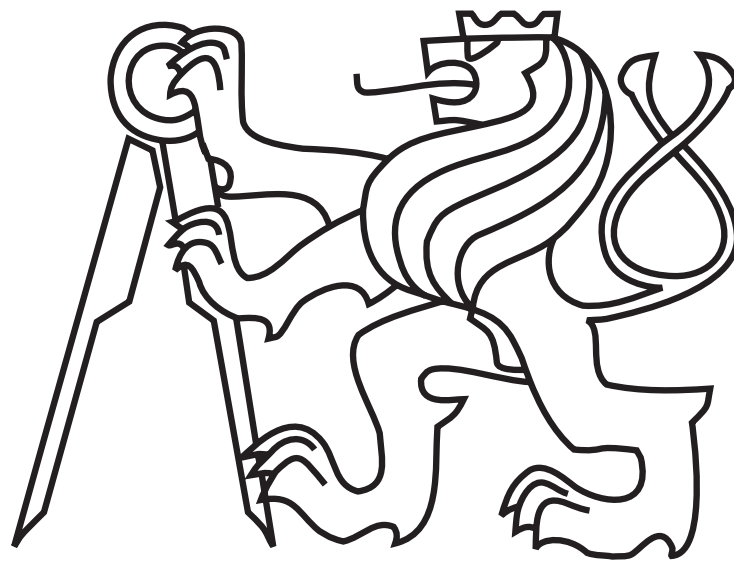


CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

# DIPLOMA THESIS



Jiří Mosinger

**A geometric approach to exploration of an unknown 3D environment**

**Department of Cybernetics**

Thesis supervisor: **RNDr. Miroslav Kulich, Ph.D.**

## **Prohlášení autora práce**

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....

.....

## **Acknowledgements**

I would like to express my thanks to the thesis supervisor RNDr. Miroslav Kulich, Ph.D. for his professional guidance, countless advices and overall support. I would also like to thank my girlfriend and family for their love and support.

## ZADÁNÍ DIPLOMOVÉ PRÁCE

**Student:** Bc. Jiří M o s i n g e r

**Studijní program:** Kybernetika a robotika (magisterský)

**Obor:** Robotika

**Název tématu:** Geometrický přístup k prohledávání neznámého 3D prostředí

### Pokyny pro vypracování:

Prohledávání neznámého prostoru je komplexní úloha vyžadující řešení základních robotických úloh: od plánování trajektorií, řízení, stavby mapy, lokalizaci, až po generování cílů, kam v dalším kroku jet. Cílem práce bude vyvinout softwarové prostředí pro realizaci prohledávání ve 3D. Hlavní důraz přitom bude kladen na reprezentaci prozkoumávaného prostoru a plánování v této reprezentaci.

1. Seznamte se s knihovnamí Robot Operating System (<http://ros.org>), Point Cloud Library (<http://pointclouds.org/>) a Carve (<http://carve-csg.com/>) a s metodami prohledávání neznámého 3D prostředí.
2. Implementujte metodu pro reprezentaci 3D scanu (získaného např. Senzorem MS Kinect) mnohostěnem s přidanou informací o typu každé stěny.
3. Implementujte skládání mnohostěňů získaných v předchozím kroku do geometrické 3D mapy a plánování v této mapě.
4. S použitím implementovaných metod vytvořte rámec pro prohledávání neznámého 3D prostředí.
5. Funkčnost implementovaných algoritmů ověřte experimenty. Experimentální výsledky popište se zaměřením na rychlost, robustnost a možné využití zkoumaného algoritmu.

### Seznam odborné literatury:

- [1] L. J. Latecki, R. Lakämper: Convexity rule for shape decomposition based on discrete contour evolution. *Comput. Vis. Image Underst.* 73, 3 (March 1999), 441-454.
- [2] T. Juchelka: Exploration algorithms in a polygonal domain, diploma thesis, Dept. of Cybernetics, FEE, CTU in Prague, 2013.
- [3] R.B. Rusu, S. Cousins: 3D is here: Point Cloud Library (PCL), IEEE International Conference on Robotics and Automation (ICRA), pp.1-4, 9-13 May 2011.
- [4] W.J. Schroeder, K. Martin, W. Lorensen: *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Third Edition. Kitware, Inc. 2006.

