Multi-Sensor Fusion for the Mason Framework

Bachelor Thesis by Marius Schnaubelt, Department of Computational Engineering
April 28, 2016
Supervisors: Prof. Dr. Oskar von Stryk, M.Sc. Alexander Stumpf,
Simulation, Systems Optimization and Robotics Group, Department of Computer Science

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Multi-Sensor Fusion for the Mason Framework
Bachelor-Arbeit
Eingereicht von Marius Schnaubelt
Tag der Einreichung: 28. April 2016

Gutachter: Prof. Dr. Oskar von Stryk
Betreuer: M.Sc. Alexander Stumpf

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Simulation, Systemoptimierung und Robotik (SIM)
Prof. Dr. Oskar von Stryk

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelor-Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 28. April 2016

Marius Schnaubelt

Abstract

Nowadays, mobile robots are equipped with combinations of complementary sensor systems enabling the robot to perceive its surrounding environment. These sensor systems can for example be stereo vision systems, RGB-D Cameras and 3D or spinning 2D laser scanners, each providing different capabilities for environment sensing. A sufficient world model estimation is crucial for the robot's ability to perform complex tasks such as footstep planning, collision avoidance, path planning or manipulation.
In this thesis, the sensor fusion capability of the new sensor fusion framework Mason is developed. Mason is designed to be deployed on multi-sensor systems and is capable to fuse measurements from an arbitrary number of sensors in order to provide accurate and dense world models. In order to gain flexibility, the framework supports loading shared libraries during runtime to add functionality to the framework dynamically. For sensor fusion, the spatially hashed truncated signed distance function (TSDF) was chosen, as it only stores curvatures of the environment and therefore reduces computational and memory consumption. The presented work is based on the OpenCHISEL library that was improved and integrated into Mason. The thesis investigates how to combine multiple local TSDF estimations from different sensors to a global TSDF representation. Afterwards, we demonstrate how different world model representations are created based on the TSDF data, for example an elevation map, tested in simulation with the multi-sensor head of the THOR-MANG humanoid robot.

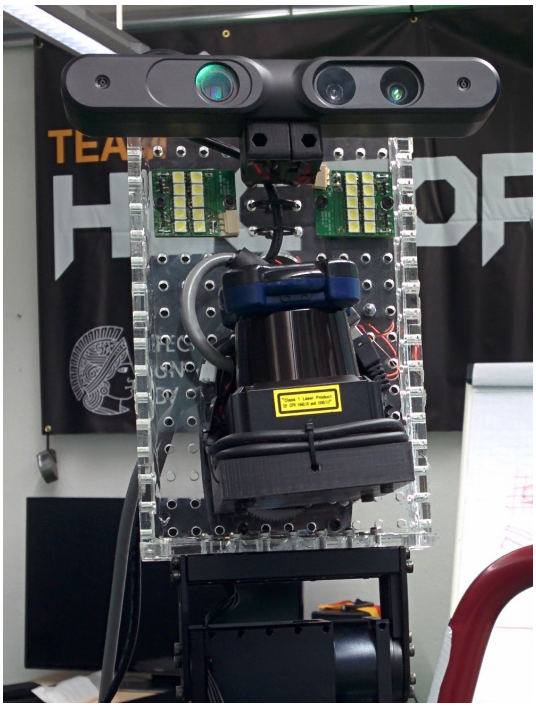
Kurzzusammenfassung

Heutzutage sind mobile Roboter mit einer Kombination aus verschiedenen, sich ergänzenden Sensorsystemen ausgestattet, die eine Wahrnehmung der Roboterumgebung ermöglichen. Diese Sensorsysteme, z.B. Stereo-Kamera-Systeme, RGB-D-Kameras oder 3D bzw. rotierende 2D Laserscanner, stellen verschiedene Eigenschaften für die Erstellung eines Weltmodells zur Verfügung. Eine hohe Genauigkeit des Weltmodells ist essentiell für verschiedene Fähigkeiten des Roboters, darunter Fußschrittplanung, Kollisionsvermeidung, Pfadplanung oder Manipulationsaufgaben.
Diese Bachelorthesis beschäftigt sich mit der Entwicklung der Sensorfusions-Fähigkeit des neuen Sensorfusions-Frameworks Mason. Mason wurde für den Einsatz in Multi-Sensorsystemen entwickelt und unterstützt die Fusion einer beliebiger Anzahl von Distanzsensoren, um ein genaues und dichtes Weltmodell bereitzustellen. Für die Flexibilität des Frameworks unterstützt jenes das dynamische Laden von „shared libraries“, wodurch das Hinzufügen von Komponenten während der Laufzeit ermöglicht wird. Die Fusion von Sensorinformationen wird mithilfe des „spatially hashed truncated signed distance function (TSDF)“ Ansatzes gelöst, was unnötige Prozessor- und Speicherauslastung vermeidet, da nur die Teile der Szene gespeichert werden, welche Oberflächeninformationen beinhalten. In der vorliegenden Arbeit wird die verbesserte OpenCHISEL Bibliothek genutzt und in Mason integriert.
Diese Bachelorthesis erforscht, wie sich mehrere lokale TSDF-Schätzungen von verschiedenen Sensoren zu einer globalen TSDF-Repräsentation fusionieren lassen. Anschließend wird die Erstellung von Weltmodellen, z.B. einer Höhenkarte, basierend auf den TSDF-Daten vorgestellt und letztlich in Simulation mit dem Multi-Sensor-Kopf des humanoiden THOR-MANG Roboters getestet.

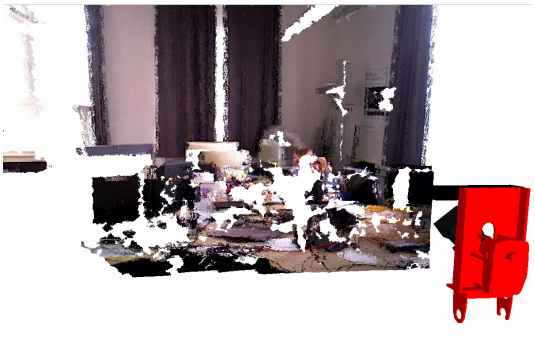
Contents

1 Introduction

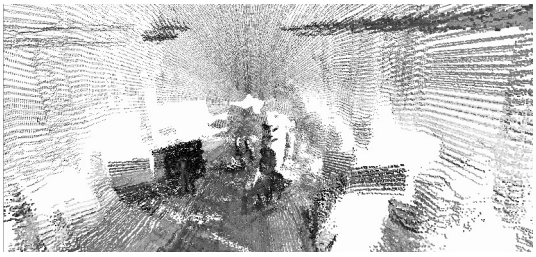
Nowadays, mobile robots are equipped with combinations of different sensor systems tailored to the planned tasks and occurring environments of the robot. These sensor systems, enabling the robot to perceive it's surrounding environment, could be stereo vision systems, RGB-D Cameras and 3D or spinning 2D laser scanners.
All these sensors have different capabilities which means different strengths and weaknesses, that have to be considered.



(a) The sensor head



(b) Data from the RGB-D camera



(c) Data from the rotating laser scanner

Figure 1.1: THOR-MANG's sensor head and the associated sensor data

The sensor head of the THOR-MANG humanoid robot for example contains two sensors; a RGB-D camera and a continuous rotating 2D laser scanner and is shown in figure 1.1.
The RGB-D camera provides dense colored images including depth information at a frame rate of 30 Hz while being affordable. Thus, the RGB-D camera enables capturing smooth movements and the color data (figure 1.1b) serves as valuable information for humans controlling the robot, as the human being is accustomed to colored perception. The sensor measurements of the RGB-D camera are very noisy and offer a low range of only 4.5 m.
Whereas the rotating laser scanner offers low-noise depth measurements with a large field of view, a sphere with a 30 m radius with the quarter part of a sphere missing. These measurements don't provide color information and the sparse data (figure 1.1a) is updated with a low rate.
Based on the different sensor systems, the environment of the robot has to be represented using the sensor information and considering different sensor capabilities.

1.1 Motivation

In order to build a world model based on different sensors, a sensor fusion framework is desirable. Such a framework should be able to combine the strengths of different sensors and compensate the weaknesses to achieve a good environment modeling.
By combining the information of different sensor systems, one sensor can confirm the measurements of another sensor. This way it is possible to check if all sensors work correctly and if a sensor malfunction is detected, the framework can react, for example by ignoring the sensor data of the broken sensor.
Furthermore, the fusion of multiple sensors can increase the reliability of the robot's mapping by compensating the failure of a sensor. The framework could simply use the remaining sensors to provide an updated world model.

A good world model is crucial for the robot's ability to perform various tasks. These tasks, among others, include footstep planning [1], collision avoidance, path planning and manipulation tasks [2, 3]. Only if the robot can locate itself and objects well, such kind of tasks can succeed.
In case of footstep planning, failing to model the floor or obstacles properly might cause the robot to fall when performing the planned footsteps based on a wrong world model. Since a robot fall often implies damage to the robot, it is mandatory to prevent such scenarios.
Furthermore, a good three dimensional world model, ideally containing color information, is able to provide improved immersion when controlling the robot in tele-operation mode[4]. This use case is common for rescue scenarios without direct sight.

1.2 Requirements

A good sensor fusion framework should be able to reconstruct the real world from an arbitrary number of sensors of any type.
Based on the world model, the framework has to generate different representations of the environment, for example an elevation map for path planning or a normal surface estimation for footstep planning.
The updates of the world model should be in "real-time". Here, it means the update frequency for a sensor configuration composed of two sensors, for example the sensor head of THOR-MANG, should be at least 1 Hz. This enables the robot to response to changes in the environment in an appropriate amount of time. The framework should model the environment while having a low memory- and computational power-consumption in order to prevent slowing down the other software components of the robot.
As robots might have different sensor configurations, the system should work with different sensor configurations on different robots and thus needs to be easily adjustable.
The framework should also provide an integration into the Robot Operating System (ROS) framework.

2 Foundations

In this chapter, we explain the foundations needed to understand this thesis.
At first we introduce the truncated signed distance function (TSDF) which is used to incrementally integrate sensor measurements into a data structure while providing certainty information.
Afterwards, we present different possibilities to update the TSDF data and how to improve the surface reconstruction by using space carving.
Then, a more efficient way to store and maintain a TSDF representation, the spatially hashed TSDF, is introduced and two important algorithms to work with the TSDF representation are explained.
Finally, there is a short introduction into modeling sensor noise.

2.1 Truncated Signed Distance Function

The truncated signed distance function (TSDF) [5] is a way to represent a three-dimensional scene, usually in a discrete voxel grid. For every voxel in the scene, a signed distance function (SDF) $\Phi : \mathbf{R}^3 \rightarrow \mathbf{R}$ and the weight $W : \mathbf{R} \rightarrow \mathbf{R}$ are stored. $\Phi(\mathbf{x})$ describes the distance of a voxel $\mathbf{x} = [\,x\ y\ z\,]^\top$ to the closest surface of an object. When the voxel $\mathbf{x}$ is inside the object, $\Phi(\mathbf{x})$ is negative; when being on the surface of the object, $\Phi(\mathbf{x})$ is 0, otherwise $\Phi(\mathbf{x})$ becomes positive, as shown in image 2.1

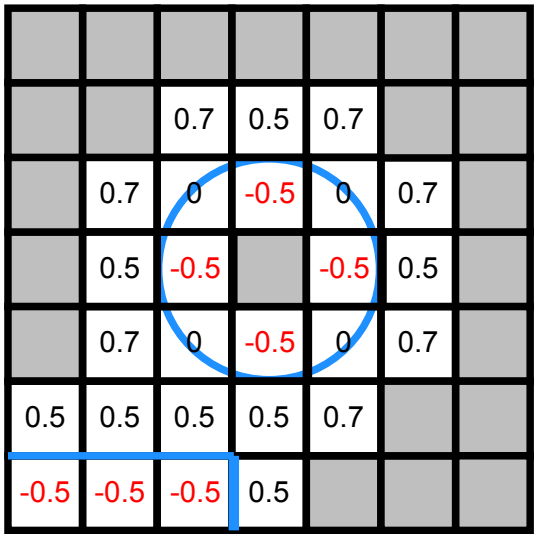


Figure 2.1: A two-dimensional truncated signed distance function (TSDF) voxel grid. It stores the value of the TSDF measured from the cell center to the closest surface marked by blue shapes. The undefined cells are marked with gray color.

Thus, the TSDF implicitly describes the surface of objects when following the isocontour ($\Phi(\mathbf{x}) = 0$).
The SDF Φ_τ gets truncated when the distance to the surface is bigger than a specific threshold $\tau \in \mathbf{R}$, which is called truncation distance [6]:

$$\Phi_\tau(\mathbf{x}) = \begin{cases} \Phi(\mathbf{x}), & \text{if } |\Phi(\mathbf{x})| < \tau \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{2.1}$$

The weight assigned to each voxel represents the certainty of the determined surface distances.
The TSDF is able to use all given distance measurements, redundant observations are used to improve the surface quality by reducing sensor noise. A scene representation is already available after a few measurements due to the incremental update capability.
Today, TSDF is often used because it is able to deliver very good surface reconstructions. A downside of the method is the high memory consumption and the requirements in terms of computing power to use TSDF for scene reconstruction in real-time. Therefore, Graphics Processing Unit (GPU) acceleration (for example realized with KinectFusion [7]) is often needed for real time applications of TSDF.

2.1.1 Update of TSDF Data

There exist different ways to update the TSDF data contained in the discrete grid, differing in terms of memory consumption and reconstruction quality.
We denote a new surface distance measurement $\Phi_k(\mathbf{x})$ given by a sensor and the currently stored TSDF estimation $\Phi_\tau(\mathbf{x})$ with respect to the truncation distance τ.

Simple Update

The easiest method to update the TSDF data is taking a linear combination of the new measurement $\Phi_k(\mathbf{x})$ and the currently stored TSDF estimation $\Phi_\tau(\mathbf{x})$ at a voxel position $\mathbf{x}$.
The speed of the update is controlled by a weight $w_k(\mathbf{x}) \in (0,1)$:

Algorithm 1 Simple Update

1: **if** $\Phi_\tau(\mathbf{x}) =$ undefined **then** ▷ If voxel is empty
2: $\quad \Phi_\tau(\mathbf{x}) \leftarrow \Phi_k(\mathbf{x})$
3: **else**
4: $\quad \Phi_\tau(\mathbf{x}) \leftarrow (1 - w_k(\mathbf{x}))\Phi_\tau(\mathbf{x}) + w_k(\mathbf{x})\Phi_k(\mathbf{x})$ ▷ Update surface distance with a linear combination
5: **end if**
6: **if** $|\Phi_\tau(\mathbf{x})| > \tau$ **then** ▷ If magnitude of surface distance is greater than truncation distance
7: $\quad \Phi_\tau(\mathbf{x}) \leftarrow$ undefined ▷ Reset voxel
8: **end if**

A low $w_k(\mathbf{x})$, e.g. $w_k(\mathbf{x}) = 0.01$, created very good surface quality in experiments of Trifonov [8], but needed some time to accumulate enough measurements.

Weighted Update

In order to wipe out the issue of choosing the right weight $w_k(\mathbf{x})$, it is possible to also store the accumulated weight function $W(\mathbf{x})$ of the point $\mathbf{x}$, shown in algorithm 2.
When accumulating the weight, at the beginning the TSDF value converges faster to the final value in a static scene [8], but both methods converge to the same value. By setting a limit of the accumulated weight W_{max}, the scanning of dynamic environments is made possible, otherwise new measurements might get neglected by a too high $W(\mathbf{x})$.
As also a weight has to be stored aside the TSDF, the memory consumption is doubled in comparison to the simple update.

Algorithm 2 Weighted Update

if $\Phi_\tau(\mathbf{x}) = \text{undefined}$ **then** ▷ If voxel is empty
  $\Phi_\tau(\mathbf{x}) \leftarrow \Phi_k(\mathbf{x})$
  $W(\mathbf{x}) \leftarrow w_k(\mathbf{x})$
else
  $\Phi_\tau(\mathbf{x}) \leftarrow \frac{W(\mathbf{x})\Phi_\tau(\mathbf{x}) + w_k(\mathbf{x})\Phi_k(\mathbf{x})}{W(\mathbf{x}) + w_k(\mathbf{x})}$ ▷ Update surface distance with a weighted update
  $W(\mathbf{x}) \leftarrow \min(W(\mathbf{x}) + w_k(\mathbf{x}), W_{max})$ ▷ Update the limited accumulated weight
end if
if $|\Phi_\tau(\mathbf{x})| > \tau$ **then** ▷ If magnitude of surface distance is greater than truncation distance
  $\Phi_\tau(\mathbf{x}) \leftarrow \text{undefined}$ ▷ Reset voxel
end if

Kalman-Filter Update

Even if the weighted update yields better results than the simple update, both update methods can create poorer surface reconstructions when integrating many low quality measurements, for example along the edges of an object.
Therefore, Trifonov [8] implemented a Kalman filter to update the TSDF data. As state variable of the Kalman filter the truncated signed distance (TSD) value $\Phi_\tau(\mathbf{x})$ of a point $\mathbf{x}$ is used and the noisy measurement to be fused is $\Phi_k(\mathbf{x})$. Now, the estimated process variance $P(\mathbf{x})$ is stored for every point and every point gets an estimated variance $p_k(\mathbf{x})$ instead of the weights $w_k(\mathbf{x})$ and $W(\mathbf{x})$. The process variance Q is used to adjust response and smoothing and K denotes the Kalman gain.

Algorithm 3 Update using a Kalman Filter

if $\Phi_\tau(\mathbf{x}) = \text{undefined}$ **then** ▷ If voxel is empty
  $\Phi_\tau(\mathbf{x}) \leftarrow \Phi_k(\mathbf{x})$
  $P(\mathbf{x}) \leftarrow p_k(\mathbf{x})$
else
  $P(\mathbf{x}) \leftarrow P(\mathbf{x}) + Q$ ▷ Prediction Step
  $K \leftarrow P(\mathbf{x})/(P(\mathbf{x}) + p_k(\mathbf{x}))$ ▷ Correction Step
  $\Phi_\tau(\mathbf{x}) \leftarrow \Phi_\tau(\mathbf{x}) + K(\Phi_k(\mathbf{x}) - \Phi_\tau(\mathbf{x}))$ ▷ Update surface distance using the Kalman gain
  $P(\mathbf{x}) \leftarrow (1-K)P(\mathbf{x})$
end if
if $|\Phi_\tau(\mathbf{x})| > \tau$ **then** ▷ If magnitude of surface distance is greater than truncation distance
  $\Phi_\tau(\mathbf{x}) \leftarrow \text{undefined}$ ▷ Reset voxel
end if

With the help of the Kalman filter, the surface estimation quality is not affected by noisy measurements as high variances lower the Kalman gain K and therefore have a lower influence on the estimations of $\Phi_\tau(\mathbf{x})$ and $P(\mathbf{x})$. Trifonov [8] found out that the Kalman filter update method handles balancing between the reconstruction quality versus the response speed with changing scenes slightly better than the weighting schemes.

