Mesh Addition Based on the Depth Image (MABDI)

by

Lucas E. Chavez

B.S., Mechanical Engineering
New Mexico Institute of Mining and Technology, 2009

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Mechanical Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2016

Acknowledgments

This work was supported in part by Sandia National Laboratories under Purchase Order: 1179196 and NSF grant OISE #1131305.

Mesh Addition Based on the Depth Image (MABDI)

by

Lucas E. Chavez

B.S., Mechanical Engineering
New Mexico Institute of Mining and Technology, 2009

M.S., Mechanical Engineering, University of New Mexico, 2016

Abstract

Many robotic applications utilize a detailed map of the world and the algorithm used to produce such a map must take into consideration real-world constraints such as computational and memory costs. Traditional mesh-based environmental mapping algorithms receive data from the sensor, create a mesh surface from the data, and then append the surface to a growing global mesh. These algorithms do not provide a computationally efficient mechanism for reducing redundancies in the global mesh. MABDI is able to leverage the knowledge contained in the global mesh to find the difference between what we expect our sensor to see and what the sensor is actually seeing. This difference between expected and actual allows MABDI to classify the data from the sensor as either data from a novel part of the environment or data from a part of the environment we have already seen before. Using only the novel data, a surface is created and appended to the global mesh. MABDI's algorithmic design identifies redundant information and removes it *before* it is added to the global mesh. This reduces the amount of memory needed to represent the mesh and also lessens the computational needs to generate mesh elements from the data.

Contents

1 Introduction **1**

1.1 Overview 1

1.1.1 RGB-D Sensors 2

1.1.2 Maps 3

1.2 Goal 4

1.3 Contribution 4

2 Related Works **6**

2.1 SLAM 7

2.1.1 Point Locations 8

2.1.2 Volumetric 10

2.1.3 Surface 12

2.2 Surface Reconstruction 15

2.2.1 Volume-based 16

2.2.2 Surface-based 18

2.3 Summary 21

3 Approach **22**

3.1 Algorithmic Design 22

3.2 Implementation 24

3.2.1 Surface Reconstruction 24

3.2.2 Software Design 28

4 Experimental Setup **32**

4.1 Simulation Overview 33

4.2 Simulating a RGB-D Sensor 33

4.2.1 Rendering Pipeline 33

4.2.2 Adding Noise to the Depth Image 34

4.3 Sensor Path 36

4.4 Simulation Parameters 37

5 Results **39**

5.1 MABDI Performance During Experiments 40

5.1.1 Experiment 1 40

5.1.2 Experiment 2 42

5.1.3 Experiment 3 43

5.2 Global Mesh Results 45

5.2.1 Mesh Quality 45

5.2.2 Mesh Progression 46

6 Conclusion **50**

A MABDI code **51**

A.1 FilterDepthImage.py 51

A.2 FilterClassifier.py 56

A.3 FilterDepthImageToSurface.py 59

A.4 FilterWorldMesh.py 64

References **66**

Chapter 1

Introduction

1.1 Overview

Many robotic applications, especially those that involve human-robot interaction, often require a rich representation of the environment in order to perform such behavior as path planning and obstacle avoidance. In general, a rich representation, or map, is useful for providing situational awareness to an autonomous agent. A map is also important for applications such as teleoperation [1].

In robotics, map building in an unknown environment is referred to as the Simultaneous Localization and Mapping (SLAM) problem [2]. This label describes the fact that a methodology which solves the SLAM problem must simultaneously locate the robot in the environment as well as map the environment. The focus of this work is the mapping aspect of the SLAM problem. Fig. 1.1 gives a visualization of the goal.

The methodology to build a map is a continuously evolving subject in the field of robotics and computer graphics. Well known works of map building methods be-

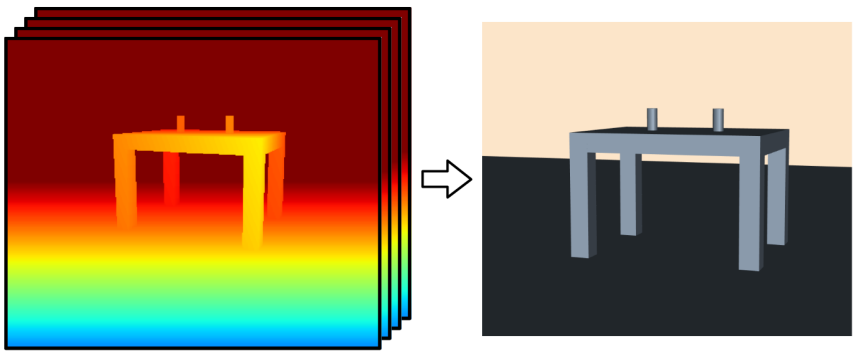


Figure 1.1: Goal is to create a map from depth images.

gan to be seen around 1987 [3]. Since then, the methods and the representations themselves have continued to evolve at an impressive rate. Growth in this field of research has been fueled by continuous advances in computing and sensing technologies. Over the years, sensors have continued to generate measurements at higher rates, higher resolution, and lower cost. RGB-D sensors are a new category of sensor that have recently gained extensive popularity in the robotics community due to their affordability and ability to generate a rich amount of data.

1.1.1 RGB-D Sensors

The popularity of RGB-D sensors began with the release and commercialization of the Kinect^{TM} by Microsoft. The arrival of the Kinect brought with it an inexpensive depth sensor that uses an active range system to generate a depth map of a given environment [4]. The Kinect and similar sensors, have come to be called RGB-D sensors. This class of sensors provide images which include both visual (RGB) and depth (D) values. Several works have taken advantage of this sensor technology in scenarios such as environmental mapping [5], 3D reconstruction [6], gesture recognition [7], and altitude control of aerial vehicles [8].

RGB-D sensors generally provide data at 30 frames per second and 640×480 resolution. Consequently, methods that use RGB-D data must handle over 9 million pixel values per second, if only using the depth information (D), and over 18 million if using both color (RGB) and depth (D). The amount of data output from RGB-D sensors creates the need for mapping methods that are computationally inexpensive and also influences the type of data structure used to store the map.

1.1.2 Maps

There are different types of data structures that can define a map. All types have both intrinsic characteristics that impact the algorithms that generate them and constraints that must be considered for real-world applications. In addition, we are concerned with rich representation types, in contrast to sparse representation types [9], because rich types have the most use in applications such as human-robot interaction.

Table 1.1: Comparison of constraints for different map types.

	Supported	Computationally Inexpensive	Low Memory Requirement
Point Clouds	x	x	-
Surfels	-	x	x
Implicit Functions	x	-	-
Mesh	x	x	x

When considering which type of map is best for real-world applications, we must consider the constraints imposed by each type:

- Supported - Is there software, tools, research, algorithms, etc., for this type of map?

- Computationally Inexpensive - Can the algorithms run quickly on low cost computers (rather than specialized hardware)?
- Low Memory Requirement - Can the algorithms run on hardware with a standard amount of RAM?

Table 1.1 compares the constraints of common map types. We can see, in general a mesh type map satisfies real-world constraints. Additionally, meshes have been used extensively by the gaming and graphics communities, and so benefits from an incredible amount of continued research and advances in hardware such as Graphics Processing Units (GPUs).

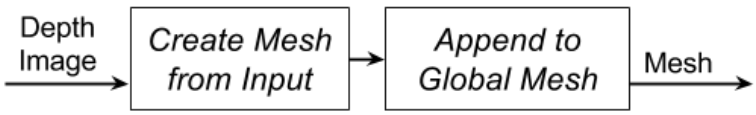
1.2 Goal

The goal of this work is to develop a mapping algorithm that can gracefully utilize the amount of data output from an RGB-D sensor. Additionally, the algorithm will make use of software tools and hardware that have been developed for mesh data structures. The algorithm will be able to make intelligent decisions using the data it receives based on the knowledge it has been building about the environment. The decisions will be driven by the leveraging the difference between what the algorithm is actually seeing and what it expects to see. The decisions will be generated using computationally inexpensive computer vision methods.

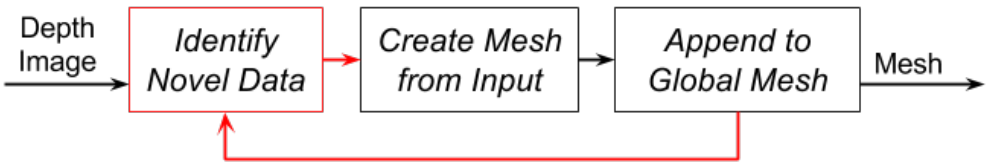
1.3 Contribution

MABDI's contribution to the state-of-the-art in mesh based environmental mapping is closing the loop of the algorithmic structure used by current methods. Fig. 1.2a

shows the structure of current methods. Data comes in from the sensor, those measurements are used to create a mesh, and then that mesh is appended to a global mesh. We can then compare the structure of current methods to the structure used in MABDI, shown in Fig. 1.2b. Both structures have the "Create Mesh from Input" component. The input to this component is different for current methods and MABDI. Current methods input all data from the sensor whereas MABDI only inputs data identified to be from the unknown parts of the environment. The MABDI algorithm is able to identify this data by leveraging the knowledge contained in the Global Mesh and intelligently categorizing the incoming data. This categorization of the incoming data closes the loop of the algorithmic structure used by current methods and is the contribution of MABDI to the state-of-the-art.



(a) Current methods



(b) MABDI

Figure 1.2: Algorithmic structure of current methods (a) and MABDI (b). Contribution of MABDI to the state-of-the-art shown in red

Chapter 2

Related Works

A major problem in robotics has been and continues to be: How can we create the "best" representation of an unknown environment? There are two main communities of researchers who been working on developing algorithms and methods to answer precisely this question. They are the robotics community and the computer graphics community, but each community has a slightly different motivation for solving this problem. The robotics community is concerned with developing a real-time solution for generating representations in large environments. In the literature, large environments usually range in size from multi-room office spaces to a few square miles on city streets. These representations are used by both fully autonomous and teleoperated systems. The common name which is used by the robotics community for this problem is Simultaneous Localization and Mapping or SLAM. The name SLAM refers to the problem of mapping and locating a robot in an unknown environment. Early methods generated very sparse representations of the world, as time and sensor technology progressed the representations became denser. A dense representation is desired for any system that must have good situational awareness of its environment. The computer graphics community is concerned with generating high quality representations of small environments. In the literature, small environments usually

range in size from a cubic meter to room size. They generally refer to the problem as surface reconstruction. These representations are used by augmented reality, computer game object creation, 3D printing, and other applications. In the following sections we will trace the development of representation generating methods in both communities.

2.1 SLAM

The problem of SLAM has been a primary focus of the robotics community for more than 25 years. A complete solution to the SLAM problem must be able to generate a representation of an unknown environment and track the robot in this new representation. In this body of literature the act of generating a representation is referred to as mapping. A good overview of the problem can be found in [10] and [11]. Each solution is designed to consider the goal application, type of sensor, computational constraints, and memory limits. All these factors influence the researcher's choice of which type of representation to use for the mapping procedure. In 2002 Thrun wrote a famous survey [2] of the SLAM literature which categorized existing algorithms on many traits including the representation. The representation choice of prior work can be roughly categorized into three types. The first type is characterized by some sort of list of 2D or 3D points and are usually considered to be sparse representations. Common names for these types are landmark locations and point clouds. The second type is considered to be more volumetric based and is often times considered to be a dense representation. Common names for these types are occupancy grid and Truncated Signed Distance Function (TSDF). The last type has the characteristic of being a surface representation and is also considered to be a dense representation. Common names for these types are surfels and mesh. In the following sections we will trace the history of each of the three types of representation that are seen in the

SLAM literature.

2.1.1 Point Locations

One of the most well known and earliest solution to the SLAM problem, which uses a point location representation, was proposed by Smith et al. in 1990 [12]. The mathematical framework that he created was the origin of a family of solutions based on the Extended Kalman Filter (EKF). The representation he chose was simply a list of 2D landmark locations. Each location was part of a state matrix which was estimated at every iteration. A list of landmark locations was chosen because it allowed the method to have a low computational cost and use a small amount of memory, important factors in the days of early computing. There have been many improvements to the family of SLAM solutions which generate a list of point locations since Smith's work. One of the first practical implementations on a real robot was done by Thrun in 1998 [13]. In this work the SLAM problem was posed in an Expectation Maximization (EM) framework which is similar to the EKF framework in that landmark locations are saved in a state vector which is estimated at every iteration. In Thrun's work an occupancy grid map is generated as a post processing step from sonar measurements. The results showed that their representation could become more accurate over time by using new observations to improve the current estimate. This a highly desired ability of any representation generation method. The next step was the ability of these methods to include a loop closure procedure. A loop closure procedure was proposed by Gutmann in 1999 [14]. The key ability of the method was it could recognize when the robot was revisiting a prior location and adjust the entire representation with the constraint that the two points must coincide. In 2001 Dissanayake et al. [9] derived three theorems to theoretically prove the convergence of the SLAM problem. Their test platform used a millimeter-wave radar mounted on a vehicle and generated a list of 2D landmark locations.

In 2001 Thrun et al. [15] cast the SLAM problem using particle filter techniques. Their results generated a 2D map and showed an increased robustness and lower computational cost than prior methods. One of the key disadvantages of methods up to this point was that complexity scaled quadratically with the number of landmark locations. In 2002 Montemerlo et al. [16] created a SLAM solution named FastSLAM, which was able to handle a much larger number of landmarks. They showed results with maps containing more than 50,000 points. Then, SLAM solutions using point locations became much more directed towards 3D.

Some of the first interesting works that represented the world as a list of 3D point locations were done by Thrun et al. in 2000 [17], Liu and Emery in 2001 [18], and Hähnel et al. in 2003. In these works the 2D landmark locations and robot position were estimated using techniques from Thrun's past work [13]. Once this had been done the 3D laser scan data was simply appended to each estimated robot location. Then, a mesh was created by post processing the 3D point cloud. Their works utilized the fact that the laser collected the data in an incremental manner and simply connected neighboring 3D points. Finally, the mesh was simplified by looking for large planar sections and merging the corresponding mesh elements. One of the first SLAM solutions which used a single camera to generate a list of 3D points was done by Davison in 2003 [19]. Here he used a single camera to generate a very sparse list of 3D points. This method was limited to small environments. Future advances allowed representations of larger environments. In 2003 Thrun et al. [20] created a SLAM procedure which did not rely on having a structured environment and was applied to mapping large mines. In 2004 Howard et al. [21] created a SLAM system based on a Segway platform equipped with a 3D laser which could map areas of roughly 0.5 km on each side. One of the results showed a map with approximately 8 million points. In 2006 Cole and Newman [22] continued work in large-scale SLAM by increasing robustness and also generated maps with many 3D points using a laser sensor. In 2007 Clemente et al. created a large-scale SLAM system that used a single

camera. The system had an advanced loop closing procedure based on visual features and created large maps of 3D points. In 2001 Klein and Murray [23] developed a SLAM solution which used a single camera. The uniqueness of their method was the algorithmic structure. Their SLAM solution consisted of two separate processes: a tracking process and a map building process. This algorithmic structure has become very common in many current SLAM solutions because of the advances in pose estimation technology. Klein and Murray were able to get very good results for a small environment and showed Augmented Reality (AR) applications. Many of the future advances of SLAM solutions, which generated 3D point sets, dealt with camera systems [24, 25, 26] and improved speed and robustness. Most current methods that produce a list of points use a relatively new type of sensor named a RGB-D sensor. One good example is a work that was produced in 2011 by Engelhard et al. [27]. In this work they used an algorithm named the Iterative Closest Point (ICP) [28] to align point clouds coming from the RGB-D sensor into a large colored point cloud. The resulting maps were visually impressive. However, the map could not be adapted to new information and was not well suited for other applications, such as obstacle avoidance. These limitations are inherent in maps that consist of lists of points.