**Vedoucí diplomové práce:** RNDr. Miroslav Kulich, Ph.D.

**Platnost zadání:** do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic  
**vedoucí katedry**

prof. Ing. Pavel Ripka, CSc.  
**děkan**

V Praze dne 10. 1. 2014

## DIPLOMA THESIS ASSIGNMENT

**Student:** Bc. Jiří M o s i n g e r

**Study programme:** Cybernetics and Robotics

**Specialisation:** Robotics

**Title of Diploma Thesis:** A Geometric Approach to Exploration of an Unknown 3D Environment

### Guidelines:

Exploration of an unknown environment is a complex task requiring solution of fundamental robotic problems: trajectory planning, control, map building, localization, and generation of next goals. The thesis aims to develop a software framework for exploration of an unknown 3D environment. Special emphasis will be put on the environment representation and planning on it.

1. Get acquainted with Robot Operating System (<http://ros.org>), Point Cloud Library (<http://pointclouds.org/>), and Carve (<http://carve-csg.com/>) libraries and with methods for exploration of a 3D environment.
2. Implement a method for 3D scan (gathered with e.g. MS Kinect sensor) representation with a polyhedron with added information about its faces' types.
3. Implement composition of polyhedra built in the previous step into a geometric 3D map and planning on this map.
4. Use the implemented methods to create a framework for exploration of an unknown 3D environment by a mobile robot.
5. Verify functionality of the implemented algorithms experimentally. Describe the obtained results with respect to the algorithms' speed, robustness, and possible application.

### Bibliography/Sources:

- [1] L. J. Latecki, R. Lakämper: Convexity rule for shape decomposition based on discrete contour evolution. *Comput. Vis. Image Underst.* 73, 3 (March 1999), 441-454.
- [2] T. Juchelka: Exploration algorithms in a polygonal domain, diploma thesis, Dept. of Cybernetics, FEE, CTU in Prague, 2013.
- [3] R.B. Rusu, S. Cousins: 3D is here: Point Cloud Library (PCL), IEEE International Conference on Robotics and Automation (ICRA), pp.1-4, 9-13 May 2011.
- [4] W.J. Schroeder, K. Martin, W. Lorensen: *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Third Edition. Kitware, Inc. 2006.

**Diploma Thesis Supervisor:** RNDr. Miroslav Kulich, Ph.D.

**Valid until:** the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic  
**Head of Department**

prof. Ing. Pavel Ripka, CSc.  
**Dean**

Prague, January 10, 2014

## *Abstrakt*

Tato diplomová práce se zabývá exploračí neznámého 3D prostředí mobilním robotem. Robot postupně zkoumá své okolí a na základě získaných informací vytváří mapu prozkoumaného prostředí. Tento problém je ve 2D prostředí tradičně řešen pomocí mřížky obsazenosti, zatímco nové přístupy jsou založeny na použití polygonální domény. Tato práce volí druhý přístup a upravuje ho pro 3D prostředí použitím polyhedrální domény. Tento přístup s sebou přináší jak výhody, tak některé problémy. Typickými příklady použití explorače jsou vojenské a záchranné operace. Hlavním cílem této práce je vyvinutí frameworku v ROSu, který bude poskytovat prostředky k exploraci 3D prostředí založeného na polyhedrální doméně. Získané výsledky jsou diskutovány na konci této diplomové práce.

