

Conclusion

Considering the results, this study is considered to accomplish the goals formulated in the problem description. The first task of this study was to determine a voxel-based reconstruction method. The method that was found to be most suitable for this application was to use a regular 3-D voxel grid to represent 3-D space. The conclusion to use the voxel grid instead of octree is based on the experiments on memory space usage and indexing speed. The regular voxel grid is a simpler structure that allows for faster indexing. The memory space usage of the octree when counting the space needed to store the child pointers to every node exceeds that of the voxel grid for the same sized volume, even though there is considerably fewer nodes required (even fewer leaf nodes/voxels).

When integrating new measurements into the regular 3-D voxel grid, the three techniques considered were a direct projection of a depth image onto the volume, a direct projection with filtering of voxels based on number of observations, and the truncated signed distance function. Although the TSDF approach has shown great promise in the study by Newcombe et al. (2011) the more simplistic direct mapping with voxel counter based filtering is used. Visual inspection and comparison of the resulting 3-D reconstruction with the measurements used and the measured scene itself, have shown that this results in a higher quality reconstruction.

The novelty detection scheme used in this study was the RDE approach by Morris and Angelov (2014). The scheme was tested with the Cauchy type kernel used to estimate the probability of the intensity occurring in the same pixel location based on all previous frames. A modified version of the RDE scheme was adapted to the assumption that the pixel intensities are distributed according to the Gaussian distribution. This approach was

found to be usable but the amount of noise present in the resulting novelty image do not justify the choice of this approach over RDE with the Cauchy type kernel.

The novelty images contain some amount of noise that is not desirable to integrate into the 3-D reconstruction of the novel ROIs. The two methods tested for suppressing the presence of noise artifacts were thresholding the surface area of these artifacts, and the erosion operation in mathematical morphology. With the Cauchy type kernel in the novelty detection scheme, the noise artifacts in the novelty image are relatively small and thus erosion by a square structuring element provides good noise suppression.

The grouping of novel voxels is performed to determine sub-volumes within the workspace to further investigate with the eye-in-hand cameras mounted on the robotic manipulators. The most suitable way to do this was to use an existing clustering algorithm known as agglomerative hierachical clustering. The criterion for the groups to further investigate, is that they need to be smaller than some pre-defined size (defined as lengths of a bounding box), but large enough to avoid excess scanning. Therefore, in order to extract from the AHC output the most suitable cluster configuration, the output is traversed from the last merge performed, backwards and terminates when the first cluster that is smaller than some desired box size is found. This box is then used to place a camera frustum so that the entire box is contained within the frustum.

Bibliography

Ball, G. H., Hall, D. J., 1967. A clustering technique for summarizing multivariate data. Systems Research and Behavioral Science 12 (2), 153–155.

Banta, J. E., Zhien, Y., Wang, X. Z., Zhang, G., Smith, M. T., Abidi, M. A., 1995. Best-next-view algorithm for three-dimensional scene reconstruction using range images. Proc.SPIE 2588, 418 – 429.

DasGupta, A., 2010. Normal Approximations and the Central Limit Theorem. Springer New York, New York, NY, pp. 213–242.

Egeland, O., Gravdahl, J. T., 2002. Modeling and simulation for automatic control. Vol. 76. Marine Cybernetics Trondheim, Norway.

Fossen, I. T., 2011. Handbook of Marine Craft Hydrodynamics and Motion Control. John Wiley & Sons, Ltd.

GOM, 2018. Atos - industrial 3d scanning technology. `https://www.gom.com/metrology-systems/atos.html`, last visited: 2018-05-25.

Gonzalez, R. C., Woods, R. E., 2010. Digital Image Processing, 3rd Edition. Pearson.

Halkidi, M., Vazirgiannis, M., 2001. Clustering validity assessment: finding the optimal partitioning of a data set. In: Proceedings 2001 IEEE International Conference on Data Mining. pp. 187–194.

Han, J., Shao, L., Xu, D., Shotton, J., 2013. Enhanced computer vision with microsoft kinect sensor: A review. IEEE transactions on cybernetics 43 (5), 1318–1334.