2.1.2 Volumetric

Many SLAM solutions generate a 2D volumetric representation of the world because they are especially advantageous in dealing with noisy sensors. Two of the first major works that generated a 2D volumetric representation were done in 1998 by Yamauchi et al. [29] and Schultz et al. [30]. These works generated a 2D occupancy grid, which is a type of volumetric representation. Here the environment was divided into a 2D grid. Each square of the grid contained the probability that it was occupied with an object. All squares would be updated iteratively based on the current sensor readings. Occupancy grids, like any other volumetric-based representation, are limited by the

amount of available memory. In 2002 Biswas et al. [31] extended occupancy grid methods by allowing dynamic environments. This was done by looking at past "snapshots" of the map. In 2004 Eliazar and Parr [32] continued the advancement by decreasing computational cost and implemented a loop closure method.

There have been a few impressive SLAM solutions that generate a 3D volumetric representation. There are three major works that generated a result very similar to a 3D occupancy grid, which was saved as in a octree data structure [33, 34, 35, 36]. Each work had a slightly different name and procedure for generating the representation, but in general the representations divided the environment into cubes and had a scalar value representing the belief of a surface being there. Octrees were used to save memory by only having a fine resolution of cubes at places where there was a surface. There are many advantages to a 3D occupancy grid representation. The representation is well suited for obstacle avoidance and path planning applications. Also, the representation is very adaptable to new information. The major disadvantage is that the representation can not be visualized immediately. In order to render, an image must be generated at each desired viewpoint by ray tracing the volume. This can be a problem when using such method for applications such as teleoperation due to the computational cost of rendering. The current state of the art for generating a volumetric representation was done by Newcombe et al. in 2011 [6]. Their system used a RGB-D sensor and generated a 3D voxelized grid Truncated Signed Distance Function (TSDF) of the environment. For this type of representation each cube contains the value of the distance to the nearest surface. The sign of the value is based on which side of the surface the cube is relative to the sensor. This work has been the most capable at dealing with extremely noisy data and dynamic scenes. However, due to memory constraints the method can only represent environments that are about the size of a 4m cube. Also, it must be ray traced in order to be visualized.

2.1.3 Surface

One of the first major works that created a surface representation of the environment in real-time was done by Martin and Thrun in 2002 [37]. Their method utilized an EM framework to fit plane models to 3D point cloud data. Polygon mesh elements were then easily assigned to each plane. The main drive behind this work was to generate a map of the environment that uses a small amount of memory. Their method worked well for structured environments. One of the major limitations of their method, and other methods that only mesh large planar sections, is that the representation will only consist of planar sections and not capture the fine detail of the environment. In 2004 Viejo and Cazorla [38] developed a methodology for generating a mesh that can contain more information of the environment than large planar sections. Due to this ability, they termed their method to be "unconstrained." Essentially their method was based on a 3D Delaunay triangulation algorithm. Giesen surveyed Delaunay triangulation methods in [39]. Viejo and Cazorla were not able to obtain real-time results and, in fact, it has been seen that it is extremely difficult to run a 3D Delaunay triangulation in real time because of the numerous distance calculations required. One of the next major advances came from Weingarten and Siegwart in 2006 [40]. Their work also created a mesh that was only capable of capturing large planar surfaces. However, they showed increased robustness. In 2007 Pollefeys et al. [41, 42] developed a large urban mapping system consisting of a vehicle and eight camera systems. The processing was carried out by multiple CPUs and optimized for speed with Graphics Processing Unit (GPU) calculations. In their work they used the camera systems to generate an initial set of depth maps. This set was then reduced using their depth map fusion method. The method combined multiple depth maps to reject erroneous depth estimates and remove redundancy from the data, resulting in a reduced set of depth maps that was more accurate than the initial set of depth maps. The reduced set was then used by a triangulation procedure to create a mesh

of the environment. The mesh generation procedure was based on a work from 2002 by Pajarola et al. [43]. This method defines a mesh in the depth image. It starts from a very coarse mesh and continues to refine in areas of the depth image based on a confidence criteria. In the work of Weingarten and Siegwart, these meshes that are defined for each fused depth image are then checked for overlaps and duplicates are removed to make a single large mesh. One of the major drawbacks of this approach is that the output mesh can not be adapted by measurements that come from revisited parts of the scene. Another major advancement came in 2008 from Poppinga et al. [44]. In this work they used a Time of Flight (ToF) camera to generate a mesh representation of the large planar structures in the environment. Here they also develop a procedure to determine a mesh in a depth image. They leverage the structure of the depth image to make the method computationally inexpensive. In their work they simply append the meshes that are created from each depth image into a global coordinate system. They obtain very good results from a simple method. However, the method is not adaptive to new information. Also, a mesh is created for each depth image instead of updating and maintaining a global mesh. A major advancement came from work done by Newcombe and Davison in 2010 [45]. In this work they designed a method to create a mesh reconstruction from a single video camera. Their method used Structure From Motion (SFM) to obtain a sparse point cloud of the scene. Then an implicit function was fit to the point cloud using the methodology of Ohtake et al. [46]. A bundle of depth maps is then selected. From the bundle a single reference depth image is selected and a "base" model is constructed by sampling the implicit surface for vertices in the reference frame. The neighboring frames are used to better the "base" model and create a more accurate mesh. Each reference frame has its own mesh and all the meshes are put into a global coordinate system. Duplications are then detected and removed. Again, the representation is not adaptive to new information. In 2010 Stühmer et al. [47] generated very accurate depth maps from several color images in real-time.

They showed very impressive results but their method was not designed to maintain a representation in a global coordinate frame.

The next major advances in methods that generated surface representations of the environment, were based on RGB-D sensors. This type of sensor has become very popular since the release of the Kinect from Microsoft that was the first mass produced RGB-D sensor of its kind. RGB-D sensors are inexpensive and produce noisy 640x480 depth images at 30Hz. The RGB-D sensor has excited the robotics community because this has been the first time that depth data has been so readily accessible from such an inexpensive sensor. Therefore, these methodologies must be able to quickly deal with very high rates of information. One impressive work came from Henry et al. in 2012 [48]. In this work they designed a system that used a RGB-D sensor to build a map made of surfels (Surfels are circular disks which have a particular position and orientation and also a radial size based on confidence.). In order to generate and maintain the surfel map they used the work of Weise et al. [49]. The map consists of a large number of surfels. The surfel map can be updated given new registered depth images from the sensor. Decisions are made how to handle each measurement in the depth image based on the difference between an expectation generated using the current map and the actual readings from the sensor. Rendering a surfel map requires special methods [50] and is difficult to use in applications such as obstacle avoidance.

One of the next major advances is a highly-related work that was published by Whelan et al. in 2012 [51] and more recently in 2013 [52]. The system they developed was named Kintinuous and was able to produce a high quality mesh representation of the environment. Their hybrid system utilized the KinectFusion method [6] of Newcombe et al. to create a volumetric representation of the portion of the environment in front of the sensor. As the sensor moves, portions of the environment that leave the volume in front of the sensor are ray cast and turned into a mesh. They

obtain very impressive results but also mention a limitation of their system for future work. The limitation is that the mesh can not be updated once created, which is an issue when revisiting parts of the environment that may have changed. One of the most impressive current works which has an adaptable mesh came from Cashier et al. in 2012 [53]. In this work, they were able to generate and update a mesh with new measurements from a ToF sensor. They used the difference between the existing model and the actual measurements to decide whether to adapt the mesh or add new elements. The mesh topology was not adaptive to the environment and their experiments only showed results of mapping a single flat wall with no robot movement. The system needs to be tested for object addition and removal.

2.2 Surface Reconstruction

The computer graphics field has spent considerable effort to develop methodologies for creating representations from sets of data. Generally, these sets of data are acquired from a sensor. Methodologies have progressed steadily and are often designed for a specific application. One of the original motivations was to generate surfaces from medical imaging data. This improves a doctor's decisions because the data are presented in a more intuitive manner. Current applications include augmented reality and 3D printing. Older methodologies were not as concerned with speed and often times had a large computational cost. Also, the methodologies are often designed for single objects or small environments. Following the taxonomy of such well known works as [54, 55], the field can be roughly divided into representations that are generated with volume-based techniques and those that use surface-based techniques. Methods that use volume-based techniques are characterized by spatially subdividing the environmental volume and are usually computationally expensive and require a large amount of memory. Methods that use surface-based techniques

generate the representation using surface properties of the input data. Both types of methods can have mechanisms to adapt the mesh to noisy or new information. In the following section we will trace the progression of the methodologies.

2.2.1 Volume-based

Volume-based methods have the characteristic of spatially subdividing the volume into smaller parts. One of the first well known works that used a volume-based technique was proposed by Lorensen and Cline in 1987 [3]. In this work they proposed a method named marching cubes, which is still known for its reliability and simplicity and is used by applications that do not have a computational requirement. Marching cubes subdivides the space into cubes. The data contained in each cube dictate how the surface connectivity will be defined in that cube. Possible vertex locations are at the corners and along the edges. Once this has been done for all cubes the process is complete. One of the next major steps came from Hoppe et al. in 1992 [56] In this work they used the input points to define a Signed Distance Function (SDF) in 3D space and then meshed the zero-set to obtain the output mesh. A SDF is a spatial function that has the value of the distance to the nearest surface at each point. The sign is used to specify if the point is inside or outside of the surface relative to the sensor. The zero-set of the SDF is the surface where the values transition from positive to negative. Using a SDF has proven to be very effective and has been the core idea of many methodologies that came after this work of Hoppe et al., such as KinectFusion [6]. One of the next advances came from Edelsbrunner and Mücke in 1994 [57] with a method named alpha shapes. They used 3D Delaunay triangulation and the input point set to decompose the volume into a Delaunay tetrahedrization. This gives a triangulation of the input set which involves all points. A sphere of radius alpha is then used to remove edges and vertices to obtain a mesh of user specified resolution. Many works have made use of 3D Delaunay triangulation to create a

mesh. Methods which use 3D Delaunay on the input set have a large computational cost and often cannot be executed in real-time. The next valuable contribution came from Bloomenthal in 1994 [58] as open source software for surface polygonization of implicit functions. This was a stable and robust open source software that has been used in many well-known algorithms [45]. Another major advance came from Curless and Levoy in 1996 [59]. In this work they also constructed a Truncated Signed Distance Function (TSDF). A TSDF is very similar to a SDF; the only difference is that distance values are truncated after they exceed a threshold. Their method was one of the first to be able to handle several registered range scans. Their work showed how well a TSDF can deal with several noisy scans by naturally integrating out the noise. They obtained very good results but were not even close to real-time. A speed up in processing time was achieved by Pulli et al. in 1997 [60] by utilizing octrees. They obtained good results and their method was used by Surmann et al. [61] in a well-known robotic mapping work. Another major advance came in 2001 from Zhao et al [62]. They used Partial Differential Equation (PDE) methods to obtain a final reconstruction that was of better quality than prior methods. In 2001 Carr et al. [63] created a volumetric method based on the radial basis function (RBF). Their method was able to successfully deal with holes and generate water tight models. A water tight model is useful for single object reconstruction. However, it is not desired for mapping large environments. One of the next major advances was published in 2003 by Ohtake et al. [46]. In this work they created a method that was faster than the work of Carr et al. [63] by implementing a hierarchical approach with compactly supported basis functions. At the time, their work was considered to be the state of the art for calculating an implicit function of a noisy point set and was used by Newcombe et al. [45]. Volume-based methods have been able to create high quality representations and work well for single objects and small environments. These methods must spatially divide the environmental volume and therefore have a high memory requirement.

2.2.2 Surface-based

One of the first interesting and adaptive surface-based methods was published by Terzopoulos and Vasilescu in 1991 [64] and dealt with 2.5D data such as intensity and range images. The goal of their work was to create an adaptive mesh of an input image. The mesh was initialized as a 2D sheet of mesh elements with virtual springs along each edge. The stiffness of each virtual spring would then adjust based on the image information at its locations. The mesh was able to adapt to be more dense in regions of higher intensity. In 1992 Terzopoulos and Vasilescu extended their methodology to 3D data [65]. In this work they used the distance between the mesh and the data to drive the vertices to be near the surface. In this early work they needed to initialize the mesh and control the subdivision of mesh elements to obtain a suitable resolution. In 1993 Hoppe et al. [66] published a method that used an energy minimization framework. Their method minimized an energy function that modeled the competing desires of conciseness of representation and fidelity to the data. They successfully used their method for both surface reconstruction and mesh simplification. One of the next advances in physical based adaptation of meshes came in 1993 from Huang and Goldof [67]. In this work they were able to adjust the size of the mesh elements to obtain a dense resolution in areas of high frequency information using a physical based model. In addition, it was one of the first works to represent an object undergoing deformation. Their method was able to perform tracking on simple simulation examples. Another advancement came in 1994 Rutishauser et al. [68] with a method specifically designed for incremental data. Their methodology worked with a sequential input set of range data and used a probabilistic framework to adjust the vertices of a mesh to the expected value given the prior observations. Their methodology also modeled the noise of the sensor with a sensor model. In 1994 Delingette [69] developed a methodology to generate a simplex mesh model of structured and unstructured 3D datasets. Elastic behavior of the mesh surface

was modeled by local stabilizing functionals. Also, they implemented an iterative refinement process to refine the mesh in areas of high frequency information. One of the next steps was published by Turk and Levoy in 1994 [70]. Their method allowed overlapping meshes to be "zippered" into a single mesh surface. This ability is especially important for methods that generate a mesh for each depth image of the sensor and then need to combine all registered meshes into a single mesh. Their method is computationally expensive due to distance calculations. An interesting work came in 1995 from Chen and Medioni [71]. They devised an adaptive mesh methodology based on the inflation of a balloon. A mesh sphere was first initialized within the registered range measurements of the object. Virtual inflation forces were then used to expand the balloon until the mesh surface was a minimal distance from the range data. This method was limited to objects that are water tight. A major advancement came in 1999 from Bernardini et al., [72] in a method named the ball-pivoting algorithm. Their method is a good example of an advancing front method. These types of algorithms start with a seed mesh element and advance the boundary by adding new mesh elements in the immediate area of the boundary which is supported by measurements. Advancing front algorithms differ in how it is decided to add new mesh elements. In the work of Bernardini et al., a virtual sphere of a user defined radius is rolled along the boundary of the mesh and new elements are added if the ball touches another measurement. Their methodology became popular because of its simplicity. One major disadvantage was that the generated mesh was a fixed topology. Another advancing front method came in 2001 from Gopi et al. [73, 54]. Here, they sampled the input dataset to obtain a new dataset with a lower density of points in areas of lower frequency information. This effectively gave their method an adaptive topology. Next, a local neighborhood was computed at each data point and projected to a plane tangent to the surface. The triangulation is then computed on this local tangent plane. They obtained impressive results on datasets of varying sample density and curvature. An interesting work was published in 2003 by Ivris-

simtzis et al. [74]. Here they used a neural network model to adapt a mesh model to the data. They claimed that their method is computationally independent of the size of the input dataset because the dataset is only sampled by the method. There obtained good results. In 2004 Alexa et al. published a very interesting work to generate point set surfaces from an input dataset [75]. They use moving least squares (MLS) to locally approximate the surface with polynomials. The original dataset is then no longer used. Instead, they develop tools to sample the approximated surface to any resolution desired so that the end result is another point set of user specified resolution lying closer to the surface than the input dataset. One drawback is they had to develop their own methodology to render a point set. In 2005 Scheidegger et al. used the work of Alexa et al. to develop an advancing front methodology to generate concise meshes of high accuracy. Their main contribution was to augment an advancing front algorithm with global information so that the triangle size could adapt gracefully to any change. They obtained very impressive results. Most methodologies in Surface Reconstruction had been solely concerned with object or small environment recreation and have computational or memory requirements that do not work well with large environments. One of the first successful methods intended for large environments was published in 2009 by Marton et al. [76]. Their methodology was an advancing front algorithm that worked on a point set sampled from the MLS surface of the original point set. They were able to obtain impressive and near real-time results on datasets of large environments. They also developed a method to deal with revisited parts of the scene by determining the overlapping area and reconstructing only the updated part of the surface mesh. To support dynamic scenes they developed mechanisms to decouple and reconstruct the mesh quickly. They only discussed these mechanisms in theory and had no results of how these mechanisms work.