2.1.2 Space Carving

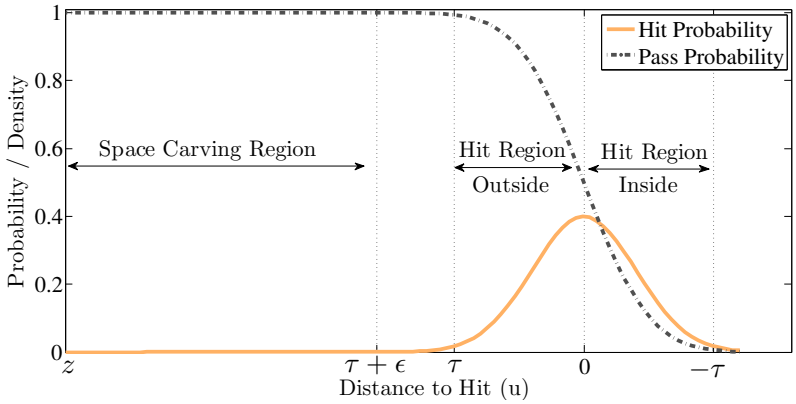


Figure 2.2: The space carving and hit regions [6]

Space carving is used to improve the surface reconstruction by marking voxels as empty that fulfill special conditions. This improvement is especially noticeable in the area of object-edges and the representation errors caused by moving obstacles in the reconstruction are removed.
When doing space carving, the line of sight of a sensor ray is followed back from the observed surface and all the voxels along the ray are marked as free [5]. When also considering the truncation distance, all voxels with a distance to the surface $u \in [-\tau, \tau]$, i.e. the voxels in the hit region (see figure 2.2), are updated as they have a very high hit probability [6]. All the voxels with a surface distance $u > \tau + \epsilon$, where τ is the truncation distance and ϵ is a constant offset, are carved – which means that they are removed.

2.2 Spatially Hashed TSDF

When using a regular voxel grid for storing the TSDF data – both empty and occupied space – the memory consumption is very high even though most of the scene is empty. This means that only small scenes can get reconstructed using high resolutions. When reconstructing larger scenes the quality has to be reduced due to memory limitations of the GPU.
Therefore, Nießner et al. [9] introduced a two-level data structure (shown in figure 2.3) based on spatial hashing which uses the fact that the TSDF structure is sparse.

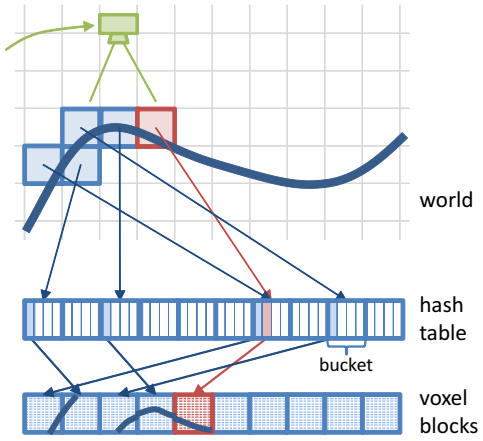


Figure 2.3: Splitting the recorded scene into voxel blocks containing a regular voxel grid [9]

A hash table stores pointers to voxel blocks – also called chunks – which contain a regular voxel-grid of $N_x \cdot N_y \cdot N_z$ voxels. To integrate a new voxel block into the hash table, the rounded integer world position (x, y, z) of the block gets mapped to a hash value $H(x, y, z)$ using the hashing function:

$$H(x, y, z) = (x \cdot p_1 \oplus y \cdot p_2 \oplus z \cdot p_3) \bmod n \tag{2.2}$$

where p_1, p_2, and p_3 are large prime numbers (e.g. 73856093, 19349663, 83492791 were used by Teschner et al. [10]), n is the hash table size and $\oplus$ is the exclusive or operation.
Only voxel blocks containing valid TSDF data, this means voxels with a weight $W > 0$, get allocated and therefore less memory is wasted to store empty space.
When retrieving a voxel block from a given world position, the position gets firstly rounded to the next integer position and then hashed. The hash table lookup returns a pointer to the array containing the contiguously stored voxels of the voxel block. When using one bucket per hash entry, a hash map can be used as well. The performance of a voxel lookup is therefore $\mathcal{O}(1)$ [6].
When a voxel block is allocated but does not contain data anymore, e.g. because of moving objects in the scene, the garbage collection is deployed. The garbage collection recognizes unneeded voxel blocks by checking if the maximum weight in the voxel block is zero or if the minimum TSDF is larger than a given threshold. In case of finding unneeded voxel blocks, the voxel block and the hash entries get deleted.

2.3 Ray casting

Ray casting is an algorithm used to traverse a 3D space by following a ray constructed by a given start and end point. Casting rays in a regular grid enables the detection which voxels are visited by the ray. In the case of spatially hashed TSDF, ray casting is used to determine which chunks (voxel blocks) shall be updated by sensor measurements. The fast voxel traversal algorithm [11] always goes into the direction of the closest chunk boundary and therefore detects the affected chunks efficiently.

2.4 Marching Cubes

The Marching cubes algorithm [12] is used to render surfaces given a voxel grid. Using a divide-and-conquer approach the grid is split up into cubes of 8 neighboring voxels, one at each corner. For each cube, the mesh defined by the 8 corner values is searched, thus there are $2^8 = 256$ possibilities for triangulating the cube. When also considering the symmetry, only 15 cases remain. These cases are shown in figure 2.4.

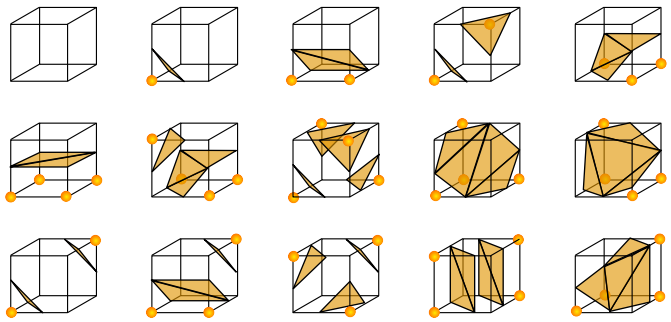


Figure 2.4: Triangulated cubes when symmetry is considered [13]

Marching cubes is well suited for the case of spatially hashed TSDFs as it uses single cubes and therefore exploits the sparsity of the scene, as Trifonov [8] stated.

2.5 Sensor Noise Modeling

A perfect depth sensor, for example a pinhole depth camera, emits rays from the sensor center **c**. Each ray hits onto a surface at the endpoint **x** of the ray. The measured distance $d = ||\mathbf{o} - \mathbf{x}||$ is the euclidean distance between the sensor center and the surface.
In real world, sensor measurements are noisy, therefore the rays are disturbed by noise.
The measured distance z of an imperfect sensor is modeled:

$$z = d + \epsilon \tag{2.3}$$

where the noise ϵ is assumed as Gaussian distribution:

$$\epsilon \sim \mathcal{N}(d \mid 0, \sigma_d^2) = \frac{1}{\sigma_d \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{d}{\sigma_d}\right)^2} \tag{2.4}$$

with zero mean and the standard deviation σ_d.

In order to model the sensor noise, we have to determine the standard deviations of the sensors we use in the system. This is done exemplary for the sensors of the experimental platform, whose standard deviations are shown in figure 2.5.

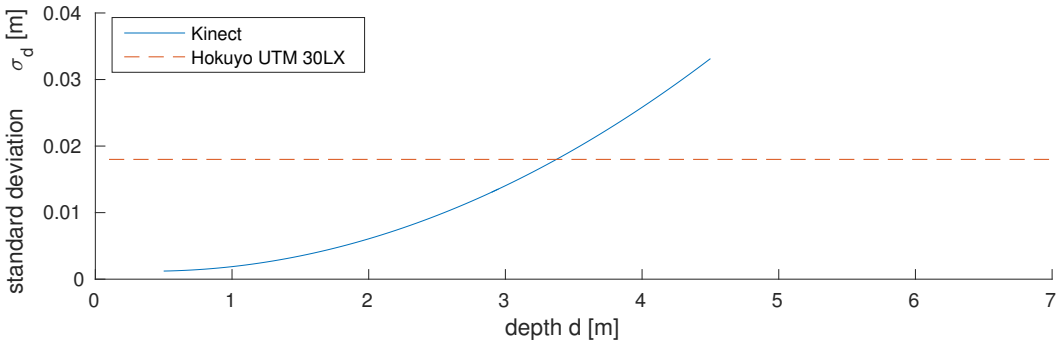


Figure 2.5: The standard deviation of the experimental platform's sensors

2.5.1 Noise Model for RGB-D Cameras

Nguyen et al. [14] researched the axial noise distribution as a function of the measured distance for the Microsoft Kinect sensor, which is comparable to the RGB-D camera used in the experimental platform (see section 3.1). The experiments showed that the axial noise increases quadratically with the measured distance:

$$\sigma_d = 0.0012 + 0.0019(d - 0.4)^2. \tag{2.5}$$

2.5.2 Noise Model for Laser Scanners

Pomerleau et al. [15] investigated the noise characterization of laser scanners, including a Hokuyo UTM-30LX comparable to the Hokuyo UTM-30LX-EW sensor of the experimental platform (see section 3.1). The laser scanner showed a constant standard deviation σ_d independent of the measured depth:

$$\sigma_d = 0.018. \tag{2.6}$$

3 Experimental Platform

In the following chapter, the used simulation environment and the humanoid robot that has been used to test and evaluate the framework and also the most important used libraries are presented.

3.1 Robot

The Simulation, Systems Optimization and Robotics Group is currently working with an electrically actuated, humanoid robot. This robot, called Johnny, is based on the THOR-MANG platform created by ROBOTIS. Johnny offers 36 degrees of freedom actuated with standardized servo motors and weights only around 50 kg while being 1.47 m tall. The robot is equipped with custom-designed hands developed at Virginia Tech University and has an upgraded mainboard that provides an Intel Core i7-4800MQ mobile quad-core processor.

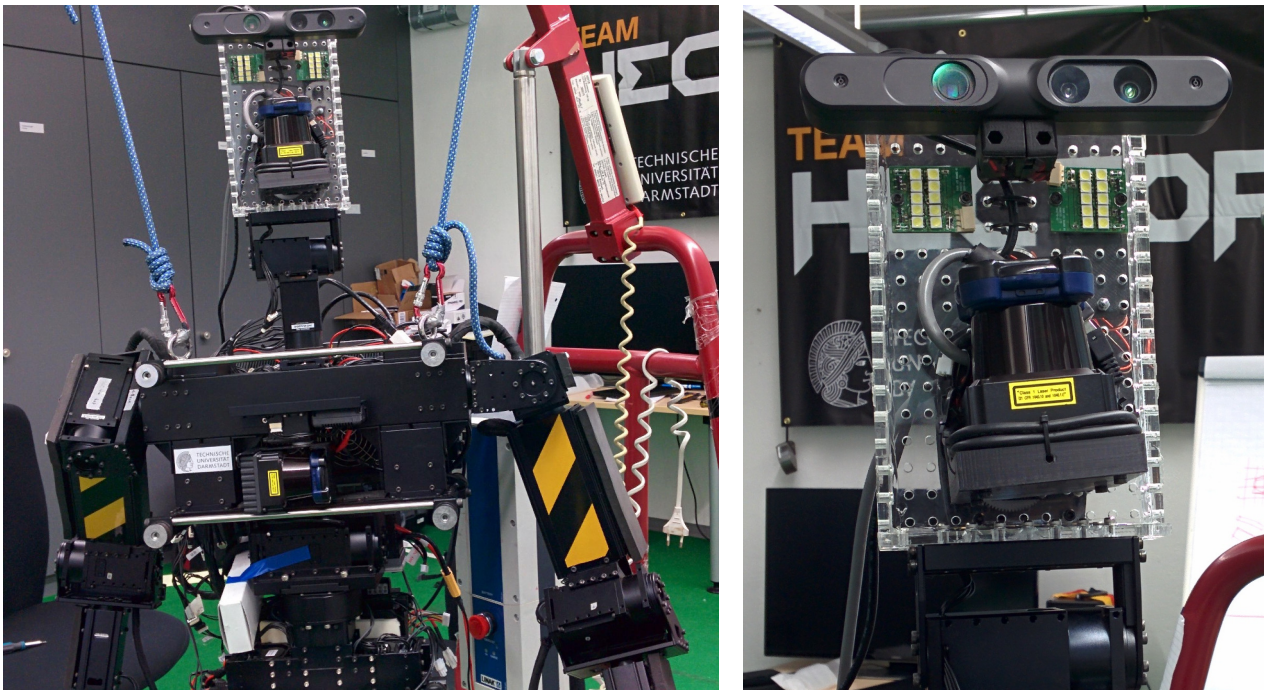


Figure 3.1: The THOR-MANG humanoid robot

For perception, Johnny comes with a multi-sensor head and a panning Hokuyo UTM-30LX-EW laser scanner in the chest, displayed in figure 3.1. This multi-sensor head was previously developed by the author of this thesis during an integrated project supervised by the Simulation, Systems Optimization and Robotics Group at Technische Universität Darmstadt. The multi-sensor head consists of a Asus Xtion Pro Live RGB-D camera and a continuous rotating Hokuyo UTM-30LX-EW laser scanner providing 3D scans with up to 30 m range.

3.2 Robot Operating System

The Robot Operating System (ROS) [16] is an open-source framework that aims at providing hardware abstraction to robot systems. A robot is defined by a Unified Robot Description Format (URDF), which contains masses, lengths, inertias et cetera.
The software is split up into multiple nodes, each running in an own process on the robot's computer. Inter-process communication is implemented by sending messages with predefined data-structures across the network of nodes. Messages are sent with a publisher, publishing to a specific topic, e.g. a laser scan. A single node or multiple nodes can receive the message by subscribing to this topic. The publisher and subscriber concept enables flexible communication between different nodes and keeps the written code independent from each other with the exception of the used messages.

3.3 Simulation

In order to be able to test new software prior to experimenting on the real robot, a simulated version of THOR-MANG exists. In this work, Gazebo 2, an open-source robotics simulator, is used for simulation. Gazebo supports the ROS framework and uses the URDF model to simulate the robot.
Gazebo is able to simulate the dynamics of the robot and also to generate sensor data including noise. Thus, we can use the same sensor fusion framework code to test in simulation and on the real robot. In figure 3.2, the simulated THOR-MANG robot is shown.

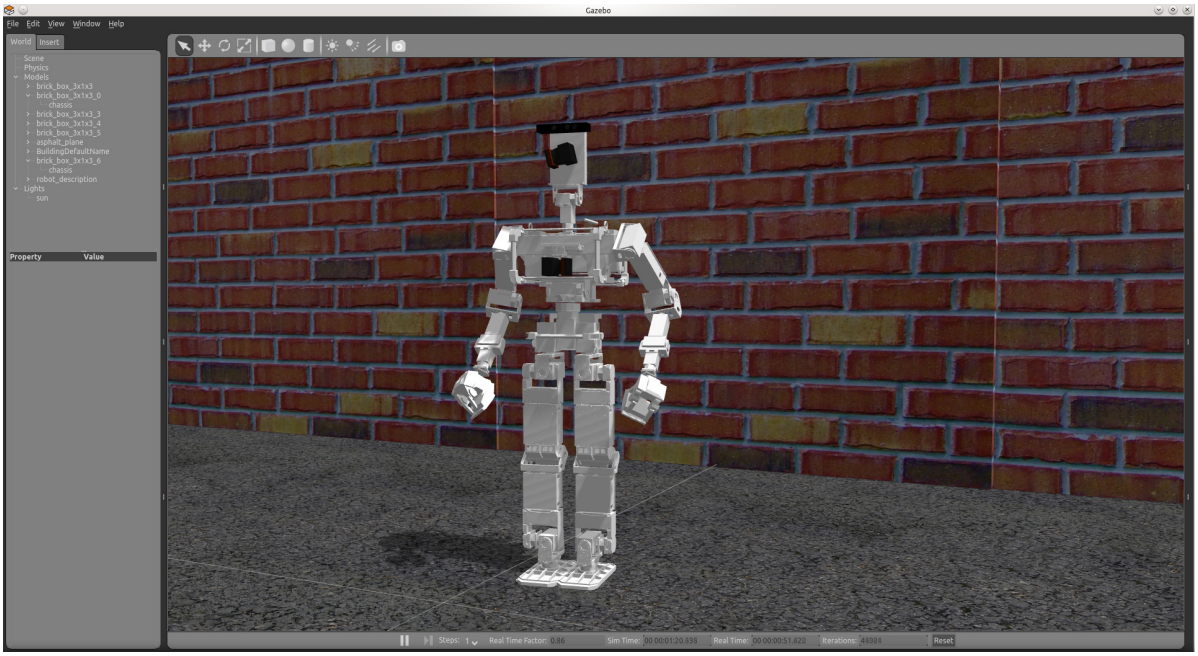


Figure 3.2: View of THOR-MANG simulated in Gazebo

3.4 Plugin Management System

To achieve a big flexibility of the framework, it will use the `vigir_pluginlib` plugin management system. The `vigir_pluginlib` [17] is an extension to the ROS pluginlib [18] that allows loading plugins dynamically. Plugins are C++ classes which are able to get loaded and unloaded from a runtime library, this means a dynamically linked library. By adding a plugin during runtime, the software gains new features or changes its behavior without needing to know the related library in advance. Therefore, user specific code can be executed while avoiding modifications to the original framework.