## *Abstract*

This thesis deals with an exploration of an unknown 3D environment by a mobile robot. The robot continuously explores its surroundings and it creates a map of the explored environment based on received information. This problem is traditionally solved in 2-dimensional environment by using an occupancy grid while more novel approach is based on polygonal domain. This work takes the latter option and modifies it for 3D environment by using polyhedral domain. This approach brings advantages along with some caveats. Typical applications of exploration are military or rescue operations. The main goal of this thesis is to develop a framework in ROS that provides means of exploring 3D environment based on polyhedral domain. Obtained results are discussed at the end of this paper.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Exploration</b>	<b>2</b>
<b>3</b>	<b>From point cloud to polyhedron</b>	<b>4</b>
3.1	Chapter outline . . . . .	4
3.2	Polyhedron . . . . .	5
3.3	Point cloud . . . . .	5
3.4	Point triangulation . . . . .	6
3.4.1	Delaunay triangulation . . . . .	6
3.4.2	2D grid method . . . . .	7
3.4.3	Neighbor selection . . . . .	7
3.5	Polygon reduction . . . . .	8
3.5.1	Reduction outline . . . . .	9
3.5.2	Centroid method . . . . .	10
3.5.3	LSQ method . . . . .	11
3.5.4	Shape decomposition . . . . .	11
3.5.5	vtkDecimation . . . . .	12
3.5.6	Summary . . . . .	12
3.6	Re-triangulation . . . . .	12
3.6.1	Ear Clipping . . . . .	12
3.6.2	Triangulation of monotone polygons . . . . .	12
3.6.3	Trapezoidalization . . . . .	14
3.6.4	Constrained Delaunay triangulation . . . . .	15
3.6.5	Summary . . . . .	16

3.7	Line reduction . . . . .	16
3.7.1	Contribution value . . . . .	16
3.8	Creation of a polyhedron . . . . .	17
3.9	Frames transformation . . . . .	17
<b>4</b>	<b>From polyhedron to path</b>	<b>19</b>
4.1	Chapter outline . . . . .	19
4.2	Frontiers . . . . .	20
4.2.1	Frontiers in occupancy grid . . . . .	20
4.2.2	Frontiers in polygonal domain . . . . .	21
4.2.3	Frontiers in polyhedral domain . . . . .	21
4.3	Dual graph . . . . .	21
4.3.1	Polygonal domain . . . . .	22
4.3.2	Polyhedral domain . . . . .	23
4.4	Frontiers in a dual graph . . . . .	25
4.4.1	Polygonal domain . . . . .	25
4.4.2	Polyhedral domain . . . . .	25
4.5	Path planning . . . . .	26
4.5.1	Shortest path . . . . .	26
4.5.2	Breadth-first search . . . . .	27
4.5.3	Depth-first search . . . . .	27
4.5.4	Dijkstra search algorithm . . . . .	27
4.5.5	A* search algorithm . . . . .	29
4.5.6	Planning to multiple frontiers (goals) . . . . .	29
<b>5</b>	<b>Framework</b>	<b>31</b>
5.1	ROS . . . . .	31
5.2	Used 3rd party libraries . . . . .	32
5.3	Binary heap . . . . .	33
5.4	Point triangulation . . . . .	33
5.4.1	Duplicate data . . . . .	33
5.4.2	Missing data . . . . .	34
5.5	Polygon reduction . . . . .	35



---

5.5.1	Binary heap . . . . .	35
5.5.2	Re-triangulation . . . . .	35
5.6	Polyhedral operations . . . . .	35
5.6.1	Polyhedron transformations between frames . . . . .	36
5.6.2	Union of polyhedrons . . . . .	36
5.7	Planner . . . . .	37
5.7.1	Frontier identification in polyhedron . . . . .	37
5.7.2	Path planning . . . . .	37
5.7.3	Tetrahedralization . . . . .	38
<b>6</b>	<b>Experiments</b>	<b>39</b>
6.1	Quality of reduction . . . . .	39
6.1.1	Scenes . . . . .	39
6.2	Time of polygon reduction . . . . .	44
6.3	Quality of map creation . . . . .	44
6.4	Discussion . . . . .	44
<b>7</b>	<b>Conclusion</b>	<b>48</b>
	<b>Appendices</b>	<b>52</b>
<b>A</b>	<b>CD Content</b>	<b>53</b>
<b>B</b>	<b>Command line switches</b>	<b>54</b>