Haralick, R. M., Sternberg, S. R., Zhuang, X., July 1987. Image analysis using mathematical morphology. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-9 (4), 532–550.

Hartley, R. I., Zisserman, A., 2004. Multiple View Geometry in Computer Vision, 2nd Edition. Cambridge University Press, ISBN: 0521540518.

Intel®Corporation, 2016. Intel ® realsense ™ camera sr300. embedded coded light 3d imaging system with full high definition color camera. product datasheet.

Jackins, C., Tanimoto, S., 1980. Oct-trees and their use in representing three-dimensional objects. Computer Graphics and Image Processing 14 (3), 249–270.

Jung, Y., Park, H., Du, D.-Z., Drake, B. L., 2003. A decision criterion for the optimal number of clusters in hierarchical clustering. Journal of Global Optimization 25 (1), 91–111.

Lacroute, P., Levoy, M., 1994. Fast volume rendering using a shear-warp factorization of the viewing transformation. In: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '94. pp. 451–458.

Lazaros, N., Sirakoulis, G. C., Gasteratos, A., 2008. Review of stereo vision algorithms: From software to hardware. International Journal of Optomechatronics 2 (4), 435–462.

Lippiello, V., Siciliano, B., Villani, L., 2005. Eye-in-hand/eye-to-hand multi-camera visual servoing. In: Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on. IEEE, pp. 5354–5359.

Marr, D., Poggio, T., 1979. A computational theory of human stereo vision. Proceedings of the Royal Society of London B: Biological Sciences 204 (1156), 301–328.

Moley, 2018. Moley - the world's first robotic kitchen. `www.moley.com`, last visited: 2018-05-25.

Morris, G., Angelov, P., Oct 2014. Real-time novelty detection in video using background subtraction techniques: State of the art a practical review. In: 2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC). pp. 537–543.

Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohli, P., Shotton, J., Hodges, S., Fitzgibbon, A., 2011. Kinectfusion: Real-time dense surface mapping and tracking. In: Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality. IEEE Computer Society, pp. 127–136.

Omran, M. G., Engelbrecht, A. P., Salman, A., 2007. An overview of clustering methods. Intelligent Data Analysis 11 (6), 583–605.

Pito, R., 1997. Automated surface acquisition using range cameras. University of Pennsylvania.

Scharstein, D., Szeliski, R., June 2003. High-accuracy stereo depth maps using structured light. In: 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings. Vol. 1. pp. I–195–I–202 vol.1.

Sobral, A., Vacavant, A., 2014. A comprehensive review of background subtraction algorithms evaluated with synthetic and real videos. Computer Vision and Image Understanding 122, 4 – 21.

Spong, M. W., Hutchinson, S., Vidyasagar, M., 2006. Robot Modeling and Control. John Wiley & Sons, Inc.

Steinbach, M., Karypis, G., Kumar, V., et al., 2000. A comparison of document clustering techniques. In: KDD workshop on text mining. Vol. 400. Boston, pp. 525–526.

Tou, J. T., Dec 1979. Dynoc—a dynamic optimal cluster-seeking technique. International Journal of Computer & Information Sciences 8 (6), 541–547.

Tsai, R. Y., Lenz, R. K., Jun 1989. A new technique for fully autonomous and efficient 3d robotics hand/eye calibration. IEEE Transactions on Robotics and Automation 5 (3), 345–358.

Walpole, E. R., Myers, H. R., Myers, L. S., Ye, K., 2012. Probability & Statistics for Engineers & Scientists, ninth Edition. Pearson Education Inc.

Werner, D., Al-Hamadi, A., Werner, P., 2014. Truncated Signed Distance Function: Experiments on Voxel Size. pp. 357–364.

Willett, P., 1988. Recent trends in hierarchic document clustering: A critical review. Information Processing & Management 24 (5), 577 – 597.

Appendix

Write your appendix here...