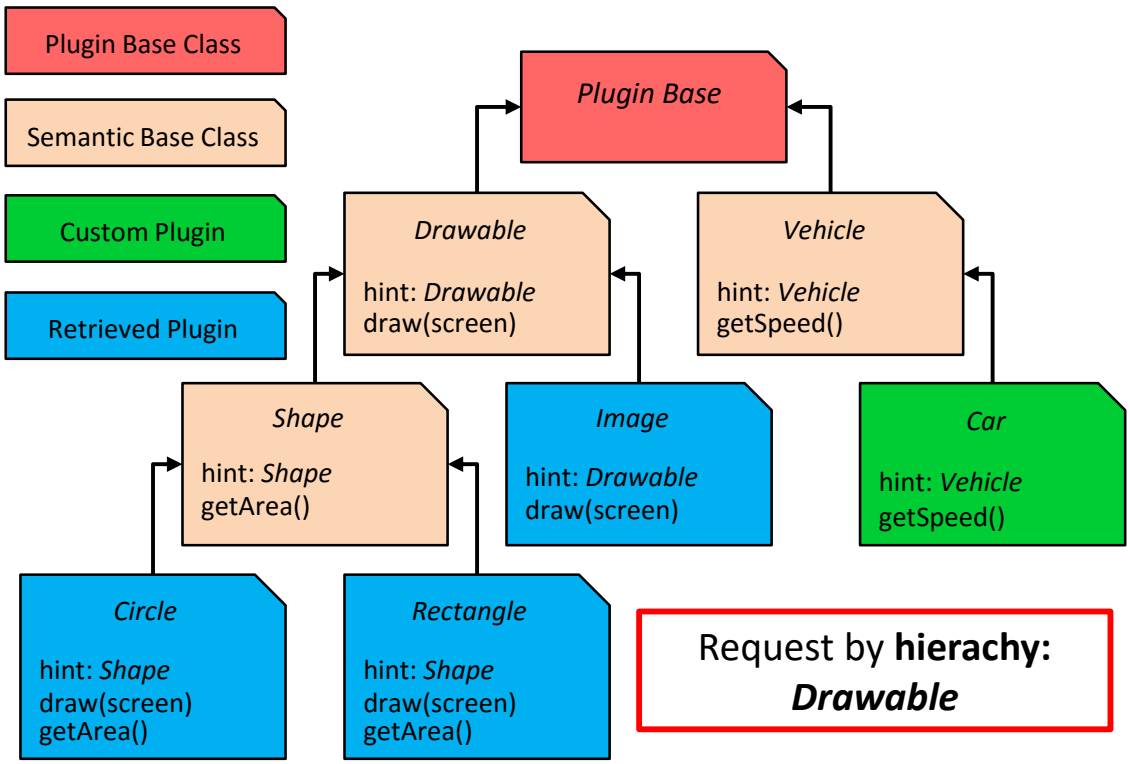


Figure 3.3: Example plugin inheritance hierarchy for the `vigir_pluginlib` [17]

The `vigir_pluginlib` adds a semantic plugin management consisting of semantic base classes and a plugin manager. A semantic base class is an abstract plugin defining the functionality and content for all derived plugins. The plugin manager maintains the plugin instances and is used for requesting plugin instances, for example based on the desired name or by the desired parent semantic base class (see figure 3.3).

4 Related Work

In this chapter an overview of relevant approaches for sensor fusion is given and evaluated for possible participations to the sensor fusion framework.
Furthermore, the current state of THOR-MANG's environment modeling is presented.

4.1 Continuous Humanoid Locomotion over Uneven Terrain Using Stereo Fusion

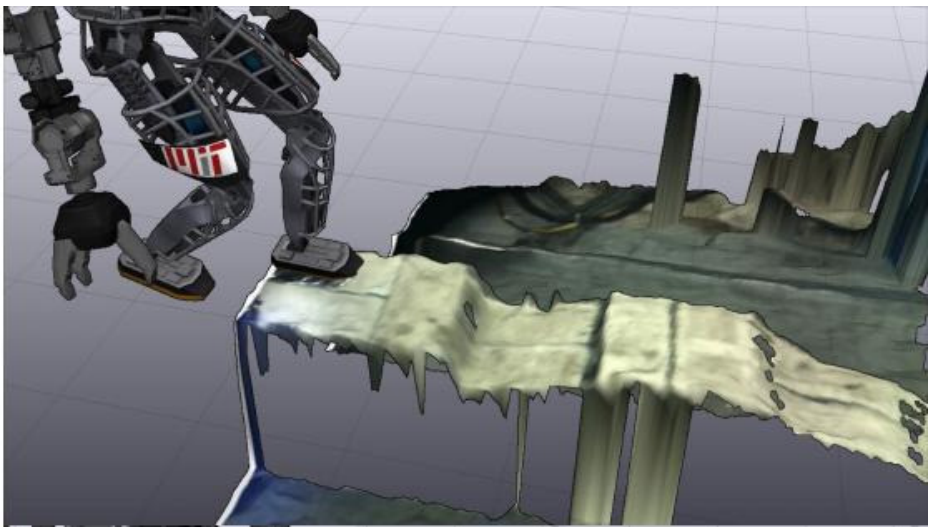


Figure 4.1: Locomotion based on Kintinous [19]

Team MIT's approach [19] for locomotion over uneven terrain, motivated by the Defense Advanced Research Projects Agency (DARPA) Robotics Challenge, contains a perception system to build a consistent 3D world model of the terrain which is shown in figure 4.1. The system is based on Kintinous [20], which is based on KinectFusion, a RGB-D data fusion system using TSDF to represent the recorded scene. The whole scene reconstruction is accelerated by using the Compute Unified Device Architecture (CUDA) and is maintained in GPU memory.
Since KinectFusion stores the data in GPU memory, the reconstructions are limited to smaller volumes (4-6m long edges) when using high voxel resolutions (about centimeters). To allow larger-scale mapping, Kintinous extends the KinectFusion algorithm with the ability that the mapped area can move over time. This moving window approach stores a moving voxel grid in the GPU. Areas leaving the field of view of the camera are meshed and the TSDF data is thrown away [6] to save memory. Loosing the TSDF data has the effect that motion planning or collision avoidance based directly on the distance field are no longer possible. Furthermore, a TSDF data fusion is prevented as well.
Team MIT adapted the Kintinous algorithm to create a real-time, high-quality fused 3D map by using stereo image data. The mapping is then used by the footstep planner for obstacle avoidance, foot placement and kinematic reachability. Because of the GPU utilization, the system is only suitable for robots with a NVIDIA GPU. The stereo vision system is the only distance measurement source, thus the system depends on sufficient lightning conditions.

4.2 Robot-Centric Elevation Mapping

Figure 4.2: Resulting elevation map [21]

The Robot-Centric Elevation Mapping software [21], created by the Autonomous Systems Lab from the ETH Zurich, is about elevation mapping for a mobile robot. It is designed for local navigation tasks with robots and based on a 2.5D grid map, also called *height map*.
The data storage is implemented as two-dimensional circular buffer which allows map repositioning (to follow the robot) without copying data in memory. With the aid of a Kalman filter the sensor measurements get fused with the height estimates and variances in each cell of the map. This way measurement uncertainties get modeled as well by using a Gaussian probability distribution. The system only supports one distance sensor and the provided elevation map is limited around the robot, as can be seen in figure 4.2. By using only the Central Processing Unit (CPU) to fuse the data, the ROS based software is usable on many robot systems.
The usage of a 2.5D grid map as world model enables high update rates, but has the drawback of being limited to one surface and not being able to model doors, bridges or tunnels. This restricts the possible application as sensor fusion framework, as many situations cannot get represented. For example, a multilevel building with a room under the stairs and different rooms upstairs. Especially for rescue robots, which were designed to work in disaster scenarios, this is a big shortcoming.

4.3 Obviously Framework

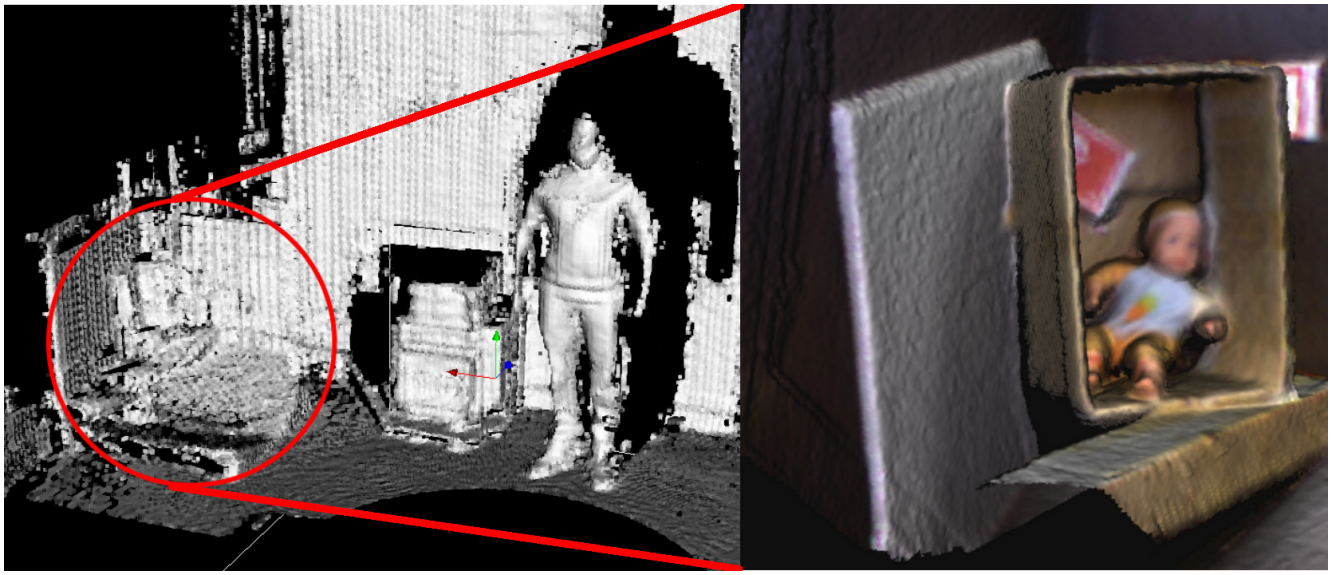


Figure 4.3: Scene created with a laser scanner and further inspected area of interest with an RGB-D camera [22]

The Obviously framework [22], created by S. May et al. at Nuremberg Institute of Technology, aims at integrating multi-sensor data for robot mapping. It generalizes the TSDF KinectFusion approach to support different sensor modalities. By default, Obviously supports two different sensor models which are used to map the data from the sensor measurement to the current sensor pose and vice versa. The first one is the pinhole camera model, which is sufficient for time of flight (ToF) cameras and RGB-D cameras. The second model is a polar model, which is used by rotating 2D laser scanners. New sensors can be added to the framework by implementing a fitting sensor model.
The world model, represented as TSDF data, only gets updated when a significant movement since last integration has been performed in order to safe computational resources. Before updating, the world model and the latest sensor measurement get aligned to each other using the Iterative Closest Point (ICP) algorithm. The framework is CPU-based and enables mapping in real-time, but there is no ROS integration. A possible use case of this framework is the exploration in rescue environments, where the large-scale scene is reconstructed using a 3D laser range finder. In order to inspect objects of interest, a higher resolution RGB-D camera can add further details of the scene; picture 4.3 shows an example of this use case.
Since Obviously extends the KinectFusion software to work without GPUs, it uses a fixed-size voxel-grid to store the TSDF data. For acceleration, the voxel-grid is divided into smaller partitions which contain a specific part of the voxel-grid. Only partitions traversed by the sensor rays get a TSD update, the other partitions remain untouched.
Nonetheless, the traversed partitions get set to the maximum TSD value and the weight is increased, which results in a big memory consumption. This can be a major drawback for the scalability of the framework when using many sensors.

4.4 Vigir Terrain Classifier

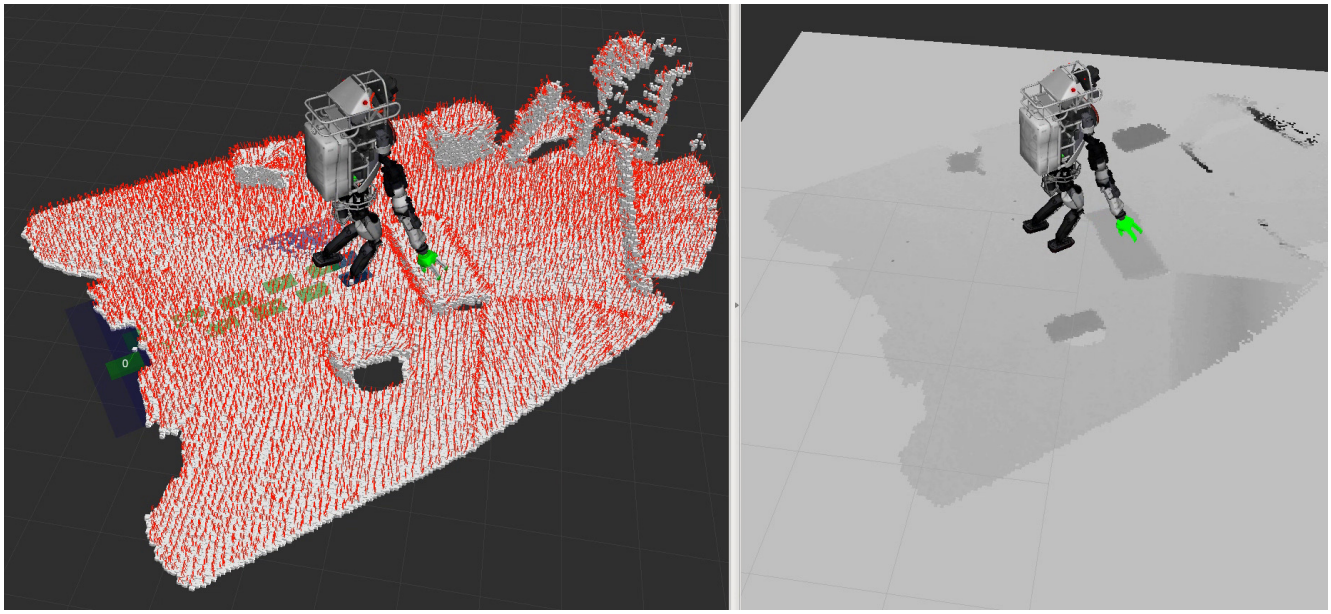


Figure 4.4: Height map and surface normal estimation created by Vigir Terrain Classifier [17]

For the DARPA Robotics Challenge, Team Vigir developed the Vigir Terrain Classifier. The system is currently used by THOR-MANG and offers an octree which stores accumulated and filtered point clouds. The octree representation discretizes the world into voxels, but stores only information about occupied voxels which makes it very memory efficient. Based on the stored measurements in the octree, height maps and surface normal estimations, shown in figure 4.4, are created and published.
In order to accelerate the update of the octree and the representations, like the height map, only the area of the latest laser scan is updated. By using an octree with accumulation of the laser scans, high update rates with low memory consumption are possible.
This approach doesn't store any certainty information about any measurements.
Furthermore, dynamic objects moving through the recorded scene remain in the world model, as neither a probabilistic model nor ray casts based on the laser scans are used. Therefore, the map can contain phantom obstacles which might complicate tasks relying on a good world representation.

4.5 CHISEL

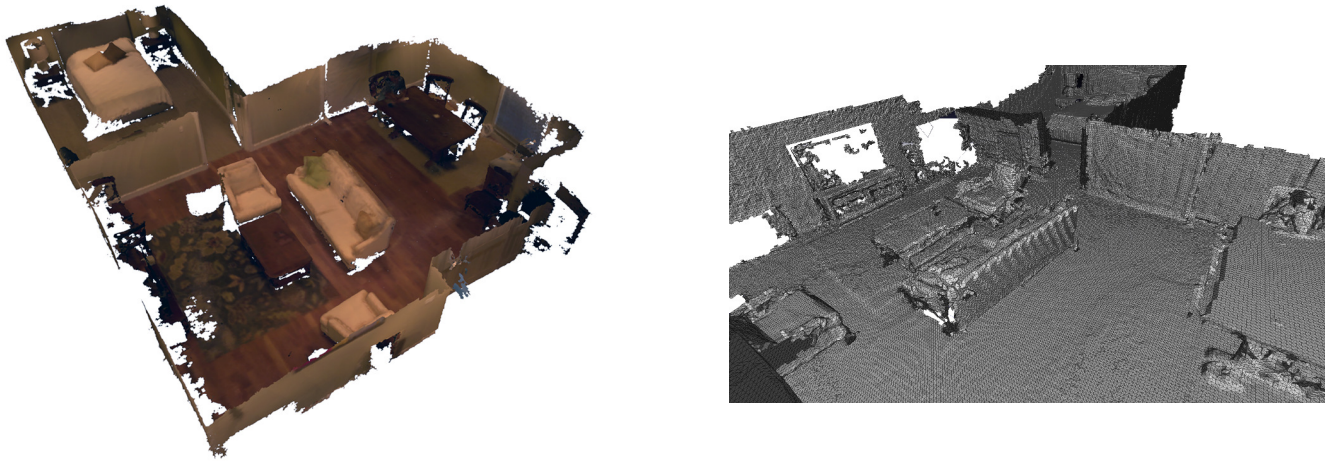


Figure 4.5: Apartment scene created by CHISEL [6]

CHISEL [6] is a library for real-time house-scale TSDF reconstruction on-board of a Google Tango device, a mobile device including a depth sensor, a color camera and an inertial measurement unit (IMU).
The purpose of CHISEL is real-time 3D mapping and localization while the device is moving around in the scene with a resolution of 2-3 cm. In figure 4.5, you can see an example is illustrated.
The computing power and memory requirements make the application of TSDF on most mobile devices unfeasible. By using a dynamic spatially hashed truncated signed distance field (see section 2.1) though, these limitations are removed. Klingensmith et al. [6] discovered in an experiment that the majority (about 93%) of the evaluated scene is empty. Therefore, saving and updating this empty space is not reasonable as it wastes memory and computing resources.
The spatially hashed TSDF exploits this fact by utilizing a two-level data structure which stores only chunks containing data. Hereby, chunks are cuboids containing a regular voxel-grid representing a small part of the scene. This concept enables real-time reconstruction of large-scale environments with mobile devices without using GPU acceleration.
Noisy sensor measurements are taken into account by dynamic truncation which adjusts the truncation distance in respect of the measured depth and the standard deviation of the depth noise for the given depth. By using space carving (see section 2.1.2), reconstruction errors caused by moving obstacles can be removed and the surface quality is improved. Projection mapping speeds up the integration of depth images by approximating ray casts. Every voxel center gets projected onto the depth image from where the depth value on the depth image gets compared with the displacement of the voxel center and the camera plane.
The provided open-source library OpenCHISEL – which is based on CHISEL – offers ROS integration but can only consider one sensor in the 3D scene reconstruction. The performance of the library is suitable for real-time sensor fusion of multiple sensors and the preserving of the TSDF data enables it to use the certainty of any place observed in the past.
As OpenCHISEL doesn't include any sensor pose estimation nor tracking, OpenCHISEL relies on an external pose estimation.