2.3 Summary

The fields of Robotics and Computer Vision have developed many exciting methodologies to construct representations from a noisy input dataset. However, there is still work to be done to obtain the ideal reconstruction method. A mesh is clearly a desirable type of representation. An ideal method both generates and maintains a mesh representation efficiently. Also, many existing methods do not leverage the inherent structural information contained within the depth image. There are imaging processing techniques that could be used to answer some of the remaining problems in surface reconstruction, such as the need for adaptive topology and the need to decide how each measurement should be used to update the existing mesh. Henry et al. [48] have already investigated using the difference between the expected and actual measurements to guide the decision of how to use each measurement. However, their work was intended for surfels and needs to be extended to meshes. A method to generate a representation is needed which is computationally and memory efficient and can adapt the representation to new information.

Chapter 3

Approach

3.1 Algorithmic Design

The algorithmic structure of MABDI can be seen in the system diagram shown in Fig. 3.1. Table 3.1 gives a description of the main variables.

Table 3.1: Description of the main variables

Variable Name	**Description**
D	Depth image from RGB-D sensor
P	Pose of the sensor
D_n	Parts of D that are *novel*
S	Novel surface generated from D_n
M	Global mesh

The system diagram of Fig. 3.1 is a more detailed version of the diagram seen in Fig. 1.2b. The "Identify Novel Data" component, shown in Fig. 1.2b, corresponds with the Classification component, shown in blue. This Classification component is MABDI's contribution to the state-of-art in mesh based mapping algorithms, and is what gives MABDI the ability to make decisions about the incoming data. The

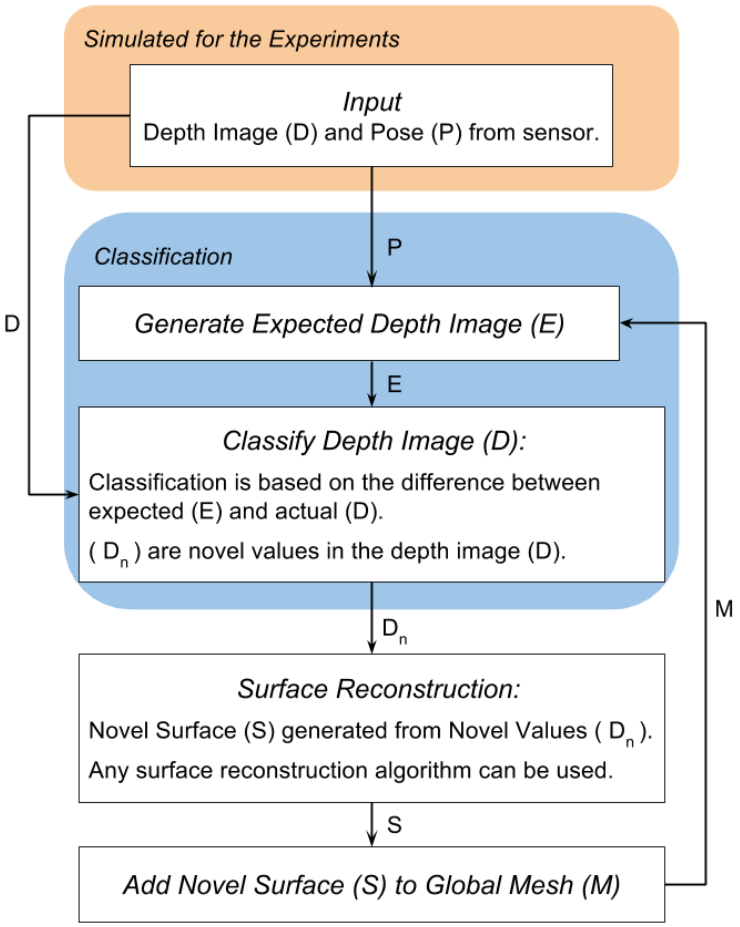


Figure 3.1: MABDI system diagram

Classification component consists of two parts:

1. *Generate Expected Depth Image E* - Here we take the global mesh M, render it using computer graphics, and use the depth buffer of the render window to create a depth image E of what we expect to see from our sensor. This method requires the current pose P of the actual sensor (simulated for our experiments).

2. *Classify Depth Image D* - Here we classify the actual depth image D (simulated

for our experiments) by first taking the absolute difference between E and D and thresholding, as shown in the equation below. If the differences are small, those points are thrown away and if the differences are large, those points are kept as D_n. The idea behind this is, if the difference is large, the measurements are coming from a part of the environment that has not been seen before, i.e. novel. We found $threshold$=0.01 worked well in our simulations. The implication of assuming all large differences signifies novel data is that this version of MABDI cannot handle object removal. It is worth noting that MABDI can be extended to handle object removal by using the sign of the difference between E and D instead of the absolute value.

$$D_n = |D - E| > threshold \tag{3.1}$$

The system diagram in Fig. 3.1 also shows the Input and the Surface Reconstruction components. The Input component has been simulated for our experiments. More details of this simulation will be covered in Chapter 4. The Surface Reconstruction component of the MABDI algorithm can be implemented with any viable surface reconstruction method. Our implementation utilizes the structural information contained within the depth image. We will discuss this in more detail in the next section.

3.2 Implementation

3.2.1 Surface Reconstruction

The Surface Reconstruction component, as shown in Fig. 3.1, is responsible for creating a surface S from the novel points D_n. The surface S is a mesh data structure that consists of a list of vertices and elements. Vertices are points and elements define

connections between vertices. Our method outputs a triangle mesh, and so elements define the connection between three vertices. D_n is a subset of D and is a list of pixel locations. For this discussion, it will also be useful to define D_k as the set of pixels in D that are not pixels of D_n, shown in the equation below. D_{known} is labeled with "*known*" because it represents data from the not novel or "known" parts of the environment. In the equation below "\" is the set difference operator.

$$D_{known} = D \setminus D_n \tag{3.2}$$

Our surface reconstruction method first defines S using all pixels from D. We define the topology of the elements on the depth image. We can do this because a depth image is not a set of unorganized points, but has inherent structural information. This characteristic of the depth image allows us to define a topology on the 2D depth image that is preserved when projected to 3D coordinates. The topology we define can be visualized in Fig. 3.2. Elements of the mesh are shown in light blue and pixels from D are shown as blue dots. Next we will identify elements to remove from S.

In order to remove elements defined by points that lie on completely different surfaces, we use an imaging technique in the form of a convolution filter. A two dimensional, differencing convolution filter is passed over D. This filter has a magnified response at points where the difference between neighboring pixels is large. Remembering pixel values signify depth, it is assumed pixels with large differences between themselves and their neighbor lie on different surfaces and therefore lie on the "boundary" of the real surface. A large difference is defined by thresholding on the result of the convolution. We found $threshold$=0.01 worked well in our simulations. (The threshold value is unitless because the depth image is defined by the z-component of the view coordinates, which are normalized between 0 and 1.) Pixels identified through this thresholding are marked as $D_{boundary}$ and are defined by the

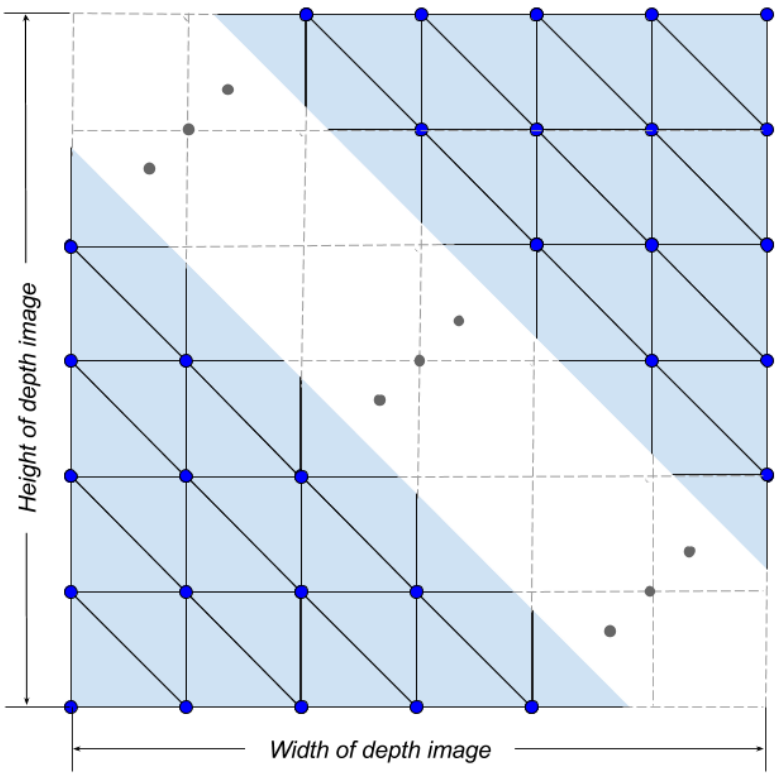


Figure 3.2: Topology defined on the depth image (not all elements are shown)

equation below where K signifies the kernel of the differencing convolution filter.

$$K = \begin{bmatrix} 2 & -1 \\ -1 & 0 \end{bmatrix} \tag{3.3}$$

$$D_{boundary} = (D * K) > threshold \tag{3.4}$$

Elements are removed from the S if they touch pixels from the sets:

- D_{known} - Pixels from the known parts of the environment.
- $D_{boundary}$ - Pixels that lie on the boundary of the actual surface.

- $D_{invalid}$ - Pixels that are invalid measurements. The RGB-D sensor naturally has pixels that are invalid, for example, those that are out of range.

Let us combine the sets defined above into one set $D_{throwaway}$:

$$D_{throwaway} = D_{known} \cup D_{boundary} \cup D_{invalid} \tag{3.5}$$

Our method removes elements that contain pixels from the set $D_{throwaway}$. This can be seen in Fig. 3.3. Red dots signify pixels from $D_{throwaway}$ and elements that contain these pixels are removed from S. In the final step, all pixels are projected into 3D coordinates using the transformation matrix of the sensor. These coordinates are the vertices of S.

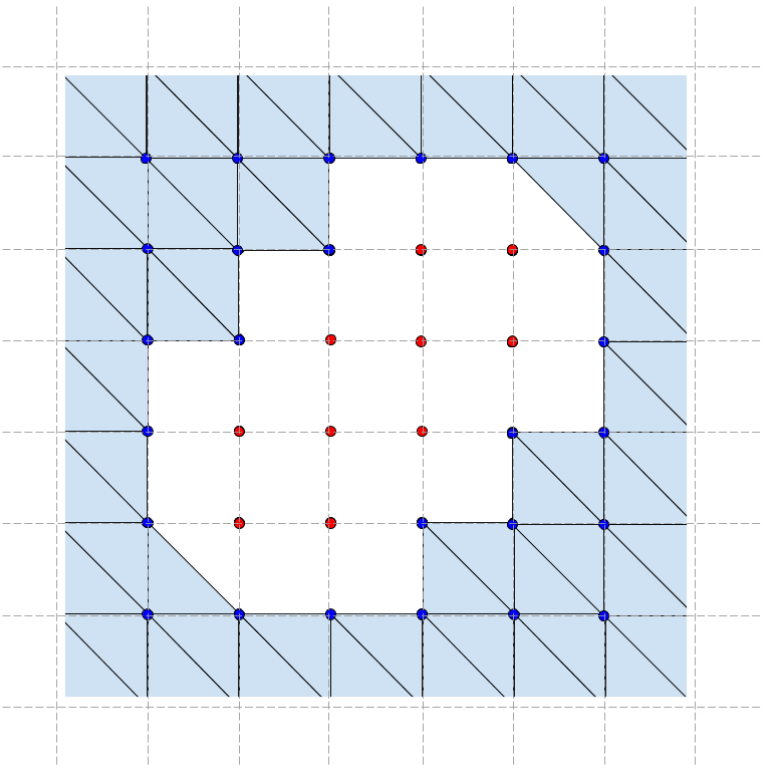


Figure 3.3: Removal of elements

Our surface reconstruction method was chosen for its ability to be implemented simply and run quickly. One consequence of our method is that the resulting surface S can have a large number of elements. For example, if no points are contained in the set $D_{throwaway}$ (this can happen on the first frame), S will contain over 600,000 elements. We can see this by looking at Fig. 3.2, assuming a depth image of size 640×480, and considering the equation below.

$$612,162 = ((640 - 1) \times 2) \times (480 - 1) \tag{3.6}$$

Many surface reconstruction methods have been developed to create a surface more intelligently than our surface reconstruction method, as discussed in Chapter 2. For example, the advancing front method developed by Marton et al. [76] is capable of creating surfaces with fewer elements than our method by utilizing a robust resampling method. A capability of the MABDI algorithm is that the method developed by Marton et al. can be used in place of our surface reconstruction method. This characteristic of MABDI is advantageous because MABDI does not depend on the choice of surface reconstruction method and the method can be chosen as the state-of-the-art changes or to suit a particular application. Also, due to our implementation's modular software design, the entire code base would not need to be changed in order to accomplish this. We will discuss the software design in the next section.

3.2.2 Software Design

From a software perspective, the major difficulty of implementing the MABDI algorithm was found to be creating both the simulated depth image D and the expected depth image E. In addition, managing the complexity of the data pipeline needed to run the algorithm and the simulation of the sensor proved to be difficult. Thank-

fully, Kitware, which is a leading edge developer of open-source software, created the Visualization Toolkit (VTK) [77, 78]. At the time of this writing the VTK Github repository has over 60,000 commits and is contributed to by supporters such as Sandia National Labs [79].

VTK is suitable for the implementation of MABDI for many reasons. Perhaps the most important is the concept of a vtkAlgorithm (often called a Filter). This allows a programmer to create a custom and modular processing pipeline by defining classes that inherit vtkAlgorithm and then defining the connections between these classes. For example, you could have a pipeline that reads an image from a source (component 1), performs edge detection (component 2), and then renders the image (component 3).

Using the concept of VTK filters, the individual elements of MABDI can be succinctly defined in individual classes. With that in mind, we can see in Fig. 3.4 the layout used in our implementation of MABDI. vtkImageData and vtkPolyData are VTK types used to represent an image and mesh respectively. The elements shown in blue in Fig. 3.4 are the core components of the MABDI algorithm and are implemented as custom VTK filters. Their source code is included in Appendix A. Here we will discuss all components in detail:

- *Source* - Classes with the prefix Source define the environment that is used for the simulation and provide a mesh in the form of a vtkPolyData.

- *FilterDepthImage* - Render the incoming vtkPolyData in a window and output the depth buffer from the window as a vtkImageData. The output additionally has pose information of the sensor.

- *FilterClassifier* - Implements the true innovation of MABDI, i.e., takes the difference between the two incoming depth images (vtkImageData) and outputs

a new depth image where the data that is not novel is marked to be thrown away.

- *FilterDepthImageToSurface* - Performs surface reconstruction on the novel points. For more detail see Section 3.2.1. The surface is output as a vtkPolyData.

- *FilterWorldMesh* - Here we simply append the incoming novel surface to a growing global mesh that is also output as a vtkPolyData.