Listing 6.1: Blueprint of the RenderBundle type which keeps the required information for rendering any entity by the visualization module

```
1 typedef struct RenderBundle {
2     void* entity;
3     void (*render_procedure)(void*, RenderInfo*);
4     void (*destroy_procedure)(void*);
5 } RenderBundle;
```

Listing 6.2: Blueprint of the structure containing information on how the rendered entity is to be viewed

```
1 typedef struct RenderInfo {
2     glm::mat4 view_proj;
3     glm::vec3 light_pos;
4     glm::vec3 view_pos;
5 } RenderInfo;
```

Listing 6.3: Example of generation of a Render Bundle for the voxel grid

```
1 RenderBundle* voxelgrid_create_renderbundle(VoxelGrid* voxelgrid) {
2     RenderBundle* rbundle = new RenderBundle;
3     rbundle->entity = (void*) voxelgrid;
4     rbundle->render_procedure = &voxelgrid_render;
5     rbundle->destroy_procedure = &voxelgrid_destroy;
6     return rbundle;
7 }
```

Listing 6.4: Code for activation of depth or IR stream of Intel® RealSense™ devices with libreal-sense

```
rs2::device dev = GetDevice("Intel");
pipeline_ = rs2::pipeline();
ActivateStream("depth_stream", &config_);
ActivateStream("ir_stream", &config_);
pipeline_profile_ = pipeline_.start(config_);
Warmup(&pipeline_, 50);

rs2::device CameraSensor::GetDevice(const std::string dev_name) {

    rs2::context ctx;
    rs2::device_list devices = ctx.query_devices();
    rs2::device selected_device;

    if (devices.size() == 0) {

        std::cerr << "No device connected" << std::endl;

        rs2::device_hub device_hub(ctx);
        selected_device = device_hub.wait_for_device();
    }
    else {
        for (rs2::device device : devices) {

            std::string curr_dev_name = GetDeviceName(device);

            if (dev_name ==
                curr_dev_name.substr(0, dev_name.length())) {

                std::cout << "Found desired device: "
                << curr_dev_name << std::endl;
                selected_device = device;
                break;
            }
        }
```

```
35      }
36
37      return selected_device;
38  }
39
40  void CameraSensor::ActivateStream(const std::string stream,
41                                    rs2::config* config) {
42
43      rs2_stream stream_type;
44
45      if (stream == "depth_stream")
46          stream_type = RS2_STREAM_DEPTH;
47      else if (stream == "ir_stream")
48          stream_type = RS2_STREAM_INFRARED;
49
50      config->enable_stream(stream_type);
51  }
```

Listing 6.5: Capture a depth frame.

```
1   rs2::frameset CameraSensor::Capture() {
2
3       rs2::frameset frames = pipeline_.wait_for_frames();
4       return frames;
5   }
6
7   void CameraSensor::CaptureDepth(cv::Mat* image) {
8
9       rs2::depth_frame frame =
10          Capture().first(RS2_STREAM_DEPTH).as<rs2::depth_frame>();
11
12      *image = cv::Mat(cv::Size(frame.get_width(),
13          frame.get_height()), CV_16UC1,
14          (void*) frame.get_data());
15  }
```

Listing 6.6: Type definition of the data structures representing a voxel and a grid of these.

```
typedef enum {
    VOID,
    FILL,
    MIXED
} Label;

 typedef struct Voxel {
    Label label;
    float truncated_signed_distance;
    float tsd_weight;
    uint32_t n_hits = 0;
} Voxel;

typedef struct VoxelGrid {
    Voxel* data;
    float resolution;
    uint32_t n_voxels_per_side;
    float dimension;
    uint32_t n_total;
} VoxelGrid;
```