5 Concepts

In this chapter, we explain the design considerations for the framework and present the core components of the framework.

5.1 Design Considerations

The main goal of this thesis is to create a framework to model the environment in 3D based on an arbitrary number of sensors provided by a mobile robot. As the framework should be flexible and portable as much as possible, it makes use of the `vigir_pluginlib` which extends the plugin concept provided by the ROS framework.
Plugins allow the user to select the needed components adjusted to the current needs. This is even possible while the framework is running, as a compilation of the framework core code is not needed to add new components because plugins are dynamically loaded from shared libraries during runtime.
For example, the user of the framework might decide to replace a currently used sensor fusion plugin by another one using different parameters or even implementing another fusion approach to improve results. This is even possible while the robot is in action, as the framework can continue running while replacing the fusion plugins.
After fusing the measurement data of individual sensors, the framework should be able to fuse sensor data from different sensors to a single environment model and as well to generate different representations based on the captured environment modeling, for example a height grid map. This goal is also achieved by plugins named *generator plugins*.
In order to keep the computational costs low, the framework should be able to use incremental updates. The framework is implemented in C++ to enable efficient real time sensor data integration.

5.2 Mason Framework Design

In the following section, we will describe the components composing the Mason framework illustrated in figure 5.1. A major part of the framework's core components described below in this chapter were contributed by Alexander Stumpf at Simulation, Systems Optimization and Robotics Group.

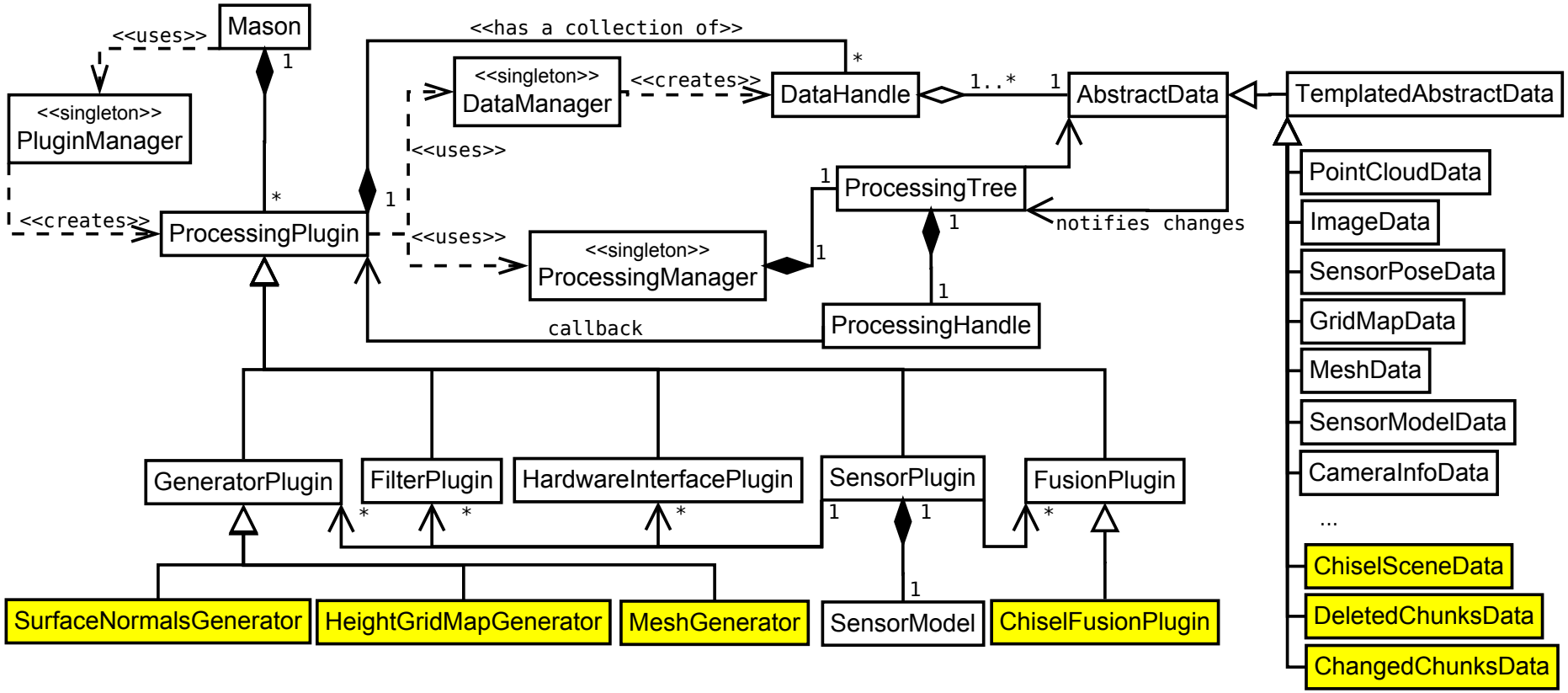


Figure 5.1: The framework concept consisting of the Mason core components and the first OpenCHISEL based plugins which are highlighted in yellow color

5.2.1 Mason Core Node

The Mason core node maintains all plugin instances inheriting from `ProcessingPlugin` which got created by the `PluginManager` implemented in the `vigir_pluginlib`. At the current implementation, these instances can be `SensorPlugin`, `FusionPlugin`, `GeneratorPlugin`, `FilterPlugin`, `HardwareInterfacePlugin` plugins which create, update or exchange `AbstractData` instances.

5.2.2 Abstract Data

The Mason Framework uses the `AbstractData` interface to exchange data between different plugins. `AbstractData` can also wrap different data types, e.g. the ROS data types which have no common interface by default. The `TemplatedAbstractData` simplifies the usage of the `AbstractData` interface, by accepting the stored object class as template argument, e.g. `TemplatedAbstractData<sensor_msgs::Image>` wraps a ROS image.

5.2.3 Processing Plugin

The `ProcessingPlugin` is the base for plugins in the Mason framework. It provides two modes for updating the plugin's data, an asynchronous mode and a synchronous mode when subscribed data has been changed. Each `ProcessingPlugin` uses the `DataManager` to register provided data or request data by using `DataHandles`. `DataHandles` provide access to the data of the framework; as long the `DataHandle` is alive, the access to the data is granted.
In the asynchronous mode, the periodically update of the plugin's data happens in the `process` method, controlled by a timer. Mason uses the `ProcessingManager` to control the processing procedure of the synchronous plugins by using the `ProcessingTree`. For plugins in the synchronous mode, the update of plugins is induced by calling the previously specified callback method of the processing plugin based on the information in the `ProcessingTree`, when the data assigned to the `ProcessingHandle` notifies a change.

5.2.4 Abstract Sensor Plugin

The `AbstractSensorPlugin` inherits from the `ProcessingPlugin` and represents a sensor in the framework. Each sensor plugin consists of a sensor model and one or multiple hardware interface plugins which are responsible for acquiring the sensor measurement data from the sensor hardware.
Sensor plugins can serve multiple fusion and generator plugins, creating local environment models and generating different environment representations based on a single sensor.

Sensor Model

Each sensor has its own sensor model which describes the attributes of the sensor. These attributes are the noise model coefficients, the minimum and maximum measurement range and the information if the sensor provides color information.
The Gaussian based noise model describes the standard deviation of the axial noise dependent on the measured distance:

$$\sigma_d = a + bd + cd^2 \tag{5.1}$$

with the coefficients a, b and c and is capable to model different sensor noise characteristics, including laser scanners and RGB-D cameras sufficient.

Hardware Interface Plugin

In order to provide the sensor plugin with the latest available sensor measurement data, each sensor plugin has one or multiple hardware interface plugins.
A hardware interface plugin creates `AbstractData` by subscribing to specified ROS topics that provide the data or by using custom hardware drivers.

5.2.5 Abstract Fusion Plugin

The `AbstractFusionPlugin` inherits from the `ProcessingPlugin` and is responsible for combining data when its `process` method is called or a callback has been triggered.
Each fusion plugin should be able to cover two cases: Firstly, the sensor internal fusion of measured data over time and secondly, the fusion of previously fused data from different sensors.
The `AbstractFusionPlugin` has access to the sensor model in order to consider the noise characteristic and also the measurement range of the sensors during the fusion process.
By incorporating the sensor noise, an improved world model is achievable.

5.2.6 Abstract Generator Plugin

The `AbstractGeneratorPlugin`, inheriting from the `ProcessingPlugin`, provides an interface for plugins which are supposed to transform given data into another representation, e.g. a surface normals estimation point cloud. This data can come from a single sensor (local data) or already fused by a fusion plugin (global data).

5.2.7 Abstract Filter Plugin

In order to enable filtering the data used in the framework, the `AbstractFilterPlugin`, provides an interface for this purpose. By using the `AbstractFilterPlugin`,the framework could for example apply a robot self filtering, removing measured data corresponding to links of the robot from measurement data or reduce the noise of measurements.

6 Implementation

During this chapter, the development of the first plugins for Mason consisting of a fusion plugin and multiple generator plugins that work with a spatially hashed TSDF representation, is presented.

6.1 OpenCHISEL as Mason Fusion Plugin

In the following section, the development of the chisel fusion plugin is described. The chisel fusion plugin is responsible for integrating sensor measurements of a single sensor into a local TSDF representation and for integrating local TSDF representations into a global TSDF representation as well.
The chisel fusion plugin is located in the `chisel_mason_bridge` ROS package and implements the interface defined by the `AbstractFusionPlugin`. It wraps the OpenCHISEL library and provides the functionality of the OpenCHISEL library to the complete Mason framework. Therefore, all the other plugins can make use of the features provided by the chisel fusion plugin.
OpenCHISEL was selected as basis of the fusion plugin as it uses the spatially hashed TSDF concept. The requirements in terms of memory and computing power are reduced compared to the conventional TSDF approach, which enables to use it in real time running on the CPU. Furthermore, the reduced memory consumption allows the usage of multiple chisel fusion plugins for the simultaneous fusion of multiple sensors. By using the TSDF for creating a 3D representation of the robot's environment, we are able to generate dense surfaces even from spinning laser scanners although they are known to deliver rather sparse measurements of the environment.

6.1.1 TSDF Fusion of a Single Sensor

The mono sensor fusion of the distance measurements is implemented by the OpenCHISEL library that uses the weighted update strategy described in section 2.1.1. But, the weight of the voxels depends on the measured distance and the estimated sensor noise using a quadratic sensor noise model. This means that the weight

$$w_k(\mathbf{x}, \mathbf{z}) = \frac{\alpha}{2\tau} = \frac{\alpha}{\beta\sigma_z} = \frac{\alpha}{\beta(a + bz + cz^2)} \tag{6.1}$$

is defined by the scaling factor α, the noise model coefficients a, b, c and β as scaling parameter controlling the amount of noise standard deviations that should be considered. The weight is lower for distant points, i.e. they have a lower certainty than closer points.
Currently, OpenCHISEL supports the integration of colored and monochrome point clouds and depth images. Besides the integration of sensor data, OpenCHISEL also offers an implementation of marching cubes for meshing and the computation of normals for shading the generated mesh.

Assignment of a Voxel to a given World Position

In order to work with the spatially hashed TSDF data structure, the assignment of a position $\mathbf{x}$ to the closest voxel in the closest chunk is important. At first, the chunk size $\mathbf{N} = \begin{bmatrix} N_x & N_y & N_z \end{bmatrix}^\top$ needs to be defined which implies the amount of voxels for every direction contained in a single chunk. Based on the chunk size and the voxel resolution r, the integer chunk world coordinates

$$\mathbf{ChunkID}(\mathbf{x}) = \begin{bmatrix} \text{ChunkID}_x & \text{ChunkID}_y & \text{ChunkID}_z \end{bmatrix}^\top = \text{floor}\begin{bmatrix} \frac{x}{rN_x} & \frac{y}{rN_y} & \frac{z}{rN_z} \end{bmatrix}^\top \tag{6.2}$$

are computed. Based on the chunk coordinates, the chunk origin

$$\textbf{ChunkOrigin}(\textbf{ChunkID}(\mathbf{x})) = r \cdot \begin{bmatrix} \text{ChunkID}_x \cdot N_x & \text{ChunkID}_y \cdot N_y & \text{ChunkID}_z \cdot N_z \end{bmatrix}^\top \tag{6.3}$$

is defined. The voxel coordinates relative to the chunk origin are set as

$$\textbf{VoxelCoordinates}(\mathbf{x}) = \begin{bmatrix} \bar{x} & \bar{y} & \bar{z} \end{bmatrix}^\top = \text{floor}\left(\begin{bmatrix} \frac{x}{r} & \frac{y}{r} & \frac{z}{r} \end{bmatrix}^\top - \textbf{ChunkOrigin}\left(\textbf{ChunkID}\left((\mathbf{x})\right)\right) \right). \tag{6.4}$$

Now we can compute the voxel ID, defining the voxel's storage position in the chunk's vector of all voxels assigned to the chunk:

$$\textbf{VoxelID}(\bar{\mathbf{x}}) = (\bar{z} \cdot N_z + \bar{y}) \cdot N_x + \bar{x} \tag{6.5}$$

For the found voxel we can now update the related TSDF value $\Phi_\tau(\mathbf{x})$ and the weight $W(\mathbf{x})$.

Enhancements of the OpenCHISEL Library

There were many minor and major enhancements and bug fixes applied to the OpenCHISEL library. The most important changes are presented here.

Enabling Incremental Updates

In order to enable Mason to work with incremental updates, OpenCHISEL now remembers all the chunks which got changed or deleted since the last request of incremental data. This has the advantage that fusion plugins for global TSDF updates and generator plugins only have to consider the recently changed and deleted chunks and therefore gain efficiency.

Detection and Removal of Empty Chunks

In the OpenCHISEL library, space carving – the removal of dynamic objects from the scene after their disappearance – was implemented for the integration of depth images. Originally, only non-positive SDF values outside of the truncation region have been removed during space carving within a chunk. Unfortunately, chunks remained in the memory, although they were empty.
This behavior has been changed in a way that every voxel outside of the truncation distance (see section 2.1.2) gets deleted too when the SDF is positive, as the surface referred to the SDF value is gone.

Algorithm 4 Given Space Carving

```
1: for v_c ∈ V do                                   ▷ For each voxel
2:     u ← ComputeSurfaceDistance(v_c)
3:     if u(v_c) ≥ τ + ϵ then ▷ Do space carving
4:         if Φ_τ(v_c) ≤ 0 then
5:             Φ_τ(v_c) ← undefined
6:             W(v_c) ← 0
7:         end if
8:     end if
9: end for
```

Algorithm 5 Changed Space Carving

```
1: for v_c ∈ V do                                   ▷ For each voxel
2:     u ← ComputeSurfaceDistance(v_c)
3:     if u(v_c) ≥ τ + ϵ then ▷ Do space carving
4:         Φ_τ(v_c) ← undefined
5:         W(v_c) ← 0
6:     end if
7: end for
```

Now, we can search for allocated chunks that do not contain valid data anymore and remove them. This is implemented by periodically iterating through all voxels of a chunk until a valid voxel – a voxel with a weight $W(\mathbf{x}) > 0$ – is found. Once we have passed all voxels, we know that the chunk is empty and we can delete it safely.

Improved Point Cloud Integration Method

The experience with the OpenCHISEL library showed us that the given implementation for point cloud integration was a big bottleneck, as it did not scale with an increasing amount of updated chunks based on the point cloud. The given implementation (see algorithm 6) first iterates through all points of the point cloud in order to find all chunks passed by sensor rays using ray casting. After the detection of all chunks passed by sensor rays, each chunk is handled individually. For each chunk, we iterate through the point cloud again and search all voxels inside the truncation region of a single point using ray casting. Afterwards, we compute for each found voxel the surface distance u related to the data point. If the surface distance u is inside the truncation interval, i.e. $\|u\| < \tau$, we update the voxel's surface distance and weight by a weighted update.
Since we handle each chunk isolated and have to check the entire point cloud for each chunk, the runtime of the point cloud integration method heavily depends on the chunk distribution resulting from the point cloud. This is inefficient, especially when accumulating laser scan data into point clouds because many chunks have to be updated due to the usually high sensor range of laser scanners.

Algorithm 6 Given Point Cloud Integration Method

```
1:  for p ∈ P do                                          ▷ For each point in the point cloud
2:      C.append(RayCast(o_sensor, p))                    ▷ Collect all chunks passed by sensor ray from o_sensor to p
3:  end for
4:  for c ∈ C do                                          ▷ For each chunk passed by a ray
5:      for p ∈ P do
6:          z ← ||p − o_sensor||                          ▷ Compute depth
7:          d ← (p − o_sensor)/z                          ▷ Compute ray direction
8:          τ ← ComputeTruncationDistance(z)              ▷ Compute truncation distance for truncation region
9:          V_passed ← RayCast(p − τd, p + τd)            ▷ Remember all voxels passed inside the chunk
10:         for v_c ∈ V_passed do                         ▷ For each voxel inside the truncation region
11:             u ← ComputeSurfaceDistance(v_c, p)
12:             if ||u(v_c)|| < τ then                    ▷ If surface distance is below truncaction distance
13:                 v_c ← integrateTSDF(u)                ▷ Integrate surface distance using a weighted update
14:             end if
15:         end for
16:     end for
17: end for
```

Therefore, we improved the point cloud integration method by removing the runtime influence of the amount of updated chunks and also by adding space carving for considering dynamic objects. Instead of iterating multiple times through the point cloud, each point is now visited only once, as illustrated in algorithm 7. At first, we cast a ray from the sensor origin to the lower border of the point's truncation region and remember all passed voxels for space carving. When the sensor ray has passed these voxels, we consider them as free and remove the SDF values and the weights of the voxels. The last step is the treatment of voxels along the sensor ray inside the truncation region. These voxels are searched with help of ray casts and then are handled separately. For each voxel, we compute the surface distance and the weight of the corresponding point in the point cloud and merge the values using weighted update.