MABDI is implemented in Python and uses VTK. Our implementation is distributed under the BSD license and is available on Github at the address below:

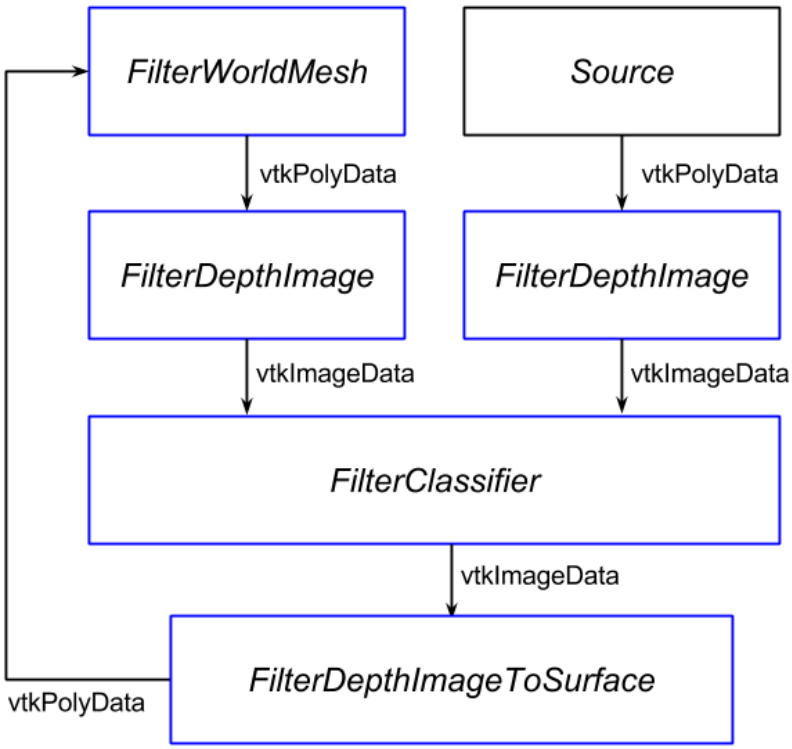


Figure 3.4: MABDI software diagram

$$https://github.com/lucasplus/MABDI$$

At the time of this writing, it consists of over 1,400 lines. The code that implements the MABDI algorithm itself is around 750 lines.

Chapter 4

Experimental Setup

MABDI was developed and tested in a completely simulated environment for several reasons. First, all results are repeatable. Having repeatable results is important for algorithm development because the effects of code changes in the implementation can be directly correlated to changes in the output. This facilitates isolation and identification of trouble spots in the code. In addition, it is possible to test the algorithm in the most ideal environment before adding complexity. The ability to ramp up the difficulty of the environment in which MABDI is performing is important for making informed design decisions. Finally, by performing the analysis in simulation we can quickly see how the map produced by MABDI compares with the simulated environment. This comparison is an important tool for development.

In this chapter we will give an overview of the simulation environment, discuss how noise was generated to mimic the input of a real RGB-D sensor, and look at the parameters chosen for the experimental runs.

4.1 Simulation Overview

For the experiments, we simulate a sensor moving in a fixed environment along a defined path. The simulation consists of two main coordinate systems. A coordinate system fixed to the environment called the global coordinate system and one attached the origin of the sensor's viewing frustum. Fig. 4.1 shows the two coordinate systems from two different vantage points. In the figure red, green, and blue arrows represent the x, y, and z axis respectively.

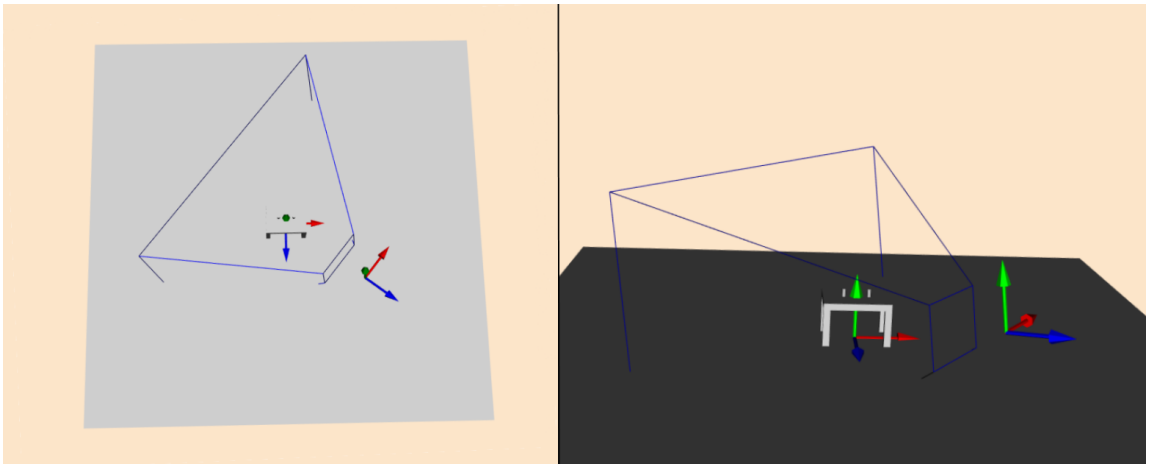


Figure 4.1: Overview of the simulation. Left: Top view. Right: Third person view.

4.2 Simulating a RGB-D Sensor

4.2.1 Rendering Pipeline

In order to simulate the depth output of a RGB-D sensor, the environment is rendered from the sensor's point of view. The rendering process produces a depth image and this image is used as the simulated output of the sensor. Rendering is performed by the Open Graphics Library (OpenGL). OpenGL creates a rendering pipeline that

consists of a series of transformations to project 3D global coordinates to 2D pixel coordinates. A diagram of the rendering pipeline is shown in Fig. 4.2. T_{pcm} represents the pinhole camera model and transforms geometry in the sensor's coordinate system to homogenous coordinates. The z-component of the homogenous coordinates is what defines the depth image. Note, the use of a pinhole camera model for simulating RGB-D output has been validated in the localization work of Fallon [80] and the intrinsic camera parameters of the model were chosen to replicate the Kinect sensor [81].

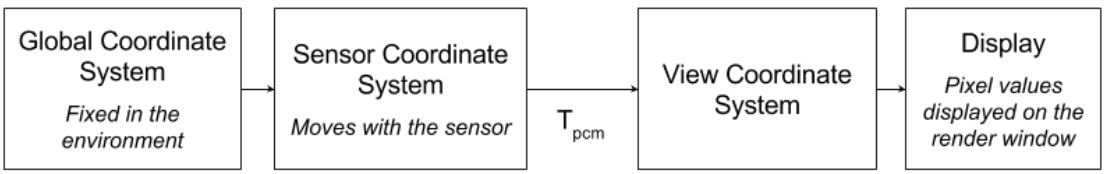


Figure 4.2: Render pipeline: projects 3D global coordinates to 2D pixel coordinates.

The pinhole camera transformation, T_{pcm}, creates a non-linear relationship between values in the depth image and their corresponding location in the sensor's coordinate system. This relationship is visualized in Fig. 4.3.

4.2.2 Adding Noise to the Depth Image

To simulate a realistic RGB-D sensor, we add noise to the depth image D with the goal of approximating RGB-D error models from the literature. Researchers have created error models to describe the standard deviation of measurement error found in various RGB-D sensors. For this work, we seek to match the well-known error model of Khoshelham [82] that is based on the original Kinect. The error model is defined in Equation 4.1. The equation expresses the standard deviation of error in the z-component of a point in the sensor's coordinate system σ_z (cm) as a function of the value of the z-component Z (m). Measurements further away from the sensor

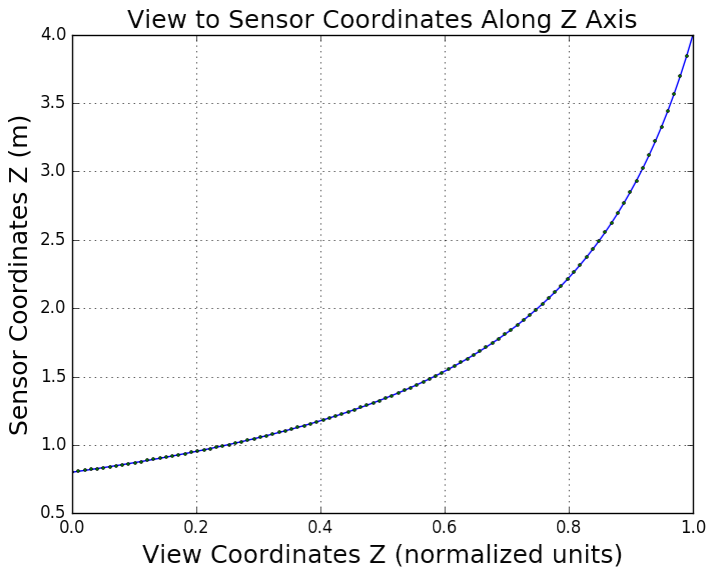


Figure 4.3: View coordinates to the sensor's coordinates.

have a larger standard deviation of error. The error model is graphed as the red line in Fig. 4.4.

$$\sigma_z = 1.425\text{e}{-5} \times Z^2 \tag{4.1}$$

To approximate a real RGB-D sensor that matches Khoshelham's noise model, noise is added to the depth image D by sampling a normal distribution and adding the value to each pixel. as defined in the equation below. The mean of the normal distribution in Equation 4.2, σ=0.002, was experimentally found to provide a conservative approximation of Khoshelham's error model.

$$D_{noisy}(i,j) = D(i,j) + \mathcal{N}(\mu{=}0, \sigma{=}0.002) \tag{4.2}$$

In order to compare the magnitude of the standard of deviation of error used in our experiments with that of Khoshelham's error model, we graph them on the

same plot (Fig. 4.4). Each line shows how the measurement's standard deviation of error changes as the point moves along the z axis in the sensor's coordinate system. The standard deviation of error simulated in our experiments is larger than that defined by Khoshelham's model for points within the sensor's range. Therefore, our experiments are a conservative estimate of the error found in real world RBG-D sensors.

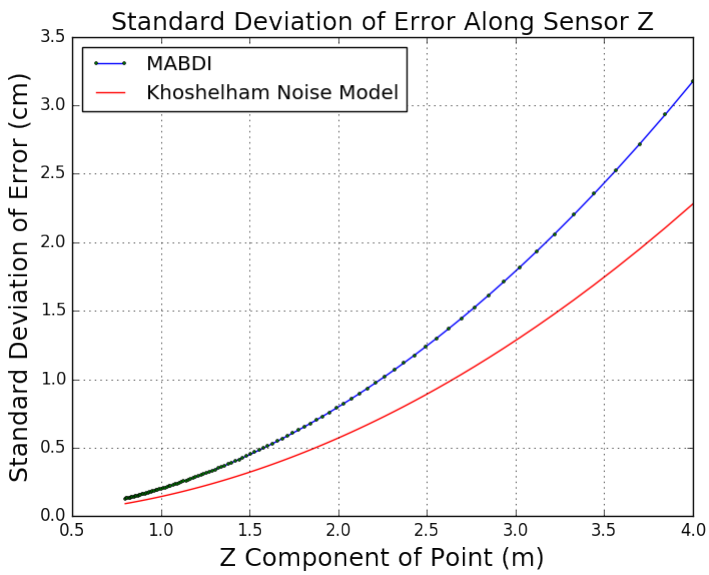


Figure 4.4: Comparison of standard deviation of the error used in the MABDI simulation and the error model from Khoshelham.

4.3 Sensor Path

All experimental runs define a helical path for the sensor to follow during the simulation. The path is shown in Figure 4.5. The blue line indicates the path and the pink points indicate where the sensor stops along the path. The path circles the objects in the environment twice. A helical path was chosen because it returns to a part of the environment that has already been mapped and is thus "known" to the

algorithm. Also, because the path is a helix and not just a circle, the sensor views the environment from a slightly different position on each pass.

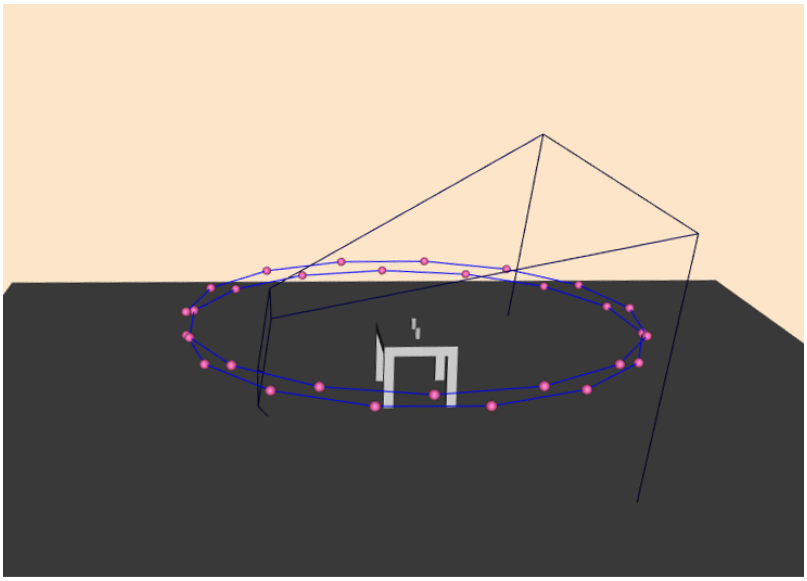


Figure 4.5: View of the sensor path. The blue line indicates the sensor path and the pink points indicate where the sensor stops along the path.

4.4 Simulation Parameters

The simulation was designed to be highly configurable and is implemented by a class named MabdiSimulate. This class is responsible for connecting all the components expressed in Fig. 3.4 of Chapter 3. MabdiSimulate is initialized with parameters that control all aspects of the simulation. Parameters of a particular importance are discussed in more detail here:

- Environment - This parameter specifies the environment used to generate the simulated depth images. *Table* is an environment consisting of a table and two cups placed on the table. The table is 1 meter tall. *Bunnies* is an environment

consisting of three bunnies that are around 1.5 meters tall. These bunnies are created using the Stanford Bunny [70], a well known data set in computer graphics.

- Noise - If true, adds noise to the depth image of the simulated sensor.
- Dynamic - If true, adds an object during the simulation. In the case of this analysis, a third bunny is added half-way through the simulation.
- Iterations - The number of times MABDI will run. This number is equal to the number of stops the sensor makes along the path because every time the sensor stops MABDI is run to update the global mesh. Figure 4.5 shows sensor stops along the sensor path.

We will be exploring three experimental runs to demonstrate the ability of the MABDI implementation to generate valid results. Additionally, the experimental runs will be able to show the capabilities of the MABDI algorithm such as handling object addition in the environment.

Table 4.1: Description of the experimental runs.

	Environment	Noise	Dynamic	Iterations
Run 1	Table	False	False	30
Run 2	Bunnies	True	False	50
Run 3	Bunnies	True	True	50

Chapter 5

Results

For each experimental run, a dashboard view was created that can be shown for each iteration of the simulation. The dashboard view combines several different views of information useful for understanding the inner workings of the MABDI algorithm. As an example, Figure 5.1 shows the dashboard view for the first experimental run. For these experiments, all dashboard views follow the same pattern as described below:

- (a) - Shows the global mesh M from a third-person point of view and in the context of the simulated environment. The multi-colored mesh is M. The mesh is multi-colored in order to show the passage of time. For example, in Run1, The mesh is colored yellow, light green, and dark green for iterations 1, 2, and 3 respectively. Additional items in the view show elements of the simulated environment: the wire frame corresponds to the viewing frustum of the sensor, the light blue helical line is the path of the sensor, and the translucent gray mesh is the simulated environment.
- (b) - Same as (a) except it shows the novel surface S instead of the global mesh M.