Listing 6.7: Class definition of the octree representation. Only core function signatures are shown.

```

class Octree {

public:
    Octree(float diameter, float resolution);

    ~Octree();

    /**
     * @brief Get voxel at max octree depth from index.
     * If it does not exist, create it from nearest leaf node.
     */
    Node* GetVoxelAt(int i, int j, int k);
```

```
14
15      /**
16       * @brief Mark voxel at given indices as occupied.
17       */
18      void Occupy(int i, int j, int k);
19
20      /**
21       * @brief Get index voxel at lowest level that surrounds a
22       * given point in world space aligned with the voxel
23       */
24      Eigen::Vector3i PointToVoxelIndex(Eigen::Vector3f point) const;
25
26      /**
27       * @brief Mark all leaf nodes with TSDF value less
28       * than a given threshold to FILL and all others as VOID
29       */
30      void OccupyTsdfSurfaceNodes(float tsdf_threshold);
31
32      /**
33       * @brief Grow the tree to the maximum level by
34       * subdividing every node recursively until the
35       * maximum level is reached
36       */
37      void Grow();
38
39  private:
40      Node* root_;
41      int n_levels_;
42      float resolution_;
43  };
44
45  class Node {
46
47  public:
48      /**
49       * @brief Construct node at a given world space location
```

```
50          * @brief size Size of the sides of the node
51          */
52         Node(Node* parent, float x, float y, float z, float size);
53
54         ~Node();
55
56         /**
57          * @brief Sudivide node into equally sized octants
58          */
59         void Subdivide();
60
61         /*
62          * @brief Grow a given number of levels under this node.
63          */
64         void Grow(int n_levels);
65
66         void DestroyChildren();
67
68         void Destroy();
69
70         /*
71          * @breif Get index of sub-octant in which the
72          * voxel with given indices is located
73          */
74         int GetOctantIndexOfVoxel(int i, int j, int k,
75                                   const float& resolution) const;
76
77         void Draw(Camera* camera);
78
79         /**
80          * @brief Get all leaf nodes under the given node
81          *
82          * @param leaf_nodes Vector where all the leaf nodes are
83          * added incrementally as they are discovered
84          */
85         void GetLeafNodes(std::vector<Node*>& leaf_nodes);
```

```
86
87      bool IsLeafNode() const;
88
89      /**
90       * @brief Apply TSDF update scheme
91       */
92      void UpdateTsdf(float tsdf, float weight);
93
94  private:
95      Node** children_;
96      Label status_;
97      Node* parent_;
98      float size_;
99      float min_x_, min_y_, min_z_;
100     float max_x_, max_y_, max_z_;
101     Eigen::Vector3f center_;
102
103     float tsd_;
104     float tsd_weight_;
105 };
```

Listing 6.8: Implementation of the octree node subdivision

```
1
2  void Node::Subdivide() {
3
4      children_ = new Node*[8];
5
6      float half_size = size_ / 2.0f;
7
8      children_[0] = new Node(this, min_x_, min_y_ + half_size,
9          min_z_ + half_size, half_size);
10
11     children_[1] = new Node(this, min_x_,
12     min_y_ + half_size, min_z_, half_size);
13
14     children_[2] = new Node(this, min_x_, min_y_,
```

```
15          min_z_ + half_size, half_size);
16
17      children_[3] = new Node(this, min_x_, min_y_,
18          min_z_, half_size);
19
20      children_[4] = new Node(this, min_x_ + half_size,
21          min_y_ + half_size, min_z_ + half_size, half_size);
22
23      children_[5] = new Node(this, min_x_ + half_size,
24          min_y_ + half_size, min_z_, half_size);
25
26      children_[6] = new Node(this, min_x_ + half_size,
27          min_y_, min_z_ + half_size, half_size);
28
29      children_[7] = new Node(this, min_x_ + half_size,
30          min_y_, min_z_, half_size);
31  }
32
33  void Node::Grow(int n_levels) {
34
35      Subdivide();
36
37      if (n_levels > 1) {
38
39          for (int i = 0; i < 8; i++)
40              children_[i]->Grow(n_levels - 1);
41      }
42  }
```