## List of Figures

3.1	Example of a polyhedron . . . . .	5
3.2	Delaunay triangulation . . . . .	6
3.3	Triangulation . . . . .	8
3.4	Process of re-triangulation . . . . .	9
3.5	Triangulation of a monotone polygon . . . . .	13
3.6	Trapezoidalization of a polygon . . . . .	15
3.7	Constrained Delaunay triangulation . . . . .	15
3.8	Line reduction . . . . .	16
3.9	Polyhedron . . . . .	18
4.1	Occupancy grid . . . . .	20
4.2	Polygonal domain . . . . .	21
4.3	Voronoi diagram and its connection to Delaunay triangulation . . . . .	22
4.4	Dual graph without frontiers . . . . .	23
4.5	Dual graph with frontiers . . . . .	26
6.1	Scene 01 . . . . .	40
6.2	Comparison of two polygonal reduction methods on Scene 01 . . . . .	41
6.3	Scene 02 . . . . .	42
6.4	Comparison of two polygonal reduction methods on Scene 02 . . . . .	43
6.5	Scenes . . . . .	45
6.6	World map creation . . . . .	46

# List of Tables

4.1	Comparison between constrained Delaunay triangulation and tetrahedralization	24
5.1	Time complexity of a binary heap . . . . .	33
5.2	Summary of triangulation methods provided by PolyPartition . . . . .	36
6.1	Execution times . . . . .	44
A.1	CD Content . . . . .	53
B.1	Command line switches . . . . .	54

## List of Scenarios

1	Pseudocode of the whole process . . . . .	3
2	Pseudocode of makePolyhedron process . . . . .	5
3	Pseudocode of point cloud triangulation . . . . .	7
4	Pseudocode of getNextNeighbor process . . . . .	8
5	Pseudocode of a polygon reduction . . . . .	10
6	Triangulation of a monotone polygon . . . . .	14
7	Creation of a polyhedron . . . . .	17
8	Pseudocode of path planning . . . . .	19
9	Pseudocode of Dijkstra search algorithm . . . . .	28
10	Pseudocode of path reconstruction . . . . .	28
11	Pseudocode of A* search algorithm . . . . .	30

# Chapter 1

## Introduction

Exploration is an important problem in mobile robotics. There is sometimes need to deploy a mobile robot into an unknown environment and let it explore and build a map of this environment. Yamauchi introduced a solution to the exploration problem which is based on identification of boundaries between known and unknown environment. These boundaries are called frontiers in his work and they function as points of interest to which robots plan their paths. When they get to them, they look around and pick another frontier once again [1].

Juchelka [2] built upon this idea when working on exploration of an unknown environment using multiple robots. This topic was solved several times before on a map called occupancy grid. Juchelka, however, took a different approach when he considered a map made of polygons. This so called polygonal domain brings advantages along with some caveats. Disadvantages are, that methods once working on an occupancy grid need to be adjusted for a polygonal domain. Among advantages belongs potentially more detailed map with lower memory usage when compared to occupancy grid.

This paper takes Juchalka's work into account and tries to extend it from polygonal domain to polyhedral domain or, when talking about space dimensions, from 2-dimensional space to the 3rd dimension.

The thesis is divided into several chapters. Chapter 2 serves as an introduction to exploration in 3D environment using polyhedral domain. Chapter 3 describes the process of converting a point cloud to a polyhedron. Path planning is described in Chapter 4. Inside look into implementation details is given in Chapter 5 and Chapter 6 presents results of made experiments. Summary and evaluation of this paper are presented in Chapter 7.

## Chapter 2

# Exploration

As was said in previous chapter, exploration is an important problem in mobile robotics. It deals with procedural map construction during the exploration of an unknown environment. This paper focuses on exploration of 3D environment and map construction on polyhedral domain which is done by converting a sequence of point clouds to a polyhedron.

Scenario 1 presents the algorithm of acquiring a map of 3D scene through processing point clouds taken by a stereo camera. The algorithm assumes input in form of a continuous sequence of point clouds. After the sequence ends, the algorithm outputs a map of 3D scene. Line 1 of the scenario is a flag indicating that the first point cloud is being processed.

The while cycle on lines 2 through 11 takes care of incoming point clouds while the stereo camera is recording. A polyhedron is created from a point cloud on line 3. The process of creation of the polyhedron from the point cloud is thoroughly described in Chapter 3.

Line 4 checks whether the first point cloud is being processed. If the condition is met, then the created polyhedron from line 3 is copied to a global variable *polyhedra* on line 6. If the condition on line 4 is not met, then there are two polyhedrons - one from previous run(s) and one from the current point cloud. Line 8 makes a union of these two polyhedrons. A C++ library Carve was used in this paper to perform this boolean operation. Carve is described in Section 5.2.