Algorithm 7 Improved Point Cloud Integration Method

for $\mathbf{p} \in \mathbf{P}$ **do** ▷ For each point in the point cloud
  $z \leftarrow \|\mathbf{p} - \mathbf{o}_{sensor}\|$ ▷ Compute depth
  $\mathbf{d} \leftarrow (\mathbf{p} - \mathbf{o}_{sensor})/z$ ▷ Compute ray direction
  $\tau \leftarrow$ **ComputeTruncationDistance**(z) ▷ Compute truncation distance for truncation region
  $\mathbf{V}_{free} \leftarrow$ **RayCast**$(\mathbf{o}_{sensor}, \mathbf{p} - (\tau + \epsilon)\mathbf{d})$ ▷ Remember voxels in front of truncation region as free
  for $\mathbf{v}_c \in \mathbf{V}_{free}$ **do** ▷ For each by the sensor ray passed voxel
    $\Phi_\tau(\mathbf{v}_c) \leftarrow$ undefined ▷ Delete Voxel
    $W(\mathbf{v}_c) \leftarrow 0$
  end for
  $\mathbf{V}_{passed} \leftarrow$ **RayCast**$(\mathbf{p} - \tau\mathbf{d}, \mathbf{p} + \tau\mathbf{d})$ ▷ Remember all voxels passed inside the chunk
  for $\mathbf{v}_c \in \mathbf{V}_{passed}$ **do** ▷ For each voxel inside the truncation region
    $u \leftarrow$ **ComputeSurfaceDistance**$(\mathbf{v_c}, \mathbf{p})$
    if $\|u(\mathbf{v}_c)\| < \tau$ **then** ▷ If surface distance is below truncaction distance
      $\mathbf{v}_c \leftarrow$ **integrateTSDF**(u) ▷ Integrate surface distance using a weighted update
    end if
  end for
end for

6.1.2 TSDF Fusion of Multiple Sensors

In order to fuse local TSDF representations from different sensors to a single global TSDF representation, we assume that the local and global spatially hashed voxel grids are defined in the same coordinate system, i.e. with the same coordinate system origin and orientation. In practical application, this is ensured by transforming the sensor data into a common coordinate system defined by the framework before the local TSDF fusions by OpenCHISEL. The first step of the fusion is the alignment of the local source voxel grid structure with the structure of the global target voxel grid. This enables us to use sensors with different resolutions and chunk sizes to fit the characteristic of the sensors.
We have to consider four different cases for aligning, namely:

1. Same voxel resolution and amount of voxels in each chunk (chunk size)
2. Same voxel resolution and different chunk size
3. Different voxel resolution and different chunk size
4. Different voxel resolution and same different chunk size

After the alignment is done, we are able to fuse the local chunks into the global TSDF representation using a weighted update.

Case 1: Same Voxel Resolution and Chunk Size

In the trivial case, the source chunks have the same voxel resolution and the same amount of voxels in each chunk and thus are already aligned as illustrated in figure 6.1.

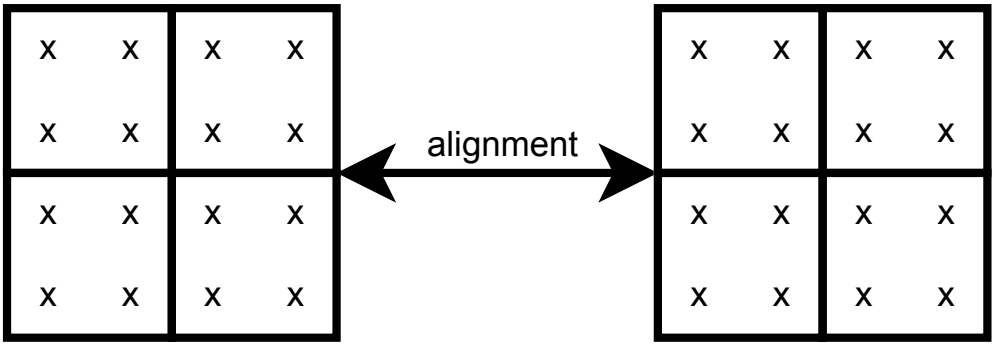


Figure 6.1: 2D voxel grids with same voxel resolution and chunk size

This means we can integrate the chunks directly into the global TSDF representation.

Case 2: Same Voxel Resolution but Different Chunk Size

The next possible case consists of a local voxel grid with the same voxel resolution as the global voxel grid, but the voxel grid is divided into chunks of different sizes as shown in figure 6.2.

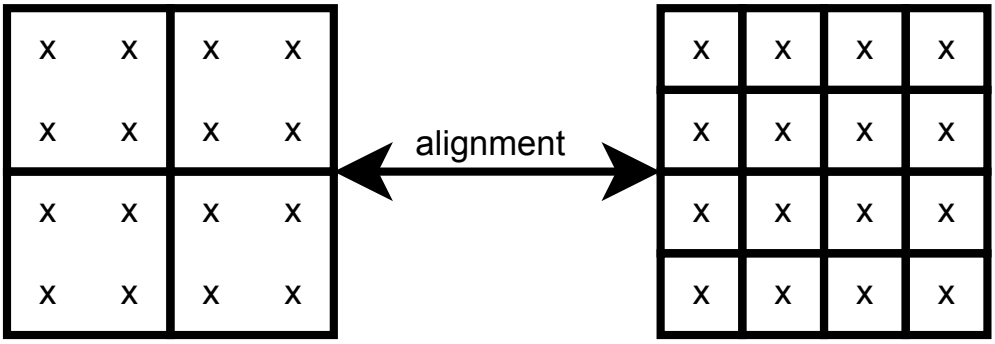


Figure 6.2: 2D voxel grids with same voxel resolution but different chunk size

Therefore, we have to search the new voxel position in the target grid for every voxel of the source grid and reassign the SDF values $\Phi_\tau(\mathbf{x})$ and the weights $W(\mathbf{x})$ from the source voxel to the target voxel for all chunks. Given the source chunk ID $\mathbf{ChunkID_s}$, the source chunk size $\mathbf{N}_s$, the vector of voxels and the voxel resolution r, we have to compute the source voxel's world position $\mathbf{x}$ first. This is done by inverting equation (6.5) for every voxel ID i in the chunk in order to get the relative voxel coordinates:

$$\bar{\mathbf{x}}(i) = \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{bmatrix} = \begin{bmatrix} i \bmod N_x \\ \frac{i-\bar{x}}{N_x} \bmod N_z \\ \left(\frac{i-\bar{x}}{N_x} - \bar{y}\right)\frac{1}{N_z} \end{bmatrix} \tag{6.6}$$

Now, the world position – the continuous point defined in the coordinate system of the framework – of the voxel center is:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \underbrace{r \cdot \begin{bmatrix} N_x \cdot \text{ChunkID}_x \\ N_y \cdot \text{ChunkID}_y \\ N_z \cdot \text{ChunkID}_z \end{bmatrix}}_{\text{chunk origin}} + \underbrace{r \cdot \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{bmatrix}}_{\text{relative voxel position}} + \underbrace{\frac{r}{2} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{\text{voxel offset}}, \tag{6.7}$$

consisting of the world position of the chunk origin, the voxel position relative to the chunk origin and the offset of the voxel corner to the voxel center of the cuboid voxel.
By using this world position from equation (6.7), we can search the voxel in the target grid with help of equations (6.2) to (6.5) and copy the SDF value $\Phi_\tau(\mathbf{x})$ and the weight $W(\mathbf{x})$ of the source voxel to the target voxel.

Case 3 and 4: Different Voxel Resolution and Different Chunk Size

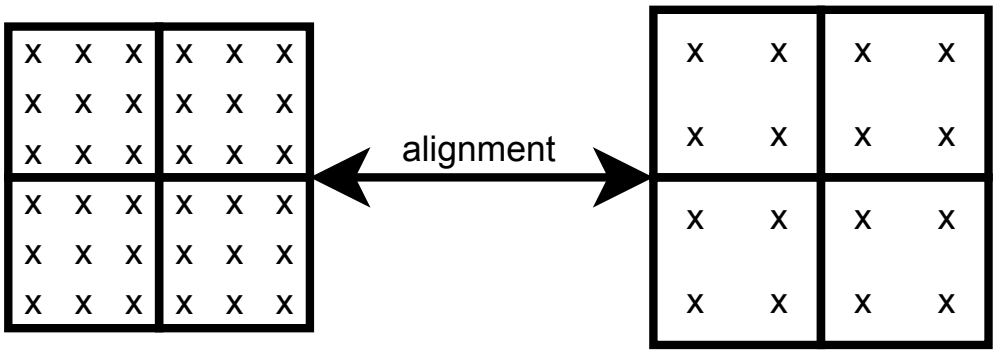


Figure 6.3: 2D voxel grids with different voxel resolution and arbitrary chunk size

The worst possible case is a differing structure of the grid as can be seen in figure 6.3. Here, the voxel resolutions of the source and target voxel grids are different. Thus, to predict the SDF values $\Phi_\tau(\mathbf{x})$ and the weights $W(\mathbf{x})$ for the voxels of the target grid, we have to interpolate the source grid, no matter if the chunk sizes are equal or not. For this purpose, two different methods have been implemented, the trilinear interpolation and the nearest neighbor interpolation.

Trilinear Interpolation

The first implemented interpolation method is the trilinear interpolation, an extension of linear interpolation to the 3D case. We consider a target voxel position $\mathbf{x} = [\, x \; y \; z \,]^\top$ of the target chunk for which we want to estimate the values using the eight neighboring lattice points. Now we determine the closest lower voxel position $\mathbf{x_0} = [\, x_0 \; y_0 \; z_0 \,]^\top$ and the next higher voxel position $\mathbf{x_1} = [\, x_1 \; y_1 \; z_1 \,]^\top$ in the regular grid of the source chunk.
Let $\mathbf{x_d} = [\, x_d \; y_d \; z_d \,]^\top$ be the normalized distance between $\mathbf{x}$ and $\mathbf{x_0}$, that is:

$$x_d = \frac{x - x_0}{x_1 - x_0}, \qquad y_d = \frac{y - y_0}{y_1 - y_0}, \qquad z_d = \frac{z - z_0}{z_1 - z_0}. \tag{6.8}$$

The first interpolation step is now along the x-direction. In this example, we want to estimate the TSD value at position $\mathbf{x}$:

$$\Phi_{00} = \Phi_\tau \begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix} (1 - x_d) + \Phi_\tau \begin{bmatrix} x_1 & y_0 & z_0 \end{bmatrix} x_d \tag{6.9}$$

$$\Phi_{01} = \Phi_\tau \begin{bmatrix} x_0 & y_0 & z_1 \end{bmatrix} (1 - x_d) + \Phi_\tau \begin{bmatrix} x_1 & y_0 & z_1 \end{bmatrix} x_d \tag{6.10}$$

$$\Phi_{10} = \Phi_\tau \begin{bmatrix} x_0 & y_1 & z_0 \end{bmatrix} (1 - x_d) + \Phi_\tau \begin{bmatrix} x_1 & y_1 & z_0 \end{bmatrix} x_d \tag{6.11}$$

$$\Phi_{11} = \Phi_\tau \begin{bmatrix} x_0 & y_1 & z_1 \end{bmatrix} (1 - x_d) + \Phi_\tau \begin{bmatrix} x_1 & y_1 & z_1 \end{bmatrix} x_d \tag{6.12}$$

where $\Phi_\tau [\, x_0 \; y_0 \; z_0 \,]$ denotes the TSD value Φ_τ at position $[\, x_0 \; y_0 \; z_0 \,]^\top$. After interpolating along the x-direction, we interpolate along the y-direction:

$$\Phi_0 = \Phi_{00}(1 - y_d) + \Phi_{10} y_d \tag{6.13}$$

$$\Phi_1 = \Phi_{01}(1 - y_d) + \Phi_{11} y_d \tag{6.14}$$

Finally we interpolate Φ_0 and Φ_1 along the z direction:

$$\Phi_\tau \begin{bmatrix} x & y & z \end{bmatrix} = \Phi_0(1 - z_d) + \Phi_1 z_d \tag{6.15}$$

which gives us $\Phi_\tau [\, x \; y \; z \,]$, the estimation of the TSD value at the target voxel position $\mathbf{x}$.
In order to estimate the weight $W(\mathbf{x})$ at position $\mathbf{x}$ as well, we have to repeat the steps in the equations (6.9) to (6.15).

Nearest Neighbor Interpolation

As the trilinear interpolation visits eight neighboring voxels for each voxel that shall be interpolated, it can be very time consuming. Therefore, the nearest neighbor interpolation has been implemented.
The nearest neighbor interpolation searches the closest voxel in the source grid to the non-existing voxel position of the target grid and uses the values of this point.
To find the nearest neighbor, we examine the distance $\Delta\mathbf{x}$ of the target voxel's position $\mathbf{x}$ to the next lower voxel in the source grid by using the source voxel resolution r_{source}:

$$\Delta\mathbf{x} = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} x \bmod r_{source} \\ y \bmod r_{source} \\ z \bmod r_{source} \end{bmatrix}, \tag{6.16}$$

and determine the source voxel position $\mathbf{x_0}$ with help of the distances $\Delta\mathbf{x}$:

$$\mathbf{x_0} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} f(x, \Delta x, r_{source}) \\ f(y, \Delta y, r_{source}) \\ f(z, \Delta z, r_{source}) \end{bmatrix}, \tag{6.17}$$

with

$$f(x, \Delta x, r_{source}) = \begin{cases} x - \Delta x, & \text{if } \Delta x < 0.5 \cdot r_{source} \\ x + \Delta x, & \text{else} \end{cases} \tag{6.18}$$

Now we can use $\Phi_\tau(\mathbf{x_0})$ and the weights $W(\mathbf{x_0})$ for the target voxel's values at $\mathbf{x}$.

TSDF Integration

As the alignment of the chunks is now ensured, we can integrate the local source chunks into the global target chunks with a weighted update as described by Werner et al. [23]:

$$\mathrm{TSDF}_i(\mathbf{x}) = \frac{W_{i-1}\mathrm{TSDF}_{i-1}(\mathbf{x}) + w(\mathbf{x})\mathrm{tsdf}(\mathbf{x})}{W_{i-1}(\mathbf{x}) + w(\mathbf{x})} \tag{6.19}$$

$$W_i(\mathbf{x}) = W_{i-1}(\mathbf{x}) + w(\mathbf{x}), \tag{6.20}$$

where $\mathrm{TSDF}(\mathbf{x})$ is the SDF value $\Phi(\mathbf{x})$ in the global TSDF representation and $\mathrm{tsdf}(\mathbf{x})$ is the truncated SDF value $\Phi_\tau(\mathbf{x})$ of the local TSDF representation.

6.2 OpenCHISEL Based Mason Generator Plugins

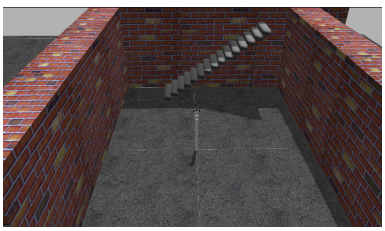
In the following section, the development of the chisel generator plugins is described. The chisel generator plugins are responsible for generating a new representation (e.g. an elevation map) based on a given TSDF scene. The chisel generator plugins are located in the `chisel_mason_bridge` ROS package and implement the interface defined by the `AbstractGeneratorPlugin`.
The Mason generator plugins based on OpenCHISEL support the previously described incremental updates, thus on each update only the recently changed and deleted chunks are considered. This enables a faster representation generation, as we do not need to check all available chunks in the global scene.
Currently, there are three generator plugins implemented, the height grid map, the surface normal estimation and the mesh generator plugins.

6.2.1 Height Grid Map Generator Plugin

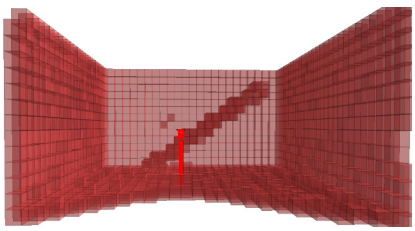
The height grid map generator plugin creates a grid map containing the maximum height for each cell, also called elevation map. In order to know which chunks are the highest in the xy-plane while supporting incremental chunk updates and using a spatially hashed TSDF, we have to keep track of the highest chunks ever found.
This is done by adapting the hash function (equation (2.2) on page 7) to the 2D case:

$$H(x, y) = (x \cdot 73856093 \oplus y \cdot 19349663) \bmod n. \tag{6.21}$$

We now maintain a hash table storing the highest existing z chunk ID for each x and y chunk ID, as seen in figure 6.4.



(a) Picture of the simulated scene



(b) Stored chunks of the scene

…	13	13	13	13	13	13	13	13	13	13	13	13	13
…	5	6	7	7	8	9	9	10	11	0	0	0	13
…	5	6	7	7	8	9	9	10	11	0	0	0	13
…	5	6	7	7	8	9	9	10	11	0	0	0	13
…	5	6	7	7	8	9	9	10	11	11	0	0	13
…	0	0	0	0	0	0	0	0	0	0	0	0	13
…	…	…	…	…	…	…	…	…	…	…	…	…	…

(c) Upper right section from the 2D map of highest chunks

Figure 6.4: A scene and the corresponding 2D map of highest chunks

Working with Incremental Updates

The first step for updating the height grid map is removing the entries of deleted chunks in the hash table of highest chunks, as these surfaces do not exist anymore. Afterwards, we iterate through all changed chunks and check for each if the z chunk ID is equal or greater than the stored z chunk ID.
If this is the case, we search in the chunk for the zero crossing, as the zero crossing of the SDF values Φ_τ represents the surface.