Listing 6.9: Procedure for integrating a depth image into a VoxelGrid by incrementing a voxel-specific counter

```
1  void SurfaceReconstructor::ConstructSurfaceCount(
2      const cv::Mat& depth_map,
3      CameraSensor* camera, VoxelGrid* grid) {
4
5      float meter_scale = camera->MeterScale();
```

```
6
7       for (int i = 0; i < depth_map.rows; i++) {
8
9           for (int j = 0; j < depth_map.cols; j++) {
10
11              float depth = meter_scale *
12                  depth_map.at<uint16_t>(i, j);
13
14              Eigen::Vector3f camera_frame =
15                  camera->PixelToPoint(i, j, depth);
16
17              Eigen::Vector3f world_frame =
18                  camera->TransformToWorldFrame(camera_frame);
19
20              Voxel* voxel = voxelgrid_get_voxel_at_position(
21                  grid, world_frame);
22
23              if (voxel != NULL)
24                  voxel->n_hits++;
25          }
26      }
27  }
```

Listing 6.10: RDE novelty detection with Cauchy type kernel.

```
1
2   typedef struct DetectionState {
3
4       cv::Mat* image;
5       uint32_t t;
6   } DetectionState;
7
8   void detection_novelty_rde_cauchy(cv::Mat* image,
9       float** mean, float** var, const float scale,
10      const float alpha, const float nov_thresh,
11      cv::Mat* novelty) {
12
```

```
13      uint32_t n_rows = image->rows;
14      uint32_t n_cols = image->cols;
15      static float* last_prob;
16
17      if (*mean == NULL && *var == NULL) {
18
19          *mean = new float[n_rows * n_cols];
20          detection_create_state(image);
21          last_prob = new float[n_rows * n_cols];
22      }
23
24      DetectionState* dstate = detection_get_state(image);
25
26      for (uint32_t i = 0; i < n_rows; i++) {
27
28          for (uint32_t j = 0; j < n_cols; j++) {
29
30              uint32_t index2d = Utility::Index2D(i, j, n_cols);
31              const float intensity = scale *
32                  image->at<uint16_t>(i, j);
33
34              // First pass
35              if (dstate->t == 1) {
36
37                  last_prob[index2d] = 0.0f;
38                  (*mean)[index2d] = intensity;
39                  (*ex2)[index2d] = pow(intensity, 2);
40              }
41              else {
42
43                  (*ex2)[index2d] = (dstate->t - 1.0f) /
44                      dstate->t * (*ex2)[index2d] + 1.0f /
45                      dstate->t * pow(intensity, 2);
46
47                  (*mean)[index2d] = (dstate->t - 1.0f) /
48                      dstate->t * (*mean)[index2d] + 1.0f /
```

```
                    dstate->t * intensity;
            }

            if (novelty->empty())
                *novelty = cv::Mat(cv::Size(image->cols,
                    image->rows), CV_16UC1);

            // Cauchy kernel
            float prob = 1.0f / (1.0f + pow(intensity -
                (*mean)[index2d], 2) +
                (*ex2)[index2d] - pow((*mean)[index2d], 2));

            if (fabsf(prob - last_prob[index2d]) >
                fabsf(0.1f * sqrt((*var)[index2d])))
                novelty->at<uint16_t>(i, j) = UINT16_MAX;
            else
                novelty->at<uint16_t>(i, j) = 0;

            last_prob[index2d] = prob;
        }
    }

    dstate->t++;
}
```

Listing 6.11: Procedure for obtaining and adding bounding boxes around clusters in the voxel grid

```
std::vector<Cube> voxelgrid_clusterize(VoxelGrid* grid) {

    std::vector<Eigen::Vector3f> dataset = voxelgrid_get_occupied(grid)
    std::vector<Cube> out;

    static const Eigen::Vector3f MAX_BOX_LENGTH = Eigen::Vector3f::Ones
    std::vector<Eigen::Vector3f> min, len;
    clustering_get_boxes(&dataset, &min, &len, MAX_BOX_LENGTH);