- (c) - Plot showing the number of elements in the global mesh M after this iteration.
- (d & e) - Actual D and expected E depth image respectively.
- (f) - The classified depth image. Points that will be used to generate the novel surface S are shown in black. Points to be thrown away are shown in white.

The dashboard views are an excellent way to visualize important aspects of MABDI. In the next section, Section 5.1, we will utilize key dashboard views to look at the behavior and performance of MABDI at one particular iteration of each experimental run. In section 5.2 we will analyze the quality and progression of the resultant global mesh from each experiment.

5.1 MABDI Performance During Experiments

5.1.1 Experiment 1

Figure 5.1 shows the dashboard view of the first experiment during the third iteration. Note that 5.1(a) shows M after the third iteration. As stated before, M is multi-colored in order to show the passage of time. The mesh is colored yellow, light green, and dark green for iterations 1, 2, and 3 respectively. *During iteration 3, M is composed of only the yellow and light green parts.*

Examining Figure 5.1 demonstrates how the novel surface S is appended to the global mesh M after each iteration of MABDI. Let's use the figure to follow the process. It will be useful to refer to Figure 3.1 for this section.

1. Input - 5.1(d) shows the depth image D generated from the simulated sensor. 5.1(a) shows us two important aspects to consider about D. First, the pose P

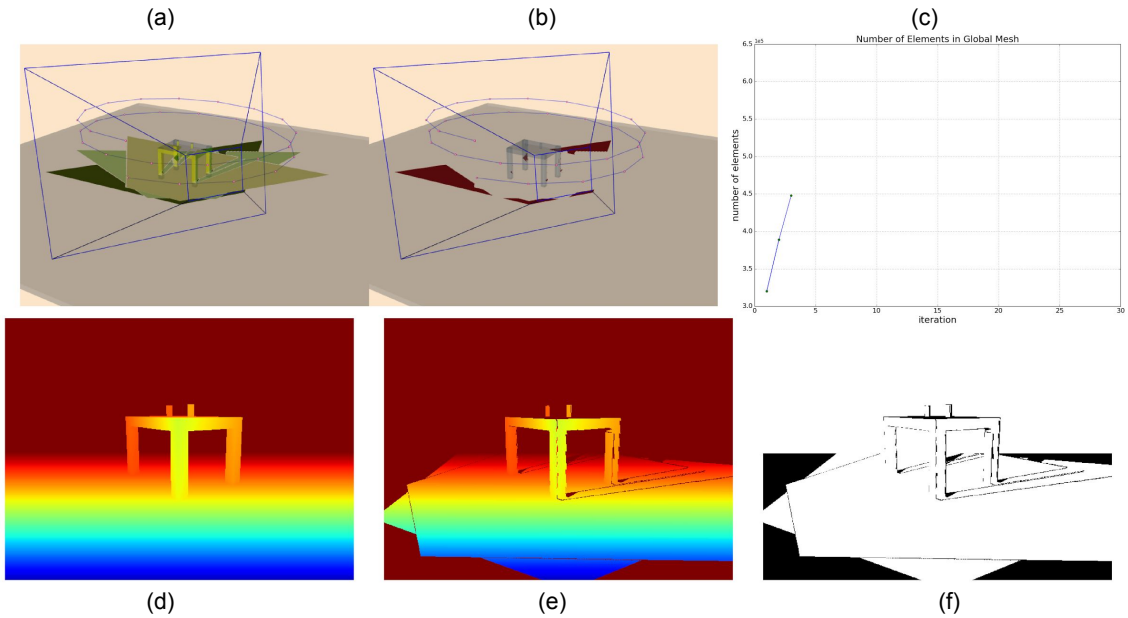


Figure 5.1: Dashboard view of the first experimental run.

of the sensor is shown by looking at the sensor's view frustum, indicated by the blue wireframe. Second, the only environmental information used to generate the depth image is shown in light gray.

2. Generate Expected Depth Image (E) - 5.1(e) shows the expected depth image E. 5.1(a) also shows us two important aspects to consider about E. First, the same pose P is used to create both D and E (as indicated by the blue wire frame). Second, the only environmental information used to create E is the yellow and light green parts of M because that is the only information M contains *during* iteration 3.

3. Classify Depth Image (D) - 5.1(f) visualizes the classification process. More specifically, it shows the points as expressed in Equation 3.5 in white ($D_{throwaway}$). 5.1(f) is important for understanding how MABDI works because it clearly shows which points will be thrown away (white) and which

points will be kept for generating the novel surface S (black).

4. Surface Reconstruction - 5.1(b) shows the novel surface S in the context of the simulated environment. S is constructed using all the points colored black in 5.1(f).

5. Add Novel Surface (S) to Global Mesh(M) - 5.1(a) shows the novel surface S appended to the global mesh M in dark green.

5.1.2 Experiment 2

The second experiment gives us a clear example of how the classification process is able to identify points from the depth image D that correspond to parts of the environment that have not been seen before. In this example the global mesh M has a partial representation of the objects in the environment and when the sensor is moved to the next pose P, the new perspective reveals a portion of the object that has not been seen before. This *novel portion* of the environment, which we will be referring to, is shown by the red ellipse in Figure 5.2.

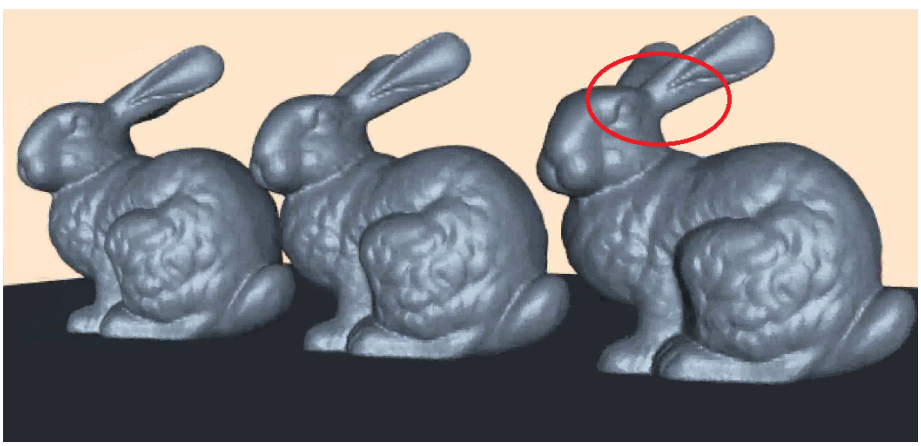


Figure 5.2: *Novel portion* of the environment that we will be referring to in this section.

Figure 5.3 shows the dashboard view of the second experiment during the second iteration. Using the dashboard view, we can follow how MABDI handles the novel portion of the object step-by-step:

1. 5.3(a) shows the global mesh M. The yellow portion of the mesh constitutes the entirety of M after the first iteration. We can see the novel portion of the environment was not represented in M after the first iteration due to occlusion.

2. 5.3(d) shows the depth image D from the new sensor pose P. We can see the novel portion can be seen by the sensor on this iteration.

3. 5.3(e) shows the expected depth image E. During the second iteration M consists of only the yellow portion shown in 5.3(a) consequently, E does not show any points in the area corresponding to the novel portion of the environment.

4. 5.3(f) shows the classification process successfully identifying points in D that correspond to the novel portion as indeed novel. In the figure the points are highlighted by a red circle.

5. 5.3(b) shows the novel surface S now represents the novel portion of the environment.

6. Finally, the orange mesh in 5.3(a) shows the novel portion of the environment is now represented by the global mesh M.

5.1.3 Experiment 3

Experiment three shows how MABDI reacts to object addition. Figure 5.4 shows the dashboard view of the third experiment during the twenty-sixth iteration. At this iteration the middle bunny is suddenly added to the simulated environment. We can use the dashboard view to see the behavior of MABDI to this new object:

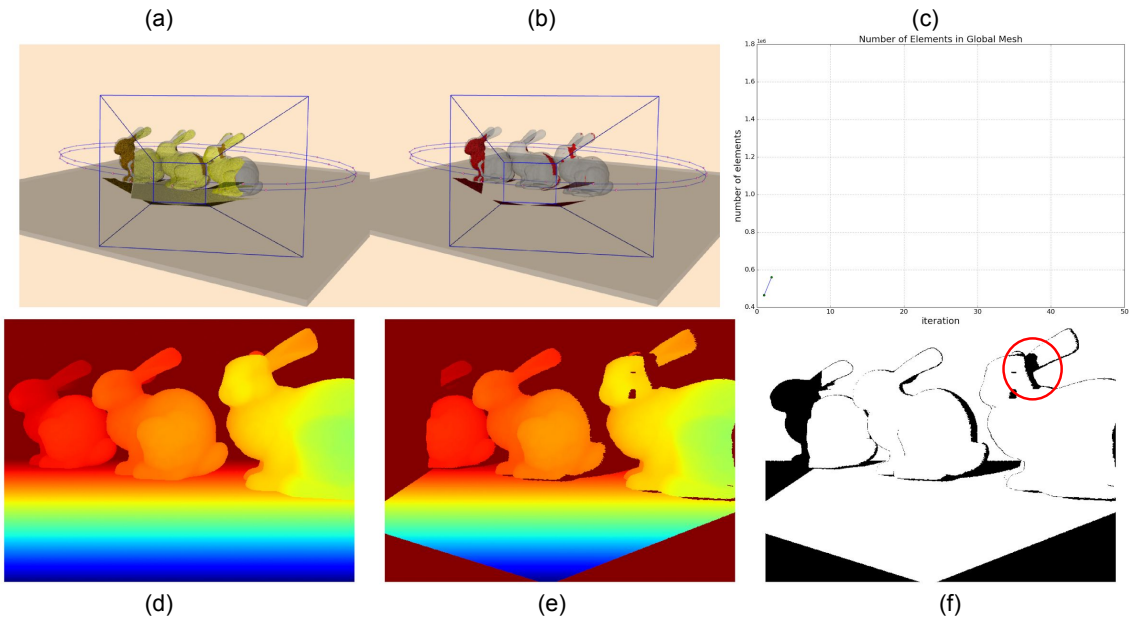


Figure 5.3: Dashboard view of the second experimental run.

1. In 5.4(d) we see the depth image D shows the new bunny.

2. In 5.4(e) the expected depth image E does not show the new bunny because M has no representation of the new bunny.

3. 5.4(f) shows the classification process successfully identified the points corresponding to the new bunny as novel.

4. The novel points are used to generate the novel surface S and then S is appended to M, shown in 5.4(a & b).

5. The addition of the new object resulted in a S with a large number of elements for this particular iteration. 5.4(f) plots the resulting jump in the number of elements contained with M.

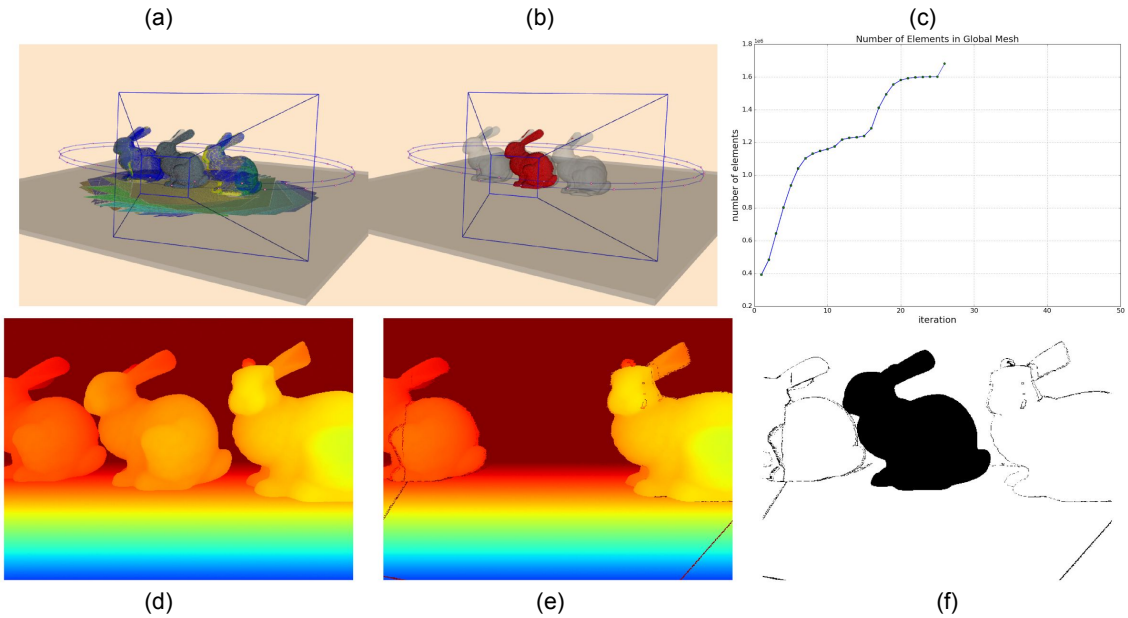


Figure 5.4: Dashboard view of the third experimental run.

5.2 Global Mesh Results

5.2.1 Mesh Quality

Figure 5.5 shows the resultant global mesh from experiment 3. In this section we will use this figure to make observations about the quality of the global mesh for all three experiments.

There are gaps in the mesh that occur typically along the boundaries of where the novel surface S is appended to the global mesh M. This behavior is common for Surface Reconstruction methods as those discussed in Section 2.2. Algorithms exist for merging these gaps as a post processing step such as Turk's Zippered Polygon Meshes [70]. The aforementioned methods are typical for single object reconstruction. Traditional mesh-based environmental mapping algorithms simply append overlapping layers of mesh resulting in no gaps but a heavily redundant representation with a

high memory cost.

The mesh is noisy. This noisiness is due to the simplicity of our implementation's surface reconstruction method as discussed in Section 3.2.1. Our method simply connects neighboring points in the point cloud without additional steps such as Laplacian smoothing [83]. Our reconstruction method was sufficient for demonstrating the usefulness of the MABDI algorithm, but results in a mesh with the same magnitude of noise as the sensor's simulated noise.

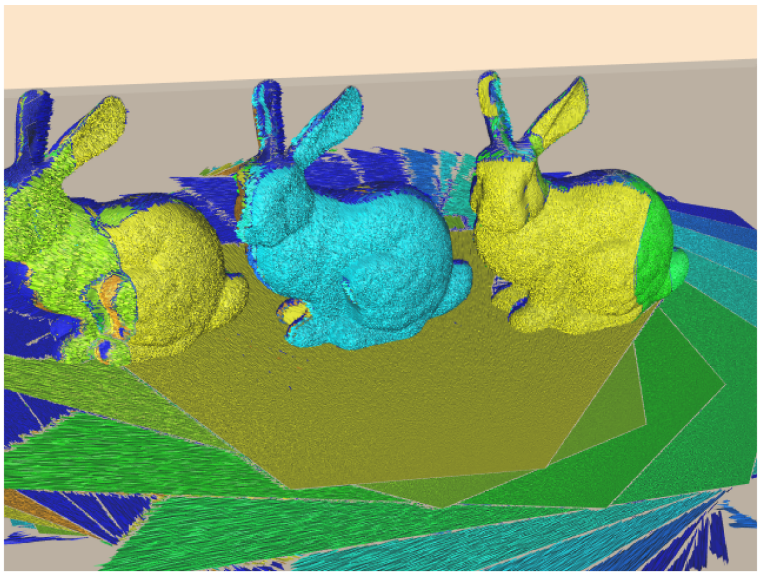


Figure 5.5: Global mesh at the end of experiment 3.