Line 10 deals with planning on a polyhedral domain. The process of planning is explained in Chapter 4. Finally, line 13 returns the constructed map of the 3D scene made of sequence of point clouds created by the stereo camera.

**Input:** *pointCloudStream*

**Output:** *map*

1. *firstCloud*  $\leftarrow$  *true*
2. **while** *isCameraStreaming* **do**
3.     *polyhedron*  $\leftarrow$  *makePolyhedron(pointCloudStream)*
4.     **if** *firstCloud* **then**
5.         *firstCloud*  $\leftarrow$  *false*
6.         *polyhedra*  $\leftarrow$  *polyhedron*
7.     **else**
8.         *polyhedra*  $\leftarrow$  *makeUnion(polyhedron, polyhedra)*
9.     **end if**
10.     *plan(polyhedra)*
11. **end while**
12. *map*  $\leftarrow$  *polyhedra*
13. **return** *map*

**Scenario 1:** Pseudocode of the whole process.

## Chapter 3

# From point cloud to polyhedron

This chapter describes the process of converting a point cloud acquired by a 3D sensor (e.g. Kinect) to a polyhedron. This is done by point cloud triangulation, followed by polygon reduction to minimize memory and computational requirements. Then, the polyhedron can be finally created.

The whole process of converting the point cloud to a polyhedron is useful for 3D scene reconstruction upon which a map is created containing points of interest (frontiers) that serve as indicators to the next desirable location to be explored by the robot.

### 3.1 Chapter outline

Scenario 2 presents the pseudocode that converts a point cloud into a polyhedron. Basic definition of a point cloud is found in Section 3.3. The algorithm assumes the input in form of a continuous sequence of point clouds, from which it selects the latest one. The algorithm then outputs a polyhedron.

The latest point cloud is selected on line 1. Line 2 handles acquiring of transformation matrix used on line 8 for transforming polyhedron's points from camera coordinates to world coordinates. This process is described in Section 3.9.

Intrinsic parameters of the stereo camera are acquired on line 3 and are used for filling any missing data in the point cloud on line 4. Lines 1 to 4 are further described in Chapter 5 that deals with implementation details.

Triangulation of the point cloud takes place on line 5 and the accompanying process is explained in Section 3.4. Line 6 represents a polygon reduction of triangulated point cloud which is described in Section 3.5 and Section 3.6.

Finally, the polyhedron is created on line 7 and Section 3.8 focuses on this topic. Simple definition of polyhedron is given in Section 3.2.



**Input:** *pointCloudStream*

**Output:** *polyhedron*

1.  $cloud \leftarrow getNextPointCloud(pointCloudStream)$
2.  $tf \leftarrow getNextTF(pointCloudStream)$
3.  $intrinsicPar \leftarrow getNextIntrinsicPar(pointCloudStream)$
4.  $cloud \leftarrow fillMissingData(cloud, intrinsicPar)$
5.  $tria \leftarrow triangulatePointCloud(cloud)$
6.  $reducePolygons(tria)$
7.  $polyhedron \leftarrow createPolyhedron(tria)$
8.  $transformPolyhedron(polyhedron, tf)$
9. **return** *polyhedron*

**Scenario 2:** Pseudocode of makePolyhedron process.

## 3.2 Polyhedron

A polyhedron is defined by a set of vertices in 3-dimensional space and a set of faces which define the connections between the vertices. The face may contain more than three vertices. In that case, all vertices of the face have to lie on the same plane. A polyhedron can be also viewed as a 3D polygonal mesh <sup>1</sup>.

An example of a simple polyhedron is a cube on Figure 3.1.

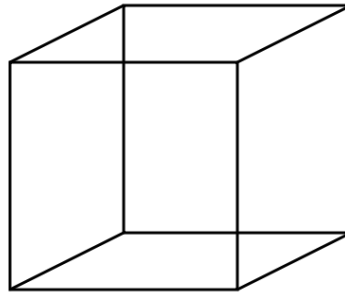


Figure 3.1: An example of a polyhedron.