```

```
10      static const Eigen::Vector3f MIN_BOX_LENGTH = Eigen::Vector3f::Ones
11
12      for (uint32_t i = 0; i < min.size(); i++) {
13
14          // Scale for rendering
15          Eigen::Vector3f min_adj = min[i] * 0.5f;
16          Eigen::Vector3f len_adj = len[i] * 0.5f;
17
18          Cube cube = cube_create(min_adj[0], min_adj[1],
19              min_adj[2], glm::vec3(0.3f, 8.0f, 0.0f),
20              len_adj[0], len_adj[1], len_adj[2]);
21
22          voxelgrid_add_bounding_box(grid, cube);
23          out.push_back(cube);
24      }
25      return out;
26  }
```

Listing 6.12: Recursive traversal of dendrogram in search for largest bounding box under given threshold

```
1   static void clustering_get_largest_under_threshold(
2       uint32_t dendrogram_index, alglib::ahcreport* rep,
3       std::vector<Eigen::Vector3f>* dataset,
4       Eigen::Vector3f box_threshold,
5       std::vector<Eigen::Vector3f>* mins_out,
6       std::vector<Eigen::Vector3f>* lens_out) {
7
8       std::vector<Eigen::Vector3f> cluster_points =
9           clustering_get_points_by_dendro_index(dendrogram_index,
10                                                rep, dataset);
11
12      Eigen::Vector3f min, max;
13      clustering_get_min_max(&cluster_points, &min, &max);
14      Eigen::Vector3f length = max - min + Eigen::Vector3f::Ones();
15
16      if (length[0] > box_threshold[0] ||
```

```
17          length[1] > box_threshold[1] ||
18          length[2] > box_threshold[2]) {
19
20          clustering_get_largest_under_threshold(
21              rep->z[dendrogram_index - dataset->size()][0],
22              rep, dataset, box_threshold,
23              mins_out, lens_out);
24
25          clustering_get_largest_under_threshold(
26              rep->z[dendrogram_index - dataset->size()][1],
27              rep, dataset, box_threshold, mins_out,
28              lens_out);
29      }
30      else {
31
32          mins_out->push_back(min);
33          lens_out->push_back(length);
34      }
35  }
```

Listing 6.13: Procedure for generating a set of n frustums around the given Cube with the given viewing frustum

```
1   std::vector<Frustum> viewpoint_generate(Cube cube,
2       uint16_t n_viewpoints, Frustum* sensor_frustum) {
3
4       std::vector<Frustum> out;
5       glm::vec3 cube_center = cube.pos + cube.length / 2.0f;
6
7       std::vector<glm::mat4> dummy_poses =
8           viewpoint_distribute_frustums(
9           std::vector<float>(n_viewpoints, 0.0f), cube_center);
10
11      std::vector<float> adjusted_frustum_dist;
12
13      for (uint32_t i = 0; i < dummy_poses.size(); i++) {
14
```

```
15          glm::vec3 cube_corners[8];
16
17          glm::mat4 inv_model = glm::inverse(dummy_poses[i]);
18
19          for (uint8_t x = 0; x < 2; x++) {
20              for (uint8_t y = 0; y < 2; y++) {
21                  for (uint8_t z = 0; z < 2; z++) {
22
23                      uint8_t index = x + 2 * (y + 2 * z);
24                      cube_corners[index] = glm::vec3(inv_model *
25                          glm::vec4(cube.pos +
26                          glm::vec3(x, y, z) *
27                          cube.length, 1.0f));
28                  }
29              }
30          }
31
32          float z_adj = frustum_get_z_containing_points(
33              sensor_frustum, cube_corners, 8);
34
35          if (z_adj < sensor_frustum->farz) {
36
37              z_adj = fmaxf(z_adj, sensor_frustum->nearz);
38              adjusted_frustum_dist.push_back(z_adj);
39          }
40      }
41
42      std::vector<glm::mat4> frustum_poses = viewpoint_distribute_frustum
43          adjusted_frustum_dist, cube_center);
44
45      for (uint32_t i = 0; i < frustum_poses.size(); i++) {
46
47          Frustum frustum = *sensor_frustum;
48          frustum.model_matrix = frustum_poses[i];
49
50          out.push_back(frustum);
```

```
51        }
52
53        return out;
54    }
```