5.2.2 Mesh Progression

To appreciate the true benefit of the MABDI algorithm it is helpful to look at how the number of elements in the global mesh M progress over time. In this section we will analyze plots showing how the number of elements in M change during the experiments. Note, the dashboard views also showed this plot. For example, the plot of Figure 5.6 is the same as Figure 5.1(c), but Figure 5.6 shows the plot at the

completion of the experiment.

Figure 5.6 shows the resultant mesh and mesh progression for the first experiment. The plot highlights the major difference between MABDI and traditional mesh-based environmental mapping methods. Traditional methods would have a plot similar to that indicated by the red arrow on the graph because these methods have no ability to identify or remove redundant mesh elements. Due to MABDI's algorithmic design, MABDI has the intrinsic ability to identify points in the depth image corresponding to parts of the environment that are already known by the global mesh M. MABDI then simply does not use those points for surface reconstruction and consequently does not create redundant mesh elements. For this reason, the number of elements in M levels off as the environment becomes more known.

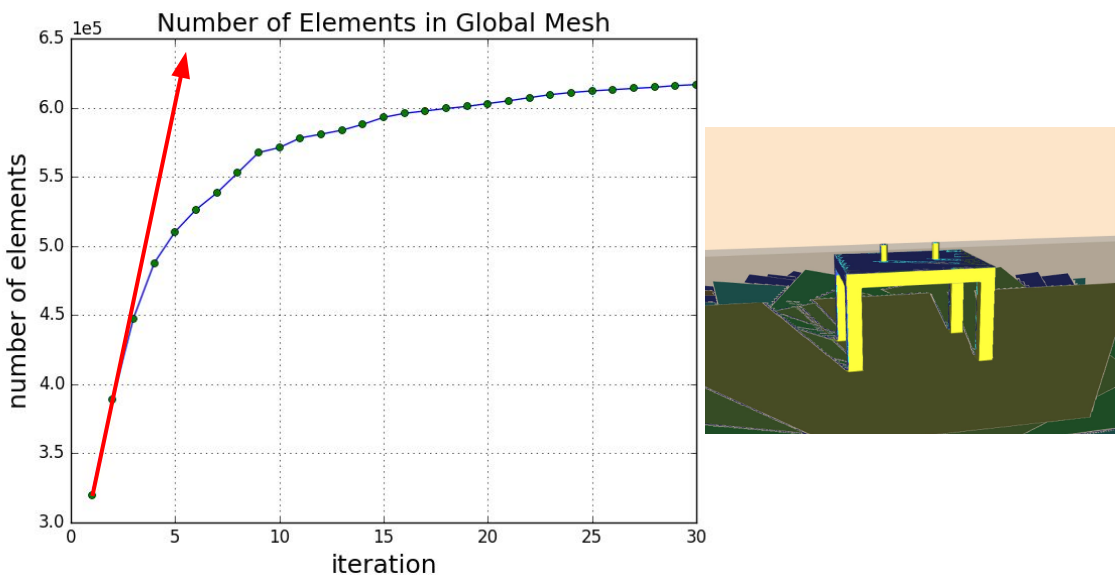


Figure 5.6: Experiment 1 global mesh results.

Figure 5.7 shows us the resultant mesh after the second experiment. Here we can see that MABDI is reactive to the environment. In the preceding experiment, the environment was symmetrical. In this experiment, the environment is not symmet-

rical and we can see the effects by looking at the progression of the global mesh M. First let us note that the sensor circles the objects twice during the experiment and in total travels 720° during the 50 iterations. We notice when the sensor gets to 90° (around iteration 7) the number of elements begins to level off and then increases again as the sensor travel to 270° (around iteration 19). This behavior occurs because the information rich perspectives of the environment occur at 0° and 180°. There is less for the sensor to look at when viewing the environment from the sides. In this way, MABDI is reactive as the sensor moves to parts of the environment that are rich in information. Consequently, the mesh grows rapidly based on the needs of the environment.

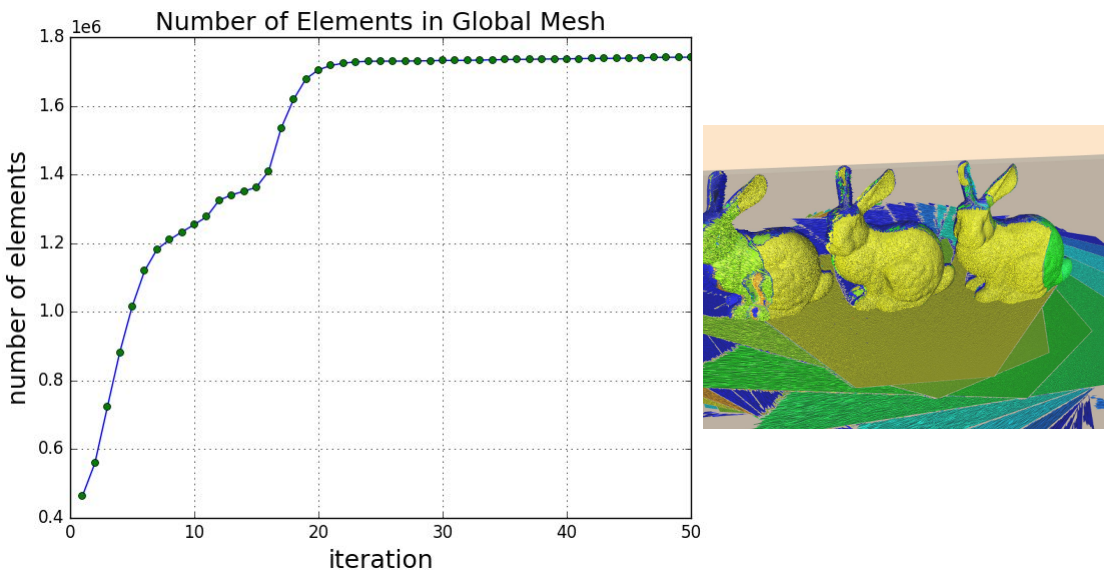


Figure 5.7: Experiment 2 global mesh results.

Figure 5.8 shows us the resultant mesh after the third experiment. In this experiment the middle bunny was added during the twenty-sixth iteration. This object addition had two effects on the global mesh. First, it created a sudden jump in the plot as highlighted by the red circle. Second, the middle bunny is colored blue in the

resultant mesh, signifying that it was added to M during a different iteration than the bunnies on the left and the right. Both of these effects indicate that MABDI was able to successfully identify the new bunny as novel and incorporate the bunny in to the global mesh within one iteration.

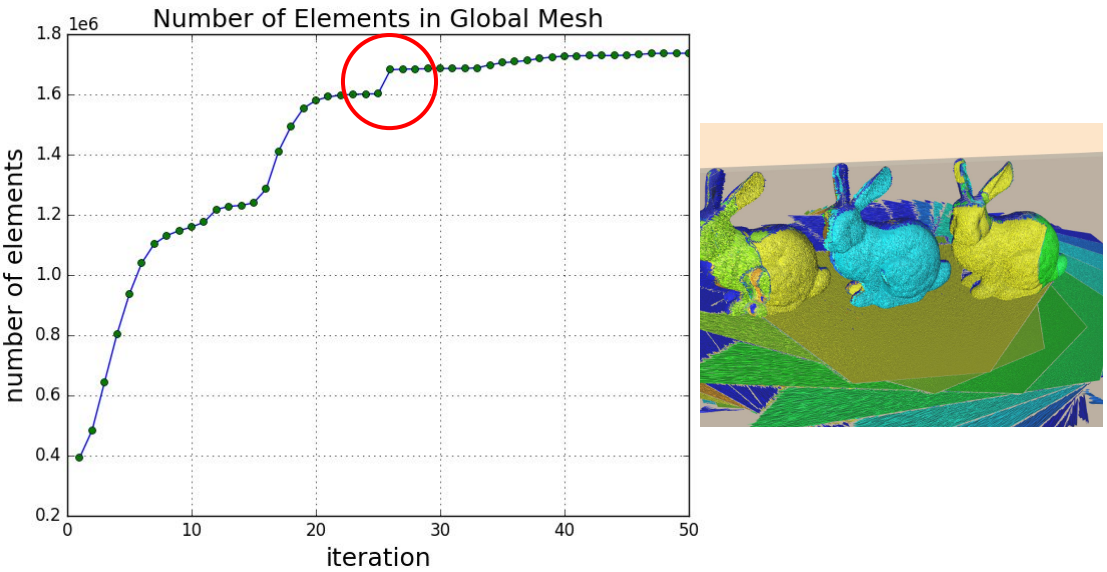


Figure 5.8: Experiment 3 global mesh results.

Chapter 6

Conclusion

The goal of MABDI is to identify data from the sensor that has not yet been represented in the map and use this data to add to the map. MABDI does this by leveraging the difference between what we are actually seeing and what we expect to see. MABDI can work in conjunction with any current mesh-based surface reconstruction algorithms, and can be thought of as a general means to provide introspection to those types of reconstruction methods.

The MABDI implementation was able to successfully perform in a realistic simulation environment. The results show how novel sensor data was successfully classified and used to add to the global mesh. Also, the MABDI algorithm runs at around 2Hz on a consumer grade laptop with an Intel i7 processor. This performance means that it is capable of real-world applications.

Currently MABDI is only designed to handle object addition, but the idea can be extended to handle both object addition and removal as discussed in Section 3.1. This would give the system the capability to handle highly dynamic environments such as a door opening and closing.

Appendix A

MABDI code

A.1 FilterDepthImage.py

FilterDepthImage.py

```
import vtk
from vtk.util.vtkAlgorithm import VTKPythonAlgorithmBase
from vtk.util import numpy_support
from vtk.numpy_interface import dataset_adapter as dsa

import numpy as np

from timeit import default_timer as timer
import logging


class FilterDepthImage(VTKPythonAlgorithmBase):
    """
    Create a depth image of a scene

    This class uses the geometric information of the scene (vtkPolyData) and the
    orientation of the depth sensor.
    """

    def __init__(self,
```

```
21                  name='none',
22                  offscreen=False,
23                  noise=0.0,
24                  depth_image_size=(640, 480)):
25         """
26         :param name: default='none'
27           Used for the logging statements.
28         :param offscreen:
29           Create the render window that is used to produce the depth image offscreen
       .
30         :param noise:
31           Noise to add to depth image.
32         :param depth_image_size:
33           Size of the depth image.
34         """
35
36         VTKPythonAlgorithmBase.__init__(self,
37                                         nInputPorts=0,
38                                         nOutputPorts=1, outputType='vtkImageData')
39         self._name = name
40
41         if type(noise) == bool:
42             if noise:
43                 noise = 0.002
44             else:
45                 noise = 0.0
46         self._noise = noise
47
48         # vtk render objects
49         self._ren = vtk.vtkRenderer()
50         self._renWin = vtk.vtkRenderWindow()
51         self._iren = vtk.vtkRenderWindowInteractor()
52
53         # wire them up
54         self._renWin.AddRenderer(self._ren)
55         self._iren.SetRenderWindow(self._renWin)
56
57         # offscreen rendering
58         if offscreen:
59             self._renWin.SetOffScreenRendering(1)
60
61         # kinect intrinsic parameters
62         # https://msdn.microsoft.com/en-us/library/hh438998.aspx
```

```
63          self._renWin.SetSize(depth_image_size)
64          self._ren.GetActiveCamera().SetViewAngle(60.0)
65          self._ren.GetActiveCamera().SetClippingRange(0.8, 4.0)
66          self._iren.GetInteractorStyle().SetAutoAdjustCameraClippingRange(0)
67
68          # have it looking down and underneath the "floor"
69          # so that it will produce a blank vtkImageData until
70          # set_sensor_orientation() is called
71          self._ren.GetActiveCamera().SetPosition(0.0, -20.0, 0.0)
72          self._ren.GetActiveCamera().SetFocalPoint(0.0, -25.0, 0.0)
73
74          # calculate image bounds
75          self._imageBounds = [0, 0, 0, 0]
76          viewport = self._ren.GetViewport()
77          size = self._renWin.GetSize()
78          self._imageBounds[0] = int(viewport[0] * size[0])
79          self._imageBounds[1] = int(viewport[1] * size[1])
80          self._imageBounds[2] = int(viewport[2] * size[0] + 0.5) - 1
81          self._imageBounds[3] = int(viewport[3] * size[1] + 0.5) - 1
82
83      def set_polydata(self, in_polydata):
84          """
85          What this filter will render and consequently produce a depth image of.
86          :param in_polydata: vtkAlgorithm that produces a vtkPolyData
87          """
88          logging.info('')
89
90          mapper = vtk.vtkPolyDataMapper()
91          mapper.SetInputConnection(in_polydata.GetOutputPort())
92
93          actor = vtk.vtkActor()
94          actor.SetMapper(mapper)
95
96          self._ren.AddActor(actor)
97
98          self._iren.Initialize()
99          self._iren.Render()
100
101     def set_polydata_empty(self):
102         """
103         Use to initialize this filter with an empty vtkPolyData
104         """
105         logging.info('')
```

```
106
107         polydata = vtk.vtkPolyData()
108
109         mapper = vtk.vtkPolyDataMapper()
110         mapper.SetInputDataObject(polydata)
111
112         actor = vtk.vtkActor()
113         actor.SetMapper(mapper)
114
115         self._ren.AddActor(actor)
116
117         self._iren.Initialize()
118         self._iren.Render()
119
120     def set_sensor_orientation(self, in_position, in_lookat):
121         """
122         :param in_position: Position of sensor in world coordinates.
123         :param in_lookat: Where the sensor is looking in world coordinates.
124         """
125         logging.info('position{} lookat{}'.format(in_position, in_lookat))
126
127         self._ren.GetActiveCamera().SetPosition(in_position)
128         self._ren.GetActiveCamera().SetFocalPoint(in_lookat)
129         self._iren.Render()
130
131     def get_vtk_camera(self):
132         return self._ren.GetActiveCamera()
133
134     def get_width_by_height_ratio(self):
135         return float(self._renWin.GetSize()[0]) / float(self._renWin.GetSize()[1])
136
137     def kill_render_window(self):
138         """
139         Kill render window that this instance owns. Only to be used when the user
140         is sure the filter will not be run again.
141         """
142         # http://stackoverflow.com/questions/15639762/close-vtk-window-python
143         self._renWin.Finalize()
144         self._iren.TerminateApp()
145         del self._renWin, self._iren
146
147     def RequestInformation(self, request, inInfo, outInfo):
148         logging.info('')
```

```
149         size = self._renWin.GetSize()
150         extent = (0, size[0] - 1, 0, size[1] - 1, 0, 0)
151         info = outInfo.GetInformationObject(0)
152         info.Set(vtk.vtkStreamingDemandDrivenPipeline.WHOLE_EXTENT(),
153                  extent, len(extent))
154         return 1
155
156     def RequestData(self, request, inInfo, outInfo):
157         logging.info('{}'.format(self._name))
158         start = timer()
159
160         # get the depth values
161         vfa = vtk.vtkFloatArray()
162         ib = self._imageBounds
163         self._renWin.GetZbufferData(ib[0], ib[1], ib[2], ib[3], vfa)
164
165         # add noise
166         if self._noise is not 0.0:
167             nvfa = numpy_support.vtk_to_numpy(vfa)
168             nvfa += self._noise * nvfa * np.random.normal(0.0, 1.0, nvfa.shape)
169             vfa = dsa.numpyTovtkDataArray(nvfa)
170
171         # pack the depth values into the output vtkImageData
172         info = outInfo.GetInformationObject(0)
173         ue = info.Get(vtk.vtkStreamingDemandDrivenPipeline.UPDATE_EXTENT())
174         out = vtk.vtkImageData.GetData(outInfo)
175         out.GetPointData().SetScalars(vfa)
176         out.SetExtent(ue)
177
178         # append meta data to the vtkImageData containing intrinsic parameters
179         out.sizex = self._renWin.GetSize()[0]
180         out.sizey = self._renWin.GetSize()[1]
181         out.viewport = self._ren.GetViewport()
182         vtktmat = self._ren.GetActiveCamera().GetCompositeProjectionTransformMatrix(
183             self._ren.GetTiledAspectRatio(),
184             0.0, 1.0)
185         vtktmat.Invert()
186         out.tmat = self._vtkmatrix_to_numpy(vtktmat)
187
188         end = timer()
189         logging.info('Execution time {:.4f} seconds'.format(end - start))
190
191         return 1
```

```
192
193     def _vtkmatrix_to_numpy(self, matrix):
194         """
195         Copies the elements of a vtkMatrix4x4 into a numpy array.
196
197         :param matrix: The matrix to be copied into an array.
198         :type matrix: vtk.vtkMatrix4x4
199         :rtype: numpy.ndarray
200         """
201         m = np.ones((4, 4))
202         for i in range(4):
203             for j in range(4):
204                 m[i, j] = matrix.GetElement(i, j)
205         return m
```