## 3.3 Point cloud

Let  $S$  be a set of points in a coordinate system. In 3D space the coordinates would take form of  $x, y, z$ . This set of points is called a point cloud and it is created by devices such as stereo cameras or laser scanners by scanning some object or scene. Point clouds are useful e.g. for surface or scene reconstruction.

<sup>1</sup>A polygonal mesh is a set of vertices, edges and faces that represent the shape of an object in 3D.

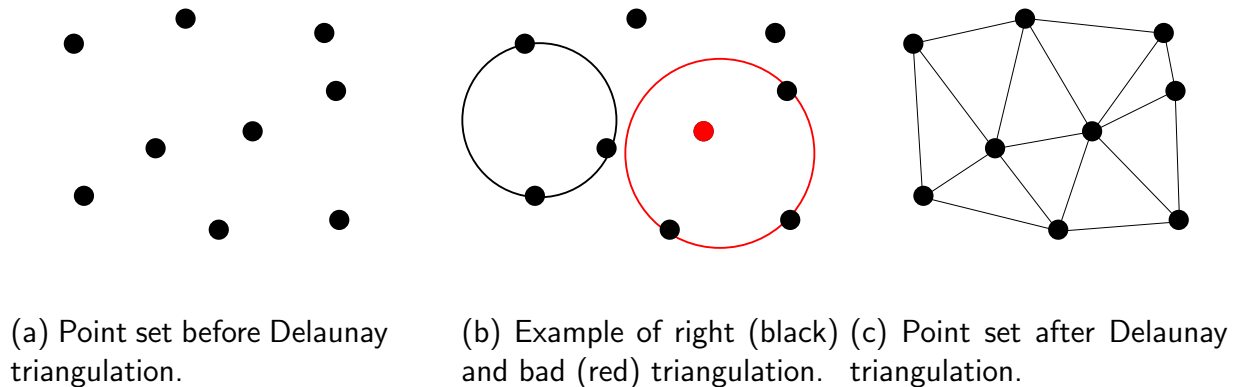


Figure 3.2: Delaunay triangulation.

This paper uses point clouds generated by MS Kinect which stores the data as an image represented by 2D matrix. Each element of the matrix represents distance from particular point of scanned scene to the position of the sensor. This image is called a *depth map*.

Elements in the depth map can be later converted to represent points with coordinates  $x_P$ ,  $y_P$ ,  $z_P$  with respect to the camera frame. After conversion, points are still stored in 2D matrix but each its element contains said coordinates. By this conversion, a form of a point cloud is created.

## 3.4 Point triangulation

MS Kinect creates a depth map that is converted to a point cloud of 3D points with coordinates  $x_P$ ,  $y_P$ ,  $z_P$  stored in a 2D matrix (grid). Every element of this 2D grid therefore represents a unique 3D point in space. The next step is to create a polygonal domain based upon these points. One way of achieving this would be to use Delaunay triangulation. The other possibility is to use advantage of the 2D grid where points are stored. In this paper, the latter option was selected.

### 3.4.1 Delaunay triangulation

Delaunay triangulation is a form of triangulation with some interesting properties e.g. it is dual to Voronoi graph and it maximizes the minimal angle in triangles. Simple description of Delaunay triangulation would be: Triangulation of a set of points in the plane is called Delaunay if and only if the circumcircle of any triangle of triangulation doesn't contain a point in its interior [3]. Figure 3.2 an example of Delaunay triangulation.

### 3.4.2 2D grid method

This approach takes an advantage of points in a point cloud being stored in a 2D grid. Scenario 3 presents the pseudocode of this process. The algorithm assumes input in form of a point cloud. The point cloud is then triangulated and the algorithm outputs this triangulation.

While point cloud is a 2D matrix consisting of points with coordinates  $x_P, y_P, z_P$ , the triangulated point cloud on top of that also contains edges connecting neighboring points. Points from point cloud are copied to a new variable *triangulatedCloud* on line 1. In a for cycle, every point  $u$  is taken from the point cloud and is further processed between lines 2 and 11. On line 3 a neighbor  $v$  of point  $u$  is selected. Next neighboring point  $w$  of  $v$  from perspective of point  $u$  is selected on line 4. Selection of these