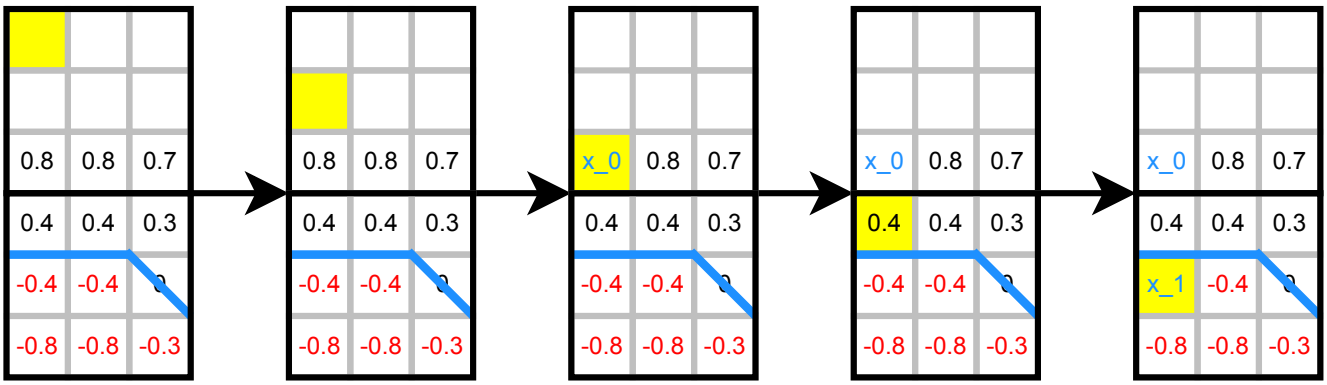


Figure 6.5: Cross section showing the descend in z direction through two chunks to find the voxel positions $\mathbf{x_0}$ and $\mathbf{x_1}$ with sign-changed Φ_τ values causes by the blue surface

Starting with every voxel in the uppermost voxel layer of the chunk, we descend along the voxels in the z direction until we find the first voxel with a valid SDF value as illustrated in figure 6.5. If we reach the lower chunk border we search the next lower chunk and continue our search. After finding the first valid voxel $\mathbf{x}_0$, we continue our descent until we find the first voxel $\mathbf{x}_1$ with a sign-changed SDF value.
If we have registered a zero crossing in the surface distances Φ_τ, we know that the found voxels $\mathbf{x_0}$ and $\mathbf{x_1}$ encompass the surface and we can interpolate in the the z-direction to get the exact position of the surface. Based on the surface distances $\Phi_\tau(\mathbf{x_0})$ and $\Phi_\tau(\mathbf{x_1})$ we can interpolate the position $\mathbf{x}$ of the surface linearly:

$$t = \frac{\Phi_\tau(\mathbf{x_0})}{\Phi_\tau(\mathbf{x_0}) - \Phi_\tau(\mathbf{x_1})} \tag{6.22}$$

$$\mathbf{x} = \mathbf{x_0} + t\,(\mathbf{x_1} - \mathbf{x_0}) \tag{6.23}$$

Now, the found surface position $\mathbf{x}$ is ready to insert into the height grid map.

6.2.2 Surface Normals Estimation Generator Plugin

The surface normals estimation generator plugin estimates the normals at the isocontour of the TSDF representation. Therefore, we have to compute the isocontour of the chunks by using OpenCHISEL's marching cube implementation in the first step and computing the surface normals based on the TSDF afterwards.

Computation of the Surface Normals

Given the isocontour, we can compute the surface normal for every voxel $\mathbf{x}$ with on the surface, i.e. $\Phi_\tau(\mathbf{x}) = 0$, based on the TSDF.
As the gradient of $\Phi_\tau(\mathbf{x})$ is assumed to be orthogonal to the isocontour, we can estimate the surface

normal directly by computing the gradient $\nabla\Phi_\tau(\mathbf{x})$ according to Newcombe et al. [7]. The gradient $\nabla\Phi_\tau(\mathbf{x})$ is computed using a numerical derivative of the TSDF:

$$\nabla\Phi_\tau(\mathbf{x}) = \left[\frac{\partial\Phi_\tau(\mathbf{x})}{\partial x} \quad \frac{\partial\Phi_\tau(\mathbf{x})}{\partial y} \quad \frac{\partial\Phi_\tau(\mathbf{x})}{\partial z}\right]. \tag{6.24}$$

Thus, the surface normal $\mathbf{N}(\mathbf{x})$ is given as the normalized gradient:

$$\mathbf{N}(\mathbf{x}) = \frac{\nabla\Phi_\tau(\mathbf{x})}{\|\nabla\Phi_\tau(\mathbf{x})\|}. \tag{6.25}$$

Working with Incremental Updates

The generator maintains a hash table containing shared pointers to meshes of OpenCHISEL's chunks. Each mesh is inserted into the hash table by computing the hash value of the chunk coordinates with the same hash function used for inserting chunks (equation (2.2) on page 7) and then saving the shared pointer pointing to the mesh in the found bucket.

The first step of updating the surface normals estimation is removing all mesh pointers of deleted chunks from the hash table containing all mesh pointers in order to consider removed surfaces. As a next step, we integrate the changed chunks into the representation. Therefore, we first check for every changed chunk if the chisel scene data, consisting of OpenCHISEL's meshes, already has the searched mesh. OpenCHISEL stores meshes composed of the vertices, colors and – if requested – normals for each chunk. We only compute the mesh if it is not available and was therefore not updated by OpenCHISEL itself, to prevent unnecessary computation. Based on the isocontour of the surface, we can compute the surface normals with help of the gradient.
Afterwards, we fill a point cloud composed of the surface points and normals with the mesh vertices and the corresponding normals that have been computed based on the TSDF representation and save it in the framework's data.

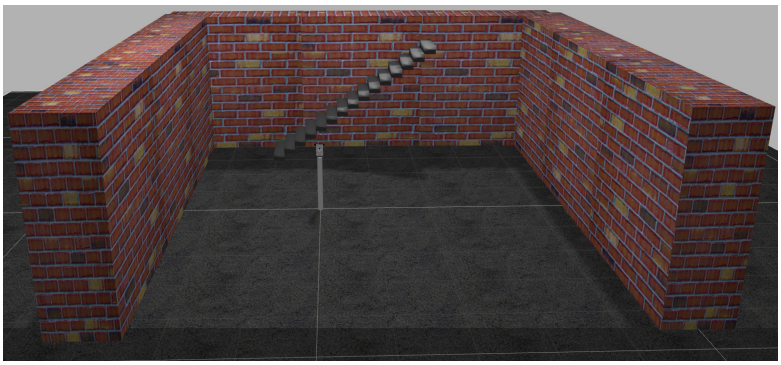
6.2.3 Mesh Generator Plugin

The mesh generator plugin creates a mesh of the surface based on the TSDF representation.
For each chunk which got changed since the last update, the plugin creates a mesh with OpenCHISEL's marching cube implementation.
If requested, the plugin fills a ROS marker array with the mesh for visualization using RViz. The outsourcing of the meshing component into an independent plugin enables the control of the meshing frequency separated from the sensor data integration.

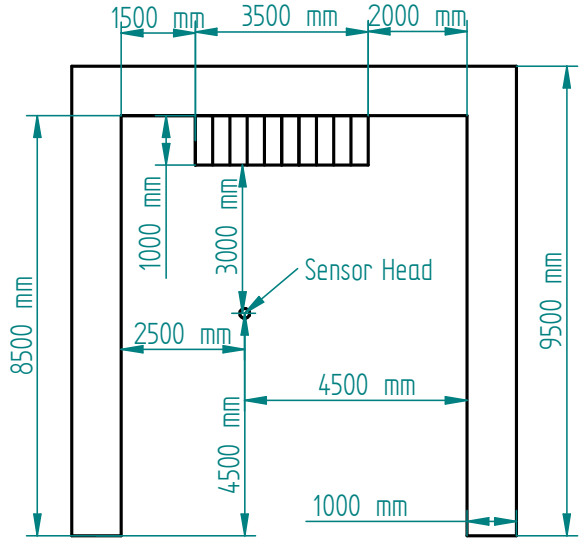
7 Results

In the following section, the OpenCHISEL based plugins – the fusion plugin, the height grid map generator, the mesh generator and the surface normals estimation plugin – are examined.

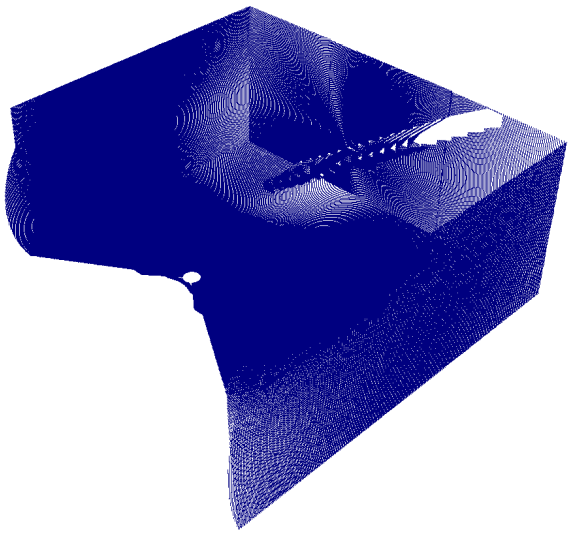
7.1 Test Scene and Setup



(a) Picture of the test scene



(b) Dimensions of the test scene



(c) The captured ground truth

Figure 7.1: The used test scene

The ground truth for comparing the accuracy of Mason's generated representation utilizing the OpenCHISEL library was captured with the slow rotating laser scanner without simulated Gaussian noise to get a dense and exact point cloud of the scene displayed in figure 7.1c.
In the tests, the Hokuyo UTM-30LX-EW laser scanner was scanning the room with of 7 m x 8.5 m floor area and 3 m high walls and simulated Gaussian noise while rotating with 0.6 rad/m. The Asus Xtion Pro Live RGB-D camera is looking straight in the direction of the stairs. Both sensors are mounted on the standalone version of THOR-Mang's sensor head, the Hector multi-sensor head.
The sensor head is mounted on a 1 m tall cylinder and is oriented perpendicular to the floor.
All the timings were determined on an Intel Core i7 5600U 15W Dual Core CPU with a Gazebo 2 simulation running on Kubuntu 14.04 LTS.

7.2 Improved Point Cloud Integration Method

In the following section, the improved point cloud integration method is examined in terms of accuracy and execution time using the rotating laser scanner.

7.2.1 Accuracy of the Resulting Mesh

In order to validate the accuracy of the improved point cloud integration method, the mesh generator plugin generates a mesh based on the TSDF representation created previously by the fusion plugin.
The created mesh is then compared with the ground truth by using CloudCompare 2.6.0 [24], an open-source 3D point cloud and mesh processing software.

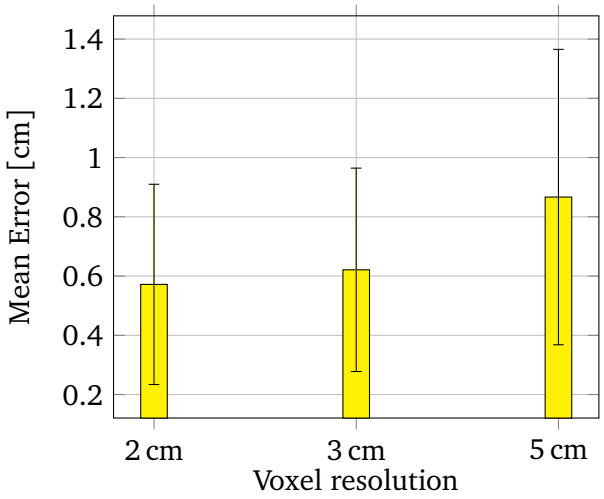
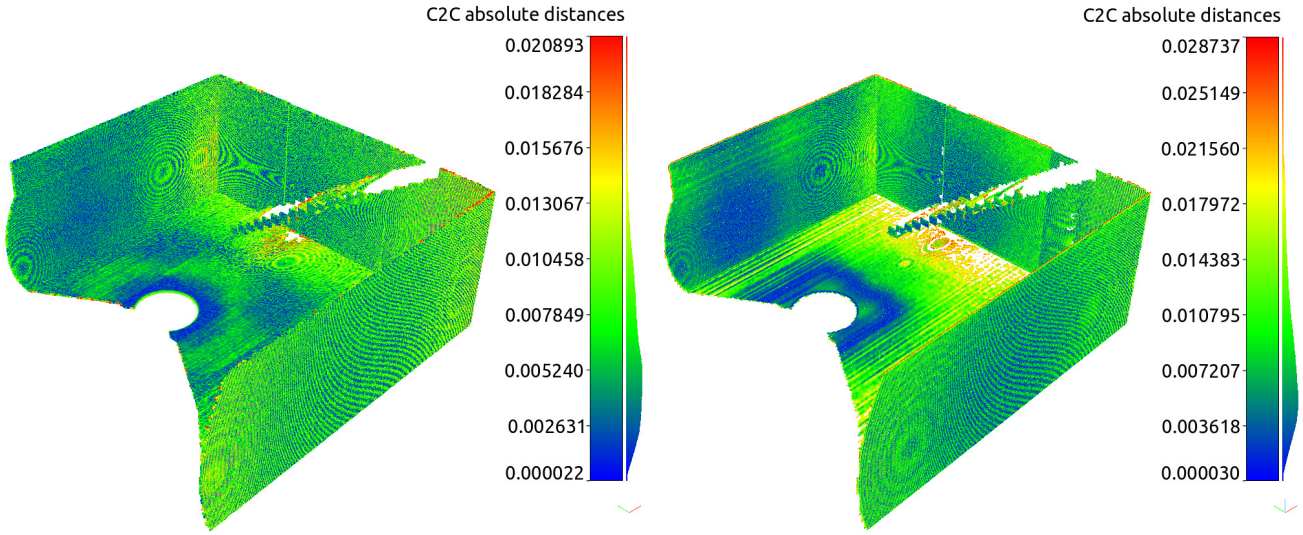


Figure 7.2: Accuracy of the laser scan integration method using a chunk size of 4^3

In figure 7.2, we can see the mean error of between the resulting mesh and the ground truth is very low. This shows the big advantage in using the TSDF representation as it behaves like a least squares optimization of the isosurface, as stated by Curless and Levoy [5].



(a) 3 cm resolution **(b)** 5 cm resolution

Figure 7.3: Absolute error [m] of the representation

Besides the mean error, the distribution of the error was analyzed as well, as showed in figure 7.3. We can see that the error of the TSDF representation in the direct surrounding of the robot is very low and is at highest in the area under the stairs. The error below the stairs reduces significantly with increasing resolution. The circular artifacts at the wall that cause a higher error are not produced by the TSDF laser scan integration, as these are already visible in the captured ground truth data (see figure 7.1c). The improved point cloud integration method is able to estimate the world model very precise, as soon as enough data is available, for example in the area around the robot. When this data is very sparse, for example in the area under the stairs, this results in a higher mean error.

7.2.2 Execution time

In this experiment, the speedup achieved by the new point cloud integration method was evaluated. The lapse of time needed to integrate a single laser scan as point cloud was measured 5000 times for different setups of resolutions and chunk size and then averaged. All the timings only consider the time needed for integrating the sensor data into the TSDF representation and therefore do not include the time needed for meshing.

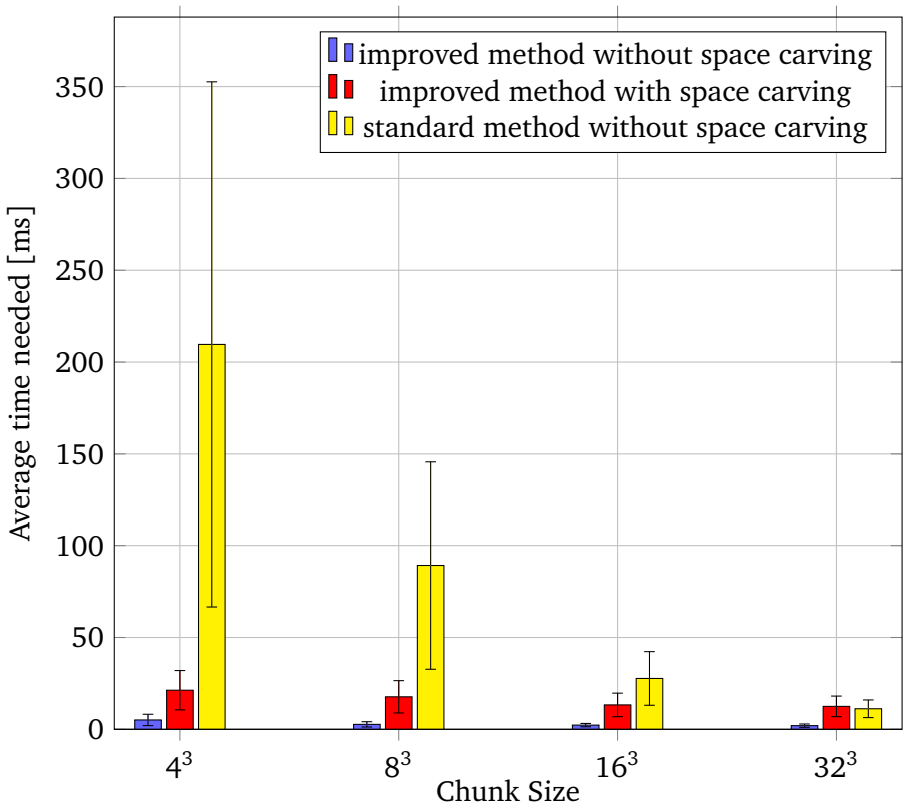


Figure 7.4: Timings for integrating a single laser scan as point cloud averaged over 5000 scans using a voxel resolution of 1.5 cm

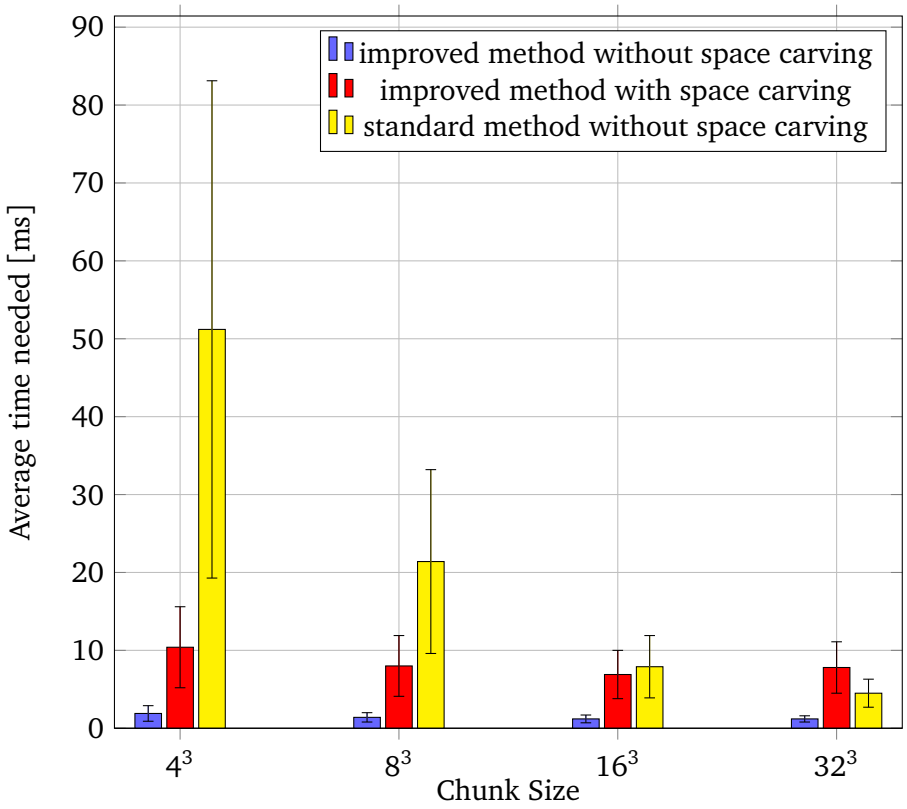


Figure 7.5: Timings for integrating a single laser scan as point cloud averaged over 5000 scans using a voxel resolution of 3 cm

In the figures 7.4 and 7.5, we can see that the elapsed time for integrating a laser scan increases with decreasing chunk size when using OpenCHISEL's standard method. This can be explained with the fact that the number of chunks is increasing for a decreasing chunk size and therefore the laser scan hits more chunks. Furthermore, the standard method doesn't support space carving. As explained in section 6.1.1, the standard method scales very bad with an increasing number of chunks and thus the elapsed times increase.
When disabling space carving, the new method outperforms the standard method all the time. With space carving enabled, the new point cloud integration method performs better than the old method for chunk sizes below 16^3. Smaller chunk size are desirable for meshing chunks as the updated amount of chunks tends to be smaller due to the sparse laser measurements and therefore less chunks needing a mesh update.

7.3 Runtime Comparison for the Chunk Alignment

As next, the execution times for aligning the chunks and integrating the chunks into a global TSDF representation were investigated. The timings do not include the time needed for meshing the created TSDF data. For this experiment, we are reusing the test scene described in section 7.1. In table 7.1, we can see that the fusion of chunks from local TSDF representations into a global TSDF representation for matching source and target chunk sizes and resolution needs for every investigated resolution around or less 1.1 ms.
When using the same target and source resolution, but different chunk sizes, the fusion plugin has to find for each source voxel the correspondence in the target grid and copy the values before the plugin can integrate the chunks. This needs in average 2.0 ms for the smallest chunk sizes increasing to 39.6 ms for the largest chunk sizes. Therefore, in order to provide real time updates, small chunk size should be preferred.
If the source and target voxel resolutions as well as the chunk sizes differ, the fusion plugin needs to interpolate. When using nearest neighbor interpolation, the timings as seen in table 7.1 have a range from 2.5 ms for downscaling from 2 cm resolution to 5 cm resolution to 161.7 ms when upscaling from 5 cm resolution to 2 cm resolution.
The trilinear interpolation is computationally more expensive, but tends to be more accurate as the target values are computed instead of taking the values of the nearest neighbor. Here, the timings shown in table 7.2 have a range from 5.2 ms for downscaling from 2 cm resolution to 5 cm resolution to 415.9 ms when upscaling from 5 cm resolution to 2 cm resolution.
We can see that the trilinear interpolation takes 2-3 times longer for interpolating the chunks compared to the nearest neighbor interpolation. Therefore, the user has to make a trade off between interpolation quality and computation time. As the nearest neighbor interpolation was able to deliver satisfying optical results and prevents the washing out of colors as well, when interpolating colored chunks, the nearest neighbor interpolation should be preferred in most cases due to the better performance.

r_{source}		$r_{target} = 2\,\text{cm}$	$r_{target} = 3\,\text{cm}$			$r_{target} = 5\,\text{cm}$
	chunk size	8^3	4^3	8^3	16^3	8^3
2 cm	8^3	(1.1 ± 0.5) ms	(10.6 ± 4.1) ms	(7.8 ± 3.1) ms	(8.0 ± 3.2) ms	(2.5 ± 1.1) ms
3 cm	4^3	(13.2 ± 6.0) ms	(0.9 ± 0.4) ms	(2.0 ± 0.8) ms	(2.6 ± 1.0) ms	(1.5 ± 0.7) ms
	8^3	(37.2 ± 14.5) ms	(11.3 ± 3.5) ms	(0.5 ± 0.3) ms	(7.4 ± 2.7) ms	(3.1 ± 1.5) ms
	16^3	(148.5 ± 63.2) ms	(39.6 ± 13.3) ms	(26.7 ± 7.9) ms	(0.9 ± 0.5) ms	(17.2 ± 6.1) ms
5 cm	8^3	(161.7 ± 67.6) ms	(53.1 ± 14.8) ms	(42.5 ± 12.7) ms	(43.4 ± 13.2) ms	(0.4 ± 0.2) ms

Table 7.1: Timings for integrating all chunks of a single laser scan averaged over 5000 scans for each setup. The cases with different resolutions were aligned using nearest neighbor interpolation

r_{source}		$r_{target} = 2\,\text{cm}$	$r_{target} = 3\,\text{cm}$			$r_{target} = 5\,\text{cm}$
	chunk size	8^3	4^3	8^3	16^3	8^3
2 cm	8^3		(21.7 ± 8.4) ms	(20.2 ± 8.3) ms	(20.1 ± 8.1) ms	(5.2 ± 2.2) ms
3 cm	4^3	(32.3 ± 13.2) ms				(2.7 ± 1.3) ms
	8^3	(102.2 ± 32.2) ms				(7.0 ± 2.7) ms
	16^3	(363.0 ± 93.4) ms				(25.8 ± 7.9) ms
5 cm	8^3	(415.9 ± 111.6) ms	(120.3 ± 31.8) ms	(102.9 ± 28.5) ms	(102.5 ± 27.3) ms	

Table 7.2: Timings for integrating all chunks of a single laser scan averaged over 5000 scans for each setup using trilinear interpolation

7.4 Height Grid Map Generator

As next, the quality of the created height grid map and elapsed time for creation was evaluated. In figure 7.6, the mean error of the created height grid map and the averaged update times for different resolutions is shown. The chunks, resulting by merging 5000 single laser scans scanning the scene described in section 7.1 were transformed into the height grid map. The resulting height map was converted into a point cloud in order to compare it with the ground truth using CloudCompare by creating an "elevation point cloud" containing the highest points in a discrete grid using the ground truth point cloud.

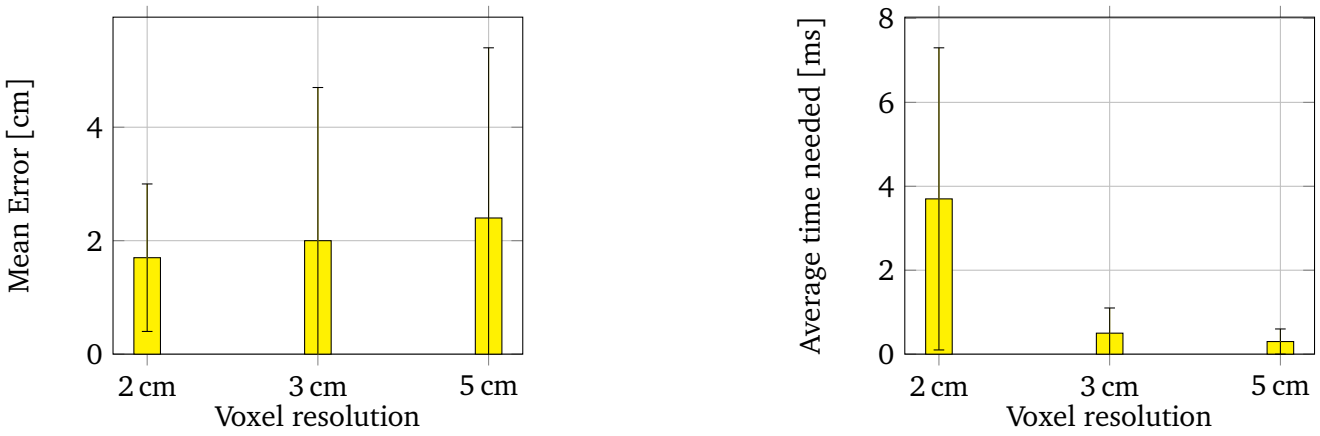
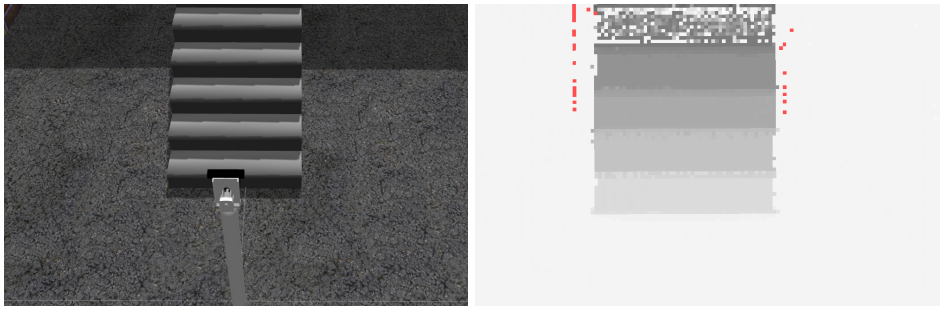


Figure 7.6: Timings for updating and resulting error for height grid map after inserting chunks 5000 times using a chunk size of 4^3

We can see, that the mean error is much higher compared to the error of the created mesh. The higher error is likely resulting due to the strong discretization during the height grid map creation.
Thanks to the incremental updates, the updating of the elevation map is easily possible in real time.

(a) The first experimental scene **(b)** The resulting height grid map



(c) The second experimental scene **(d)** The resulting height grid map

Figure 7.7: Two different experimental scenes and the resulting height grid maps

Despite the higher mean error in the elevation map of the test scene, the created height grid maps shown in figure 7.7 look very good. The base of the construction cone in figure 7.7b is good visible despite being nearly on ground level and the cone is captured as clean flow towards the center. The representation of the lower steps of the stairs in figure 7.7d reflects the steps as homogeneous areas with increasing height.

7.5 Surface Normals Estimation Generator

After addressing the elevation map creation, we tested the surface normals estimation quality and runtime in the next step. In figure 7.8, we can see that the needed time for the normals estimation update is much higher compared to the elevation map update. This is because we have to update the mesh before extracting the 3D isosurface for each updated chunk and the incremental marching cubes implementation takes some time for meshing.

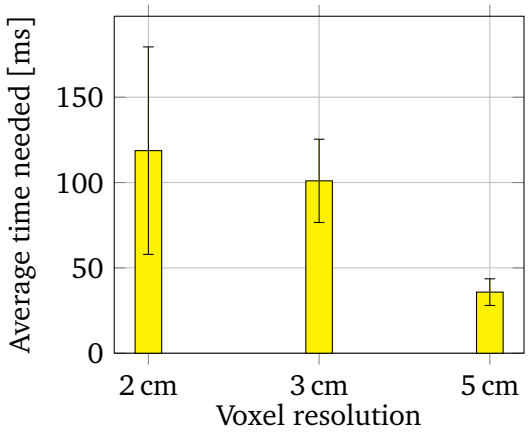
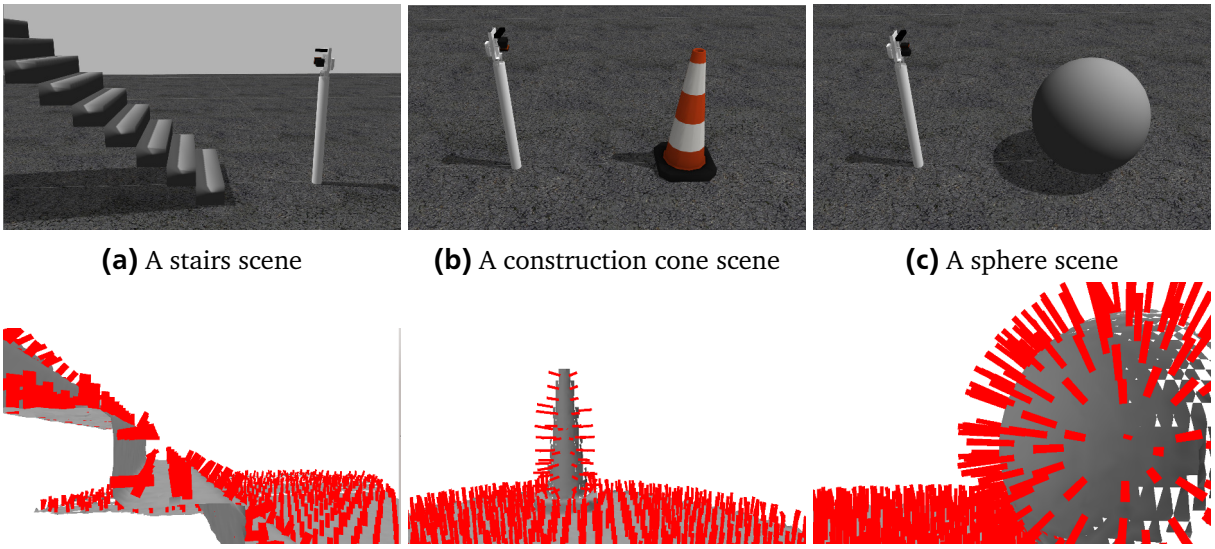


Figure 7.8: Timings for updating the surface normals estimation after inserting chunks 5000 times using a chunk size of 4^3

The resulting surface normals, shown in figure 7.9 promise very consistent surface normals.



(a) A stairs scene **(b)** A construction cone scene **(c)** A sphere scene

(d) The resulting stairs normals **(e)** The resulting cone normals **(f)** The resulting sphere normals

Figure 7.9: Different objects and their resulting surface normals estimation created at 3 cm resolution

All of the normals are oriented as expected, perpendicular to the surface. The quality of the surface normals estimation is already good, but as seen in figure 7.8, the refresh rate of the surface normals (28 Hz for 5 cm resolution to 8.5 Hz for 2 cm resolution) is below the refresh rate of the laser scanner (40 Hz) using the test setup described in section 7.1.

7.6 Fusion of Multiple Sensors

As next the fusion of multiple sensors is investigated. For this purpose, two Hector multi-sensor heads are placed in the the test scene, 0.5 m in y-direction apart and rotated 0.1π along the z-axis towards the other sensor head as seen in figure 7.10.

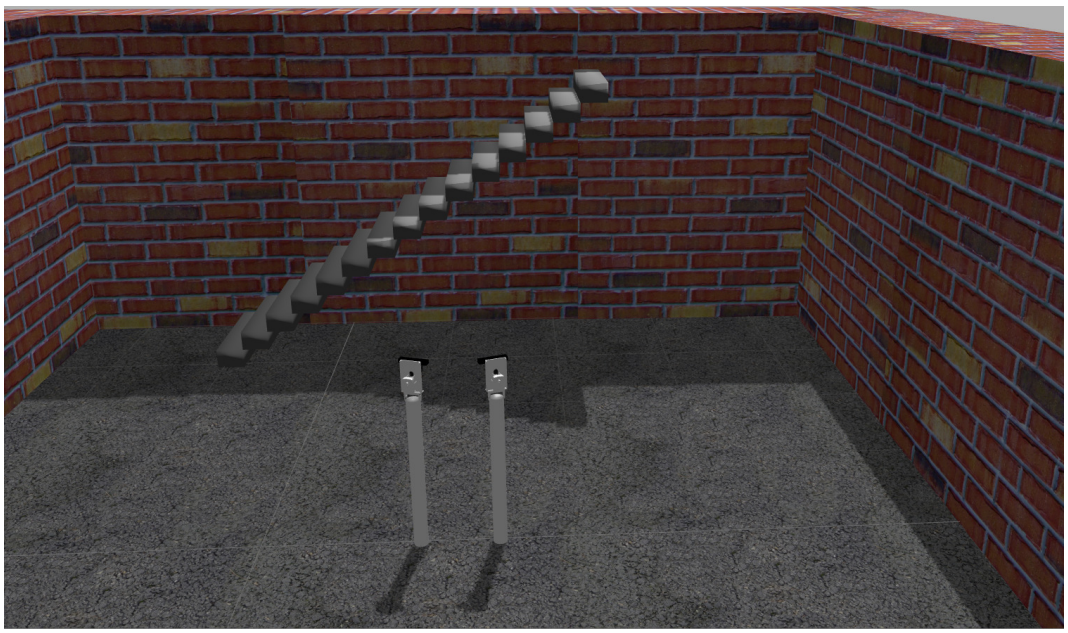


Figure 7.10: Experimental setup with two Hector multi-sensor heads 0.5 m translated in y-direction and each turned 0.1π towards the other head

Both heads are equipped with an rotating laser scanner and a RGB-D camera oriented into the direction of the stairs.

7.6.1 Two Laser scanners

In these experiments, we are only using both rotating laser scanners to fuse them and assess the resulting mean error using CloudCompare.

Correct transformation

At first, we investigate the case that both laser scanners have the correct transformation towards the world frame given.

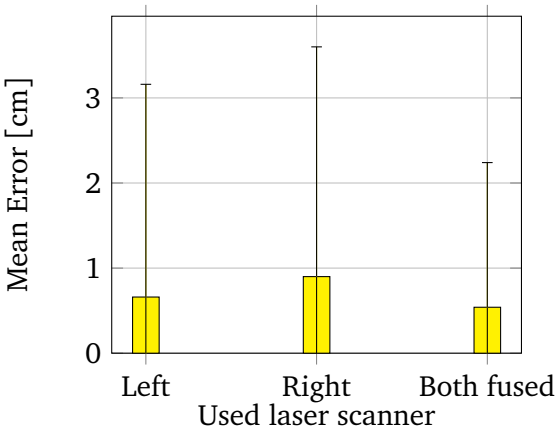


Figure 7.11: Mean error for fusing two laser scanners using a chunk size of 4^3 and 3 cm resolution

As shown in figure 7.11, the mean error of the local environment model is 0.66 cm for the left and 0.9 cm for the right laser scanner. If we fuse both local models to a global model, the mean error reduces to 0.54 cm, as the correct information of both local models is combined.

Incorrect transformation

Now, we analyze the resulting mean error for the case where only one laser scanner knows its correct position. The left laser scanner is getting a faulty position including a translational error submitted in Gazebo. The translational error is increasing from [1 cm 1 cm 1 cm] to [5 cm 5 cm 5 cm].

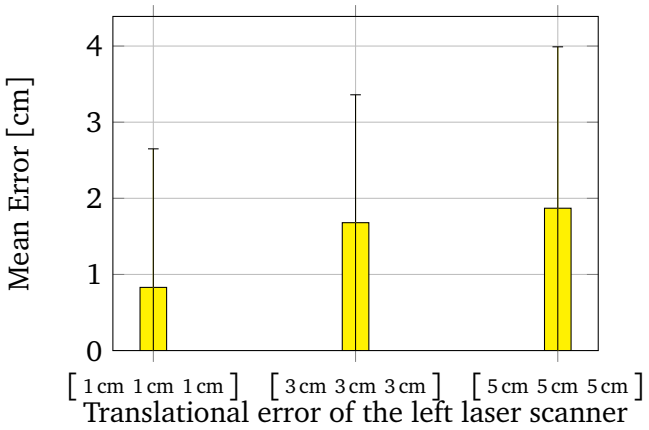


Figure 7.12: Mean error for fusing two laser scanners with incorrect transformation using a chunk size of 4^3 and 3 cm resolution

As seen in figure 7.12, the mean error increases when adding a translational error [1 cm 1 cm 1 cm] from 0.54 cm to 0.83 cm, compared to the case with correct transformations displayed in figure 7.11. By increasing the translational error to [5 cm 5 cm 5 cm] the error increases to 1.87 cm which is very good compared to the provoked error.

7.6.2 Two RGB-D cameras

Now, we are fusing both RGB-D cameras of the multi-sensor heads. Combined, the cameras are able to capture the whole wall behind the stairs.

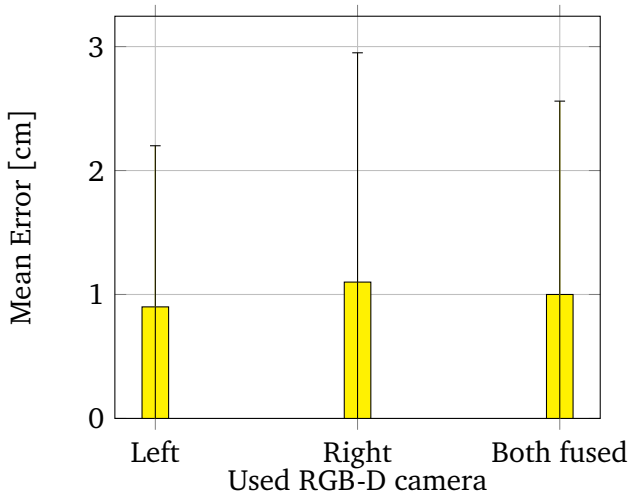


Figure 7.13: Mean error for fusing two RGB-D cameras using a chunk size of 4^3 and 3 cm resolution

The resulting mean error of a single local world model created by the RGB-D camera is in the range of 0.9 cm to 1.1 cm, see figure 7.13.
When fusing both RGB-D cameras, the resulting error is averaged to 1 cm, which is slightly worse compared to the left RGB-D camera alone, but better than the right RGB-D camera isolated.

7.6.3 One laser scanner and one RGB-D camera

Finally, we experimented with fusing one laser scanner and one RGB-D camera, testing all possible combinations of the sensors.

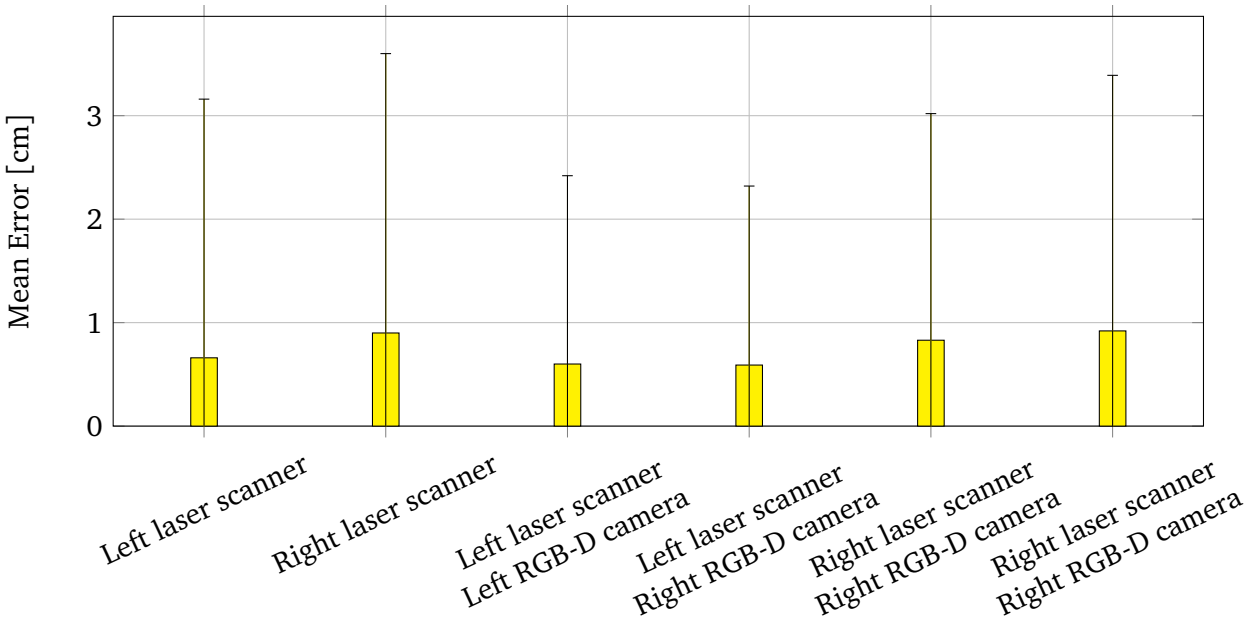
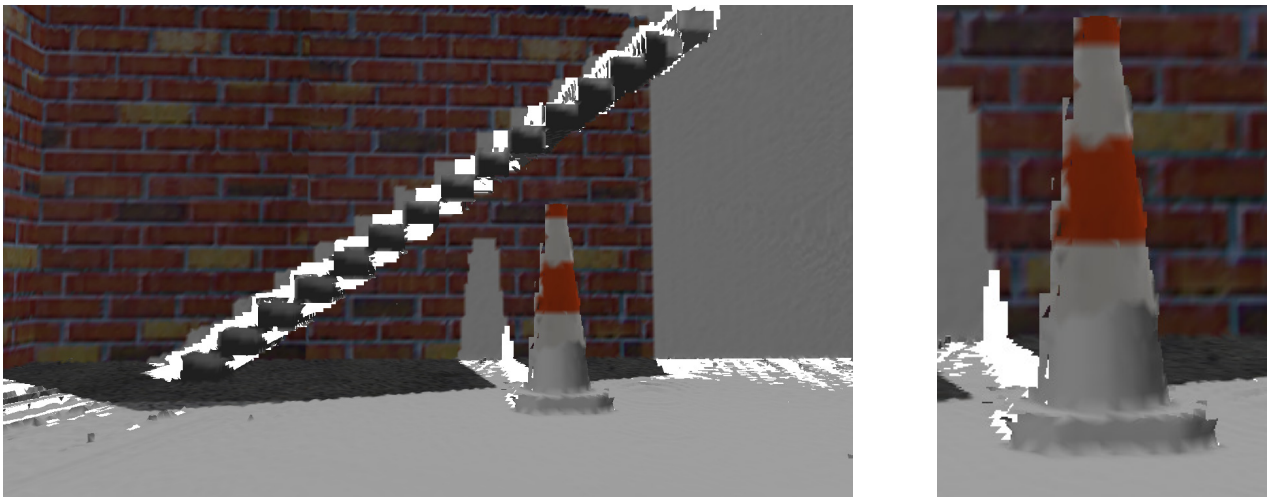


Figure 7.14: Mean error for fusing two RGB-D cameras using a chunk size of 4^3 and 3 cm resolution

When fusing the left laser scanner with an RGB-D camera – no matter which one – the error decreases compared to the error of the isolated left laser scanner (0.66 cm), as seen in figure 7.14.
In contrast, the mean error is not always decreasing when fusing the right laser scanner with the a RGB-D camera. The mean error of the isolated right laser scanner (0.9 cm) is slightly below the resulting mean error when combining the sensors of the right sensor head.



(a) A stairs scene

(b) The construction cone

Figure 7.15: Transition between the laser scanner and the RGB-D camera information

In figure 7.15, we can see the transition of the colored RGB-D data of the right camera fused with the gray data coming from the left laser scanner. The high precision of the transition is visible at the construction cone, displayed in figure 7.15b, as the white part of the construction cone based on the information from the RGB-D camera smoothly flows into the gray part based on the information from the laser scanner.
Altogether, the fusion of a laser scanner and a RGB-D camera delivers good result and can even decrease the mean error of the scene in the most cases.

8 Conclusions

The Mason framework is a highly flexible framework enabling the fusion of an arbitrary number of depth sensors. This is achieved by using the `vigir_pluginlib` for dynamically linking plugins, inheriting the `ProcessingPlugin` into the core framework. Therefore the user is able to change the behavior of the framework during runtime by exchanging or adding plugins, for example by replacing the fusion plugin by another using a different fusion approach. Despite of the implemented plugins in this work, in general all implemented plugins can be directly used for each system deployment in any combination of further plugins due to the modular design of the framework.
The first set of plugins for the Mason framework uses the spatially hashed TSDF fusion approach using and extending the OpenCHISEL library. Thanks to the spatially hashed TSDF concept the real time integration of measurement data, e.g. from laser scanners or RGB-D cameras, is possible while keeping the memory and CPU requirements low. At the current state, the integration of depth images and point clouds is supported, but the extension to more sensor types is simple. By improving the method for point cloud integration in OpenCHISEL a great speedup, especially for small chunk sizes, was achieved.
The OpenCHISEL based plugin set uses consequently incremental updates of chunks for the update procedures of the fusion plugin and the generator plugins and thus avoids unnecessary computation. As the fusion plugin aligns the chunks first, the fusion of multiple sensors is possible even for differing chunk size and resolutions.
It has been shown that the elevation map generator, mesh generator and surface normals estimation generator can be used to create different representations based on the TSDF data, which are suitable for example for high level planning software like footstep planning.

8.1 Limitations

At the current state, only depth images and point clouds can be processed by the Mason framework, respectively the OpenCHISEL based fusion plugin. But by converting the sensor data to a point cloud first, more sensor data types can be supported automatically.
The update rate for integrating depth images on 3 cm resolution without meshing is currently only 4.0 Hz for uncolored and 3.1 Hz for colored images from the RGB-D camera using the system described in section 7.1. This drawback is caused by using the CPU for processing the updated data that is provided with 30 Hz by the RGB-D camera that sums up to 9216000 data points have to be considered every second.
Additionally, the surface normals estimation update is only possible with refresh rates ranging from 29 Hz for 5 cm resolution to 9 Hz for 2 cm resolution when using the laser scanner – working with 40 Hz – as data source. This is caused by the required application of the incremental marching cubes implementation for meshing the changed chunks before being able to compute the surface normals.

8.2 Outlook

Currently, OpenCHISEL is using its own internal data types for the sensor data and transformation. By adapting OpenCHISEL to use the ROS data types, the data would no longer need to get converted before fusion; that would prevent unnecessary duplication of data.
For now, the framework is using only the CPU to merge sensor data, in order to maximize the compatibility for common robot systems. But as the framework is designed to add new plugins easily, a fusion plugin using GPU acceleration for integrating sensor data into TSDF data could speed up the process even more, especially for RGB-D cameras due to the high update rate of the sensor data.
Additionally, the space carving routine for considering dynamic objects in the scene could be adapted even more. For now, voxels in the space carving region get deleted instantly after only one sensor ray has passed the voxel. This behavior of space carving is problematic for noisy sensors, as faulty measurements could result in deleting valid parts of the scene. Therefore, the behavior routine could be improved by reducing the weight of by sensor rays passed voxels in each iteration instead of instantly removing them. All voxels with a weight $W \leq 0$ should be identified and removed afterwards.
In order to improve the work flow with other ROS packages, a plugin providing the import and export of ROS octomaps – a commonly used ROS octree data structure – is desirable for the framework.
As the framework still relies on an external pose estimation defining significantly the quality of the environment model, the usage of a scan matching algorithm or other approaches can be used to estimate and track the sensor's position which should improve the overall performance for moving sensor systems, e.g. a walking robot. However, because of the plugin system this feature can be added easily to the system without touching existing code.
Finally, the mason framework will be published as open source ROS stack that allows the community to use it as base for their environment modeling system.

Acronyms

3D three-dimensional

CUDA Compute Unified Device Architecture

CPU Central Processing Unit

DARPA Defense Advanced Research Projects Agency

ICP Iterative Closest Point

IMU inertial measurement unit

GPU Graphics Processing Unit

RGB-D Red-Green-Blue-Depth

ROS Robot Operating System

SDF signed distance function

ToF time of flight

TSD truncated signed distance

TSDF truncated signed distance function

URDF Unified Robot Description Format

List of Figures

List of Algorithms

Bibliography

[1] Alexander Stumpf, Stefan Kohlbrecher, David Conner, and Oskar von Stryk. Supervised Footstep Planning for Humanoid Robots in Rough Terrain Tasks using a Black Box Walking Controller. In *2014 14th IEEE-RAS International Conference on Humanoid Robots*, pages 287–294. IEEE, 2014.

[2] Alberto Romay, Stefan Kohlbrecher, David C Conner, Alexander Stumpf, and Oskar von Stryk. Template-Based Manipulation in Unstructured Environments for Supervised Semi-Autonomous Humanoid Robots. In *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, pages 979–986. IEEE, 2014.

[3] Alberto Romay, Stefan Kohlbrecher, David C Conner, and Oskar von Stryk. Achieving Versatile Manipulation Tasks with Unknown Objects by Supervised Humanoid Robots based on Object Templates. In *Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on*, pages 249–255. IEEE, 2015.

[4] Stefan Kohlbrecher, Alberto Romay, Alexander Stumpf, Anant Gupta, Oskar von Stryk, Felipe Bacim, Doug Bowman, Alex Goins, Ravi Balasubramanian, and David Conner. Human-Robot Teaming for Rescue Missions: Team ViGIR's approach to the 2013 DARPA Robotics Challenge Trials. *Journal of Field Robotics*, 32(3):352–377, 2015.

[5] Brian Curless and Marc Levoy. A Volumetric Method for Building Complex Models from Range Images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312. ACM, 1996.

[6] Matthew Klingensmith, Ivan Dryanovski, Siddhartha Srinivasa, and Jizhong Xiao. Chisel: Real Time Large Scale 3D Reconstruction Onboard a Mobile Device. In *Robotics Science and Systems 2015*, July 2015.

[7] Richard Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. KinectFusion: Real-Time Dense Surface Mapping and Tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011.

[8] Dmitry Trifonov. Real-time High Resolution Fusion of Depth Maps on GPU. *Computing Research Repository*, 2013. URL `http://arxiv.org/abs/1311.7194`.

[9] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. Real-time 3D econstruction at Scale using Voxel Hashing. *ACM Transactions on Graphics (TOG)*, 32, 2013.

[10] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomeranets, and Markus Gross. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *VMV*, volume 3, pages 47–54, 2003.

[11] John Amanatides, Andrew Woo, et al. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics*, volume 87, page 10, 1987.

[12] William Lorensen and Harvey Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.

[13] Jean-Marie Favreau. Marching Cubes, March 2016. URL `https://commons.wikimedia.org/wiki/File:MarchingCubes.svg`.

[14] Chuong Nguyen, Shahram Izadi, and David Lovell. Modeling Kinect Sensor Noise for Improved 3D Reconstruction and Tracking. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2012 Second International Conference on*, pages 524–530. IEEE, 2012.

[15] François Pomerleau, Andreas Breitenmoser, Ming Liu, Francis Colas, and Roland Siegwart. Noise Characterization of Depth Sensors for Surface Inspections. In *Applied Robotics for the Power Industry (CARPI), 2012 2nd International Conference on*, pages 16–21. IEEE, 2012.

[16] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[17] David Conner, Stefan Kohlbrecher, Alberto Romay, Alexander Stumpf, Spyros Maniatopoulos, Moritz Schappler, and Benjamin Waxler. Team ViGIR: DARPA Robotics Challenge. Technical report, DTIC Document, 2015.

[18] Eitan Marder-Eppstein, Tully Foote, Dirk Thomas, and Mirza Shah. Robot Operating System pluginlib, March 2016. URL `http://wiki.ros.org/pluginlib`.

[19] Maurice Fallon, Pat Marion, Robin Deits, Thomas Whelan, Matthew Antone, John McDonald, and Russ Tedrake. Continuous Humanoid Locomotion over Uneven Terrain using Stereo Fusion. In *International Conference on Humanoid Robots*, Seoul, Nov 2015.

[20] Thomas Whelan, Michael Kaess, Maurice Fallon, Hordur Johannsson, John Leonard, and John McDonald. Kintinuous: Spatially Extended KinectFusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, Sydney, Australia, Jul 2012.

[21] Péter Fankhauser, Michael Bloesch, Christian Gehring, Marco Hutter, and Roland Siegwart. Robot-Centric Elevation Mapping with Uncertainty Estimates. In *International Conference on Climbing and Walking Robots (CLAWAR)*, 2014.

[22] Stefan May, Philipp Koch, Rainer Koch, Christian Merkl, Christian Pfitzner, and Andreas Nüchter. A Generalized 2D and 3D Multi-Sensor Data Integration Approach based on Signed Distance Functions for Multi-Modal Robotic Mapping. In *VMV*, pages 95–102, 2014.

[23] Diana Werner, Ayoub Al-Hamadi, and Philipp Werner. Truncated Signed Distance Function: Experiments on Voxel Size. In *Image Analysis and Recognition*, pages 357–364. Springer, 2014.

[24] Daniel Girardeau-Montaut. CloudCompare, April 2016. URL `http://www.danielgm.net/cc`.