A.2 FilterClassifier.py

FilterClassifier.py

```
1  import vtk
2  from vtk.util.vtkAlgorithm import VTKPythonAlgorithmBase
3  from vtk.util import numpy_support
4
5  from timeit import default_timer as timer
6  import logging
7
8
9  class FilterClassifier(VTKPythonAlgorithmBase):
10     """
11     vtkAlgorithm with 2 inputs of vtkImageData and an output of vtkImageData
12     Input: Depth images
13     Output: Classified depth image
14     """
15
16     def __init__(self, param_classifier_threshold=0.01):
17         """
18         :param param_classifier_threshold: default=0.01
19           Threshold to determine when the difference in the depth images is too big
20           and is therefore a novel measurement.
21         :return:
```

```
22          """
23
24          VTKPythonAlgorithmBase.__init__(self,
25                                          nInputPorts=2, inputType='vtkImageData',
26                                          nOutputPorts=1, outputType='vtkImageData')
27
28          self._param_classifier_threshold = param_classifier_threshold
29
30          self._postprocess = []
31          self._postprocess_im1 = []
32          self._postprocess_im2 = []
33          self._postprocess_difim = []
34
35      def set_postprocess(self, do_postprocess):
36          self._postprocess = do_postprocess
37
38      def get_depth_images(self):
39          """
40          Get the depth images. User has to call set_postprocess(True) first.
41          :return: Depth images
42            return[0] - actual
43            return[1] - expected
44            return[2] - threshold absolute difference
45          """
46          return self._postprocess_im1, self._postprocess_im2, self._postprocess_difim
47
48      def RequestInformation(self, request, inInfo, outInfo):
49          logging.info('')
50
51          # input images dimensions
52          info = inInfo[0].GetInformationObject(0)
53          ue1 = info.Get(vtk.vtkStreamingDemandDrivenPipeline.UPDATE_EXTENT())
54          info = inInfo[1].GetInformationObject(0)
55          ue2 = info.Get(vtk.vtkStreamingDemandDrivenPipeline.UPDATE_EXTENT())
56          if ue1 != ue2:
57              logging.warning('Input images have different dimensions. {} {}'.format(
    ue1, ue2))
58
59          extent = ue1
60          info = outInfo.GetInformationObject(0)
61          info.Set(vtk.vtkStreamingDemandDrivenPipeline.WHOLE_EXTENT(),
62                   extent, len(extent))
63
```

```
64          return 1
65
66      def RequestData(self, request, inInfo, outInfo):
67          logging.info('')
68          start = timer()
69
70          # in images (vtkImageData)
71          inp1 = vtk.vtkImageData.GetData(inInfo[0])
72          inp2 = vtk.vtkImageData.GetData(inInfo[1])
73
74          # convert to numpy arrays
75          dim = inp1.GetDimensions()
76          im1 = numpy_support.vtk_to_numpy(inp1.GetPointData().GetScalars())\
77              .reshape(dim[1], dim[0])
78          dim = inp1.GetDimensions()
79          im2 = numpy_support.vtk_to_numpy(inp2.GetPointData().GetScalars())\
80              .reshape(dim[1], dim[0])
81
82          # difference in the images
83          # im1 is assumed to be from the actual sensor
84          # im2 is what we expect to see based on the world mesh
85          # Anywhere the difference is small, throw those measurements away
86          # by setting them to one. By doing this FilterDepthImageToSurface
87          # will assume they lie on the clipping plane and will remove them
88          difim = abs(im1 - im2) < self._param_classifier_threshold
89          if self._postprocess:
90              self._postprocess_im1 = im1.copy()
91              self._postprocess_im2 = im2.copy()
92              self._postprocess_difim = difim.copy()
93          imout = im1
94          imout[difim] = 1.0
95
96          info = outInfo.GetInformationObject(0)
97          ue = info.Get(vtk.vtkStreamingDemandDrivenPipeline.UPDATE_EXTENT())
98
99          # output vtkImageData
100         out = vtk.vtkImageData.GetData(outInfo)
101         out.SetExtent(ue)
102         (out.sizex, out.sizey, out.tmat, out.viewport) = \
103             (inp1.sizex, inp1.sizey, inp1.tmat, inp1.viewport)
104         out.GetPointData().SetScalars(
105             numpy_support.numpy_to_vtk(imout.reshape(-1)))
106
```

```
107         end = timer()
108         logging.info('Execution time {:.4f} seconds'.format(end - start))
109 
110         return 1
```

A.3 FilterDepthImageToSurface.py

FilterDepthImageToSurface.py

```
1  import vtk
2  from vtk.util.vtkAlgorithm import VTKPythonAlgorithmBase
3  from vtk.util import numpy_support
4  from vtk.numpy_interface import dataset_adapter as dsa
5  
6  from Utilities import DebugTimeVTKFilter
7  
8  import numpy as np
9  from scipy import ndimage
10 
11 from timeit import default_timer as timer
12 import logging
13 
14 
15 class FilterDepthImageToSurface(VTKPythonAlgorithmBase):
16     """
17     vtkAlgorithm with input of vtkImageData and output of vtkPolyData
18     This filter first defines a connectivity on the depth image that is like a
19     checkerboard but with two triangles in each square. It then throws away all
       points
20     farther than the param_farplane_threshold and all points with a large difference
21     between neighbors (controlled with param_convolution_theshold)
22     Input: Depth image
23     Output: Mesh created by projecting depth image
24     """
25 
26     def __init__(self,
27                  param_farplane_threshold=1.0,
28                  param_convolution_threshold=0.01):
29         """
30         Algorithm setup and define parameters.
```

```
31          :param param_farplane_threshold: default=1.0
32            Values on the depth image range from 0.0-1.0. Points with depth values
    greater
33            than param_farplane_threshold will be thrown away.
34          :param param_convolution_threshold: default=0.01
35            Convolution is used to determine pixel neighbors with a large difference.
    If
36            there is one, the point will be thrown away. This threshold controls
    sensitivity.
37          """
38
39          VTKPythonAlgorithmBase.__init__(self,
40                                          nInputPorts=1, inputType='vtkImageData',
41                                          nOutputPorts=1, outputType='vtkPolyData')
42
43          self._param_farplane_threshold = param_farplane_threshold
44          self.param_convolution_theshold = param_convolution_threshold
45
46          self._sizex = []
47          self._sizey = []
48          self._viewport = []
49
50          self._display_pts = []
51          self._viewport_pts = []
52          self._world_pts = []
53
54          self._points = vtk.vtkPoints()
55          self._polys = vtk.vtkCellArray()
56          self._polydata = vtk.vtkPolyData()
57          self._polydata.SetPoints(self._points)
58          self._polydata.SetPolys(self._polys)
59
60          self._extract = vtk.vtkExtractPolyDataGeometry()
61          DebugTimeVTKFilter(self._extract)
62          self._extract.SetInputData(self._polydata)
63          planefunc = vtk.vtkPlane()
64          planefunc.SetNormal(0.0, -1.0, 0.0)
65          planefunc.SetOrigin(0.0, -1.0, 0.0)
66          self._extract.SetImplicitFunction(planefunc)
67
68      def RequestData(self, request, inInfo, outInfo):
69
70          logging.info('')
```

```
71          start = timer()
72
73          # input (vtkImageData)
74          inp = vtk.vtkImageData.GetData(inInfo[0])
75
76          # if the vtkImageData size has changed or this is the first time
77          # save new size info and initialize containers
78          if (self._sizex, self._sizey, self._viewport) != (inp.sizex, inp.sizey, inp.
    viewport):
79              (self._sizex, self._sizey) = (inp.sizex, inp.sizey)
80              self._viewport = inp.viewport
81              self._init_containers()
82
83          # the incoming depth image
84          di = numpy_support.vtk_to_numpy(inp.GetPointData().GetScalars())\
85              .reshape((self._sizey, self._sizex))
86
87          # add z values to viewport_pts based on incoming depth image
88          self._viewport_pts[2, :] = di.reshape(-1)
89
90          # project to world coordinates
91          self._world_pts = np.dot(inp.tmat, self._viewport_pts)
92          self._world_pts = self._world_pts / self._world_pts[3]
93
94          """ Remove invalid points """
95
96          # index to pts outside sensor range (defined by vtkCamera clipping range)
97          outside_range = ~(di < self._param_farplane_threshold)
98
99          # find pixel neighbors with large differences in value
100         # http://docs.scipy.org/doc/scipy/reference/tutorial/ndimage.html
101         kh = np.array([[1, -1], [0, 0]])
102         edges_h = abs(ndimage.convolve(di,
103                                        kh,
104                                        mode='nearest',
105                                        origin=-1)) > self.param_convolution_theshold
106         kv = np.array([[1, 0], [-1, 0]])
107         edges_v = abs(ndimage.convolve(di,
108                                        kv,
109                                        mode='nearest',
110                                        origin=-1)) > self.param_convolution_theshold
111
112         # combine all the points found to be invalid
```

```
113         # and set them to a value underneath the "floor of the environment"
114         # http://stackoverflow.com/a/20528566/4068274
115         invalid_index = np.logical_or.reduce((outside_range.reshape(-1),
116                                               edges_h.reshape(-1),
117                                               edges_v.reshape(-1)))
118         self._world_pts[0:3, invalid_index] = np.array([[0.0], [-2.0], [0.0]])
119 
120         """ Update and set filter output """
121 
122         # update vtkPoints
123         vtkarray = dsa.numpyTovtkDataArray(self._world_pts[0:3, :].T)
124         self._points.SetData(vtkarray)
125 
126         # update output (vtkPolyData)
127         out = vtk.vtkPolyData.GetData(outInfo)
128         self._extract.Update()
129         logging.info('Number of triangles: {}'.format(self._extract.GetOutput().
    GetNumberOfCells()))
130         out.ShallowCopy(self._extract.GetOutput())
131 
132         end = timer()
133         logging.info('Execution time {:.4f} seconds'.format(end - start))
134 
135         return 1
136 
137     def _init_containers(self):
138         logging.info('Initializing arrays for projection calculation.')
139         tstart = timer()
140 
141         # helper variables (width, height)
142         (w, h) = (self._sizex, self._sizey)
143 
144         """ display points (list of all pixel coordinates) """
145 
146         self._display_pts = np.ones((2, w * h))
147         self._display_pts[0, :], self._display_pts[1, :] = \
148             zip(*[(j, i) for i in np.arange(h) for j in np.arange(w)])
149 
150         """ viewport points """
151         # https://github.com/Kitware/VTK/blob/52
    d45496877b00852a08a5b9819d109c2fd9bfab/Rendering/Core/vtkCoordinate.h#L26
152 
153         self._viewport_pts = np.ones((4, self._display_pts.shape[1]))
```

```
154         self._viewport_pts[0, :] = 2.0 * (self._display_pts[0, :] - w * self.
    _viewport[0]) / \
155             (w * (self._viewport[2] - self._viewport[0])) - 1.0
156         self._viewport_pts[1, :] = 2.0 * (self._display_pts[1, :] - h * self.
    _viewport[1]) / \
157             (h * (self._viewport[3] - self._viewport[1])) - 1.0
158 
159         """ new world points (just initializing the container) """
160 
161         self._world_pts = np.ones(self._viewport_pts.shape)
162 
163         """ cells (list of triangles created by connecting neighbors in depth image
    space ) """
164 
165         # connectivity on the depth image is almost like a checkerboard pattern
166         # except with two triangles in every checkerboard square
167         nt = (2*w)*(h-1)  # number of triangles
168         cells = np.zeros((3, nt), dtype=np.int)
169         i = 0
170         while i < (nt/2):
171             if ((i+1) % w) != 0:  # if on the side of the image skip
172                 cells[:, 2*i] = (i, i+1, w+i)
173                 cells[:, 2*i+1] = (i+1, w+i+1, w+i)
174             i += 1
175 
176         # remove columns with zeros (the ones we skipped in the while loop)
177         index = np.where(cells.any(axis=0))[0]  # all columns that are non zero
178         cells = cells[:, index]
179 
180         # turn our connectivity list into a vtk object (vtkCellArray)
181         for tpt in cells.T:
182             self._polys.InsertNextCell(3)
183             self._polys.InsertCellPoint(tpt[0])
184             self._polys.InsertCellPoint(tpt[1])
185             self._polys.InsertCellPoint(tpt[2])
186         self._polydata.SetPolys(self._polys)
187 
188         # time me
189         tend = timer()
190         logging.info('Initializing arrays for projection calculation {:.4f} seconds'
    .format(tend - tstart))
```

A.4 FilterWorldMesh.py

FilterWorldMesh.py

```
import vtk
from vtk.util.vtkAlgorithm import VTKPythonAlgorithmBase
from vtk.numpy_interface import dataset_adapter as dsa

import numpy as np
import matplotlib.pyplot as plt

from itertools import cycle

from timeit import default_timer as timer
import logging


class FilterWorldMesh(VTKPythonAlgorithmBase):
    """
    vtkAlgorithm with input vtkPolyData and output vtkPolyData
    Input: Surface to be added to the global mesh
    Output: The global mesh
    """
    def __init__(self, color=False):
        """
        :param color: default=False
          Color every new surface of the global mesh a different color.
        :return:
        """

        VTKPythonAlgorithmBase.__init__(self,
                                        nInputPorts=1, inputType='vtkPolyData',
                                        nOutputPorts=1, outputType='vtkPolyData')

        self._worldmesh = vtk.vtkAppendPolyData()

        # colormap for changing polydata on every iteration
        # http://matplotlib.org/examples/color/colormaps_reference.html
        self._color = color
        if self._color:
            gist_rainbow_r = plt.cm.get_cmap(name='gist_rainbow_r')
            mycm = gist_rainbow_r(range(160, 260, 5))[:, 0:3]
            self._colorcycle = cycle(mycm)
```

```
40
41      def RequestData(self, request, inInfo, outInfo):
42          logging.info('')
43          start = timer()
44
45          # input polydata
46          # have to make a copy otherwise polys will not show up in the render
47          # even though GetNumberOfCells() says they should be there
48          tmp = vtk.vtkPolyData.GetData(inInfo[0])
49          inp = vtk.vtkPolyData()
50          inp.ShallowCopy(tmp)
51
52          # change color of all cells
53          if self._color:
54              ncells = inp.GetNumberOfCells()
55              c = self._colorcycle.next()
56              vtkarray = dsa.numpyTovtkDataArray(np.tile(c, (ncells, 1)))
57              inp.GetCellData().SetScalars(vtkarray)
58
59          # add to world mesh
60          self._worldmesh.AddInputData(inp)
61          self._worldmesh.Update()
62
63          logging.info('Number of cells: in = {} total = {}'
64                       .format(inp.GetNumberOfCells(),
65                               self._worldmesh.GetOutput().GetNumberOfCells()))
66
67          # output world mesh
68          out = vtk.vtkPolyData.GetData(outInfo)
69          out.ShallowCopy(self._worldmesh.GetOutput())
70
71          end = timer()
72          logging.info('Execution time {:.4f} seconds'.format(end - start))
73
74          return 1
```

References

[1] M. W. Kadous, R. K.-M. Sheh, and C. Sammut, "Effective user interface design for rescue robotics," in *Proceeding of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction - HRI '06*. New York, New York, USA: ACM Press, mar 2006, p. 250.

[2] S. Thrun, "Robotic mapping: A survey," *Exploring artificial intelligence in the new millennium*, no. February, 2002.

[3] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *Computer*, vol. 21, no. 4, pp. 163–169, 1987.

[4] B. Freedman, A. Shpunt, M. Machline, and Y. Arieli, "Depth mapping using projected patterns," 2012.

[5] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments," *The International Journal of Robotics Research*, vol. 31, no. 5, pp. 647–663, apr 2012.

[6] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon, "KinectFusion: Real-time dense surface mapping and tracking," in *2011 10th IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2011*. IEEE, oct 2011, pp. 127–136.

[7] L. Xia, C.-C. Chen, and J. K. Aggarwal, "Human detection using depth information by Kinect," in *CVPR 2011 WORKSHOPS*. IEEE, jun 2011, pp. 15–22.

[8] J. Stowers, M. Hayes, and A. Bainbridge-Smith, "Altitude control of a quadrotor helicopter using depth map from Microsoft Kinect sensor," in *2011 IEEE International Conference on Mechatronics*. IEEE, apr 2011, pp. 358–362.

[9] M. Dissanayake, P. Newman, S. Clark, H. Durrant-Whyte, and M. Csorba, "A solution to the simultaneous localization and map building (SLAM) problem," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 3, pp. 229–241, jun 2001.

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

[11] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (SLAM): Part II," *Robotics & Automation Magazine*, no. September, 2006.

[12] R. Smith, M. Self, and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," *Autonomous robot vehicles*, 1990.

[13] S. Thrun, W. Burgard, and D. Fox, "A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots," *Autonomous Robots*, vol. 5, no. 3/4, pp. 253–271, 1998.

[14] J. Gutmann and K. Konolige, "Incremental mapping of large cyclic environments," in *Proceedings 1999 IEEE International Symposium on Computational Intelligence in Robotics and Automation. CIRA'99 (Cat. No.99EX375)*. IEEE, 1999, pp. 318–325.

[15] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust Monte Carlo localization for mobile robots," *Artificial Intelligence*, vol. 128, no. 1-2, pp. 99–141, may 2001.

[16] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," *Proceedings of the National conference on Artificial Intelligence*, pp. 593–598, 2002.

[17] S. Thrun, W. Burgard, and D. Fox, "A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 1. IEEE, 2000, pp. 321–328.

[18] Y. Liu and R. Emery, "Using EM to learn 3D models of indoor environments with mobile robots," in *Machine Learning-International Workshop Then Conference*, 2001.

[19] A. Davison, "Real-time simultaneous localisation and mapping with a single camera," in *Proceedings Ninth IEEE International Conference on Computer Vision*. IEEE, 2003, pp. 1403–1410 vol.2.

[20] S. Thrun, D. Hahnel, D. Ferguson, M. Montemerlo, R. Triebel, W. Burgard, C. Baker, Z. Omohundro, S. Thayer, and W. Whittaker, "A system for volumetric robotic mapping of abandoned mines," in *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, vol. 3. IEEE, 2003, pp. 4270–4275.

[21] A. Howard, D. Wolf, and G. Sukhatme, "Towards 3D mapping in large urban environments," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 1, pp. 419–424, 2004.

[22] D. Cole and P. Newman, "Using laser range data for 3D SLAM in outdoor environments," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE, 2006, pp. 1556–1563.

[23] G. Klein and D. Murray, "Parallel Tracking and Mapping for Small AR Workspaces," in *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*. IEEE, nov 2007, pp. 1–10.

[24] L. Paz, P. Pinies, J. Tardos, and J. Neira, "Large-Scale 6-DOF SLAM With Stereo-in-Hand," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 946–957, oct 2008.

[25] K. Konolige and M. Agrawal, "FrameSLAM: From Bundle Adjustment to Real-Time Visual Mapping," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1066–1077, oct 2008.

[26] H. Strasdat, J. Montiel, and A. Davison, "Scale drift-aware large scale monocular SLAM," *Proceedings of Robotics: Science and Systems (RSS). Vol. 2. No. 3. 2010*, 2010.

[27] N. Engelhard, F. Endres, and J. Hess, "Real-time 3D visual SLAM with a hand-held RGB-D camera," *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum*, no. c, 2011.

[28] S. Rusinkiewicz and M. Levoy, "Efficient variants of the ICP algorithm," in *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*. IEEE Comput. Soc, 2001, pp. 145–152.

[29] B. Yamauchi, A. Schultz, and W. Adams, "Mobile Robot Exploration and Map-Building with Continuous Localization," *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, vol. 4, no. May, pp. 3715–3720, 1998.

[30] A. Schultz and W. Adams, "Continuous localization using evidence grids," in *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, vol. 4. IEEE, 1998, pp. 2833–2839.

[31] R. Biswas, B. Limketkai, S. Sanner, and S. Thrun, "Towards object mapping in non-stationary environments with mobile robots," in *IEEE/RSJ International Conference on Intelligent Robots and System*, vol. 1. IEEE, 2002, pp. 1014–1019.

[32] A. Eliazar and R. Parr, "DP-SLAM 2.0," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004.* IEEE, 2004, pp. 1314–1320 Vol.2.

[33] M. Magnusson, A. Lilienthal, and T. Duckett, "Scan registration for autonomous mining vehicles using 3D-NDT," *Journal of Field Robotics*, vol. 24, no. 10, pp. 803–827, oct 2007.

[34] A. Nüchter, K. Lingemann, J. Hertzberg, and H. Surmann, "6D SLAM3D mapping outdoor environments," *Journal of Field Robotics*, vol. 24, no. 8-9, pp. 699–722, aug 2007.

[35] A. Huang and A. Bachrach, "Visual odometry and mapping for autonomous flight using an RGB-D camera," *Int. Symposium on ...*, pp. 1–16, 2011.

[36] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard, "An evaluation of the RGB-D SLAM system," in *2012 IEEE International Conference on Robotics and Automation*, vol. 3, no. c, IEEE. IEEE, may 2012, pp. 1691–1696.

[37] C. Martin and S. Thrun, "Real-time acquisition of compact volumetric 3D maps with mobile robots," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, vol. 1. IEEE, 2002, pp. 311–316.

[38] D. Viejo and M. Cazorla, "Unconstrained 3D-Mesh Generation Applied to Map Building," *Progress in Pattern Recognition, Image Analysis and Applications*, vol. 3287, pp. 161–207, 2004.

[39] F. Cazals and J. Giesen, "Delaunay Triangulation Based Surface Reconstruction : Ideas and Algorithms," *INRIA Rapport de recherche*, no. November, pp. 1–45, 2004.

[40] J. Weingarten and R. Siegwart, "3D SLAM using planar segments," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, oct 2006, pp. 3062–3067.

[41] A. Akbarzadeh, J.-M. Frahm, P. Mordohai, B. Clipp, C. Engels, D. Gallup, P. Merrell, M. Phelps, S. Sinha, B. Talton, L. Wang, Q. Yang, H. Stewenius, R. Yang, G. Welch, H. Towles, D. Nister, and M. Pollefeys, "Towards Urban 3D Reconstruction from Video," in *Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*. IEEE, jun 2006, pp. 1–8.

[42] M. Pollefeys, D. Nistér, J.-M. Frahm, A. Akbarzadeh, P. Mordohai, B. Clipp, C. Engels, D. Gallup, S.-J. Kim, P. Merrell, C. Salmi, S. Sinha, B. Talton, L. Wang, Q. Yang, H. Stewénius, R. Yang, G. Welch, and H. Towles, "Detailed Real-Time Urban 3D Reconstruction from Video," *International Journal of Computer Vision*, vol. 78, no. 2-3, pp. 143–167, oct 2007.

[43] M. Sainz, R. Pajarola, and Y. Meng, "Depth-Mesh Objects: Fast Depth-Image Meshing and Warping," *Ukpmc.Ac.Uk*, no. 03, 2003.

[44] J. Poppinga, N. Vaskevicius, A. Birk, and K. Pathak, "Fast plane detection and polygonalization in noisy 3D range images," in *Intelligent Robots and Systems (IROS) 2008*. Ieee, sep 2008, pp. 3378–3383.

[45] R. A. Newcombe and A. J. Davison, "Live dense reconstruction with a single moving camera," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE, jun 2010, pp. 1498–1505.

[46] Y. Ohtake, A. Belyaev, and H. Seidel, "A multi-scale approach to 3D scattered data interpolation with compactly supported basis functions," in *2003 Shape Modeling International.* IEEE Comput. Soc, 2003, pp. 153–161.

[47] J. Stühmer, S. Gumhold, and D. Cremers, "Real-time dense geometry from a handheld camera," *Pattern Recognition*, pp. 11–20, 2010.

[48] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments," *The International Journal of Robotics Research*, vol. 31, no. 5, pp. 647–663, feb 2012.

[49] T. Weise, T. Wismer, B. Leibe, and L. Van Gool, "In-hand scanning with online loop closure," *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, pp. 1630–1637, sep 2009.

[50] H. Pfister, M. Zwicker, J. van Baar, and M. Gross, "Surfels : Surface Elements as Rendering Primitives," in *SIGGRAPH 2000*. New York, New York, USA: ACM Press, jul 2000, pp. 335–342.

[51] T. Whelan, M. Kaess, and M. Fallon, "Kintinuous: Spatially extended kinectfusion," *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, p. 7, 2012.

[52] T. Whelan, H. Johannsson, M. Kaess, J. J. Leonard, and J. B. McDonald, "Robust Tracking for Real-Time Dense RGB-D Mapping with Kintinuous," in *ICRA 2013*, no. MIT-CSAIL-TR-2012-031. Computer Science and Artificial Intelligence Laboratory, MIT, sep 2012.

[53] L.-K. Cahier, T. Ogata, and H. Okuno, "Incremental probabilistic geometry estimation for robot scene understanding," in *ICRA 2012*. Ieee, may 2012, pp. 3625–3630.

[54] M. Gopi and S. Krishnan, "A fast and efficient projection-based approach for surface reconstruction," in *SIBGRAPI'02*, 2002, pp. 0–7.

[55] R. Mencl and H. Muller, "Interpolation and approximation of surfaces from three-dimensional scattered data points," *Scientific Visualization Conference, 1997*, 1997.

[56] H. Hoppe, T. DeRose, and T. Duchamp, *Surface reconstruction from unorganized points*. ACM, 1992, no. July 1992.

[57] H. Edelsbrunner and E. P. Mücke, "Three-dimensional alpha shapes," *ACM Transactions on Graphics*, vol. 13, no. 1, pp. 43–72, jan 1994.

[58] J. Bloomenthal, "An Implicit Surface Polygonizer," *In Graphics Gems IV*, pp. 324–349, 1994.

[59] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *SIGGRAPH '96*, 1996, pp. 303–312.

[60] K. Pulli, T. Duchamp, H. Hoppe, J. McDonald, L. Shapiro, and W. Stuetzle, "Robust meshes from multiple range maps," in *International Conference on Recent Advances in 3-D Digital Imaging and Modeling*. IEEE Comput. Soc. Press, 1997, pp. 205–211.

[61] H. Surmann, A. Nüchter, and J. Hertzberg, "An autonomous mobile robot with a 3D laser range finder for 3D exploration and digitalization of indoor environments," *Robotics and Autonomous Systems*, vol. 45, no. 3-4, pp. 181–198, dec 2003.

[62] H. Zhao, S. Osher, and R. Fedkiw, "Fast surface reconstruction using the level set method," in *Variational and Level Set Methods in Computer Vision, 2001*, 2001, pp. 0–7.

[63] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans, "Reconstruction and representation of 3D objects with radial basis functions," *SIGGRAPH '01*, pp. 67–76, 2001.

[64] D. Terzopoulos and M. Vasilescu, "Sampling and Reconstruction with Adaptive Meshes," in *Computer Vision and Pattern Recognition, 1991. Proceedings CVPR '91.*, 1991.

[65] M. Vasilescu and D. Terzopoulos, "Adaptive meshes and shells: irregular triangulation, discontinuities, and hierarchical subdivision," in *Computer Vision and Pattern Recognition*, no. 1. IEEE Comput. Soc. Press, 1992, pp. 3–6.

[66] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Mesh Optimization," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '93, vol. d. New York, NY, USA: ACM, 1993, pp. 19–25.

[67] W.-C. Huang and D. Goldgof, "Adaptive-size meshes for rigid and nonrigid shape analysis and synthesis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 6, pp. 611–616, jun 1993.

[68] M. Rutishauser, M. Stricker, and M. Trobina, "Merging range images of arbitrarily shaped objects," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*, pp. 573–580, 1994.

[69] H. Delingette, "Simplex meshes: a general representation for 3D shape reconstruction," in *Computer Vision and Pattern Recognition*, 1994, pp. 856–859.

[70] G. Turk and M. Levoy, "Zippered polygon meshes from range images," in *SIGGRAPH '94*. New York, New York, USA: ACM Press, 1994, pp. 311–318.

[71] Y. Chen and G. Medioni, "Description of Complex Objects from Multiple Range Images Using an Inflating Balloon Model," *Computer Vision and Image Understanding*, vol. 61, no. 3, pp. 325–334, may 1995.

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

[73] M. Gopi, S. Krishnan, and C. Silva, "Surface reconstruction based on lower dimensional localized Delaunay triangulation," *Computer Graphics Forum*, vol. 19, no. 3, 2001.

[74] I. Ivrissimtzis, W.-K. Jeong, and H.-P. Seidel, "Using growing cell structures for surface reconstruction," *2003 Shape Modeling International.*, vol. 2003, pp. 78–86, 2003.

[75] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva, "Point set surfaces," *Proceedings Visualization, 2001. VIS '01.*, pp. 21–537, 2004.

[76] Z. C. Marton, R. B. Rusu, and M. Beetz, "On fast surface reconstruction methods for large and noisy point clouds," in *2009 IEEE International Conference on Robotics and Automation*. IEEE, may 2009, pp. 3218–3223.

[77] W. J. Schroeder, B. Lorensen, and K. Martin, *The Visualization Toolkit*, 4th ed. Kitware, 2006.

[78] Kitware. (2016) VTK The Visualization Toolkit. [Online]. Available: http://www.vtk.org/overview/

[79] S. N. Labs. (2016) SNL Computational Systems and Software Environment. [Online]. Available: http://www.sandia.gov/asc/computational_systems/

[80] M. F. Fallon, H. Johannsson, and J. J. Leonard, "Efficient Scene Simulation for Robust Monte Carlo Localization using an RGB-D Camera," in *2012 IEEE International Conference on Robotics and Automation*. IEEE, may 2012, pp. 1663–1670.

[81] Microsoft. (2016) Microsoft Robotics Kinect Sensor. [Online]. Available: https://msdn.microsoft.com/en-us/library/hh438998.aspx

[82] K. Khoshelham and S. O. Elberink, "Accuracy and resolution of Kinect depth data for indoor mapping applications." *Sensors (Basel, Switzerland)*, vol. 12, no. 2, pp. 1437–54, 2012.

[83] A. Nealen, T. Igarashi, O. Sorkine, and M. Alexa, "Laplacian mesh optimization," *Proceedings of the 4th ...*, p. 381, 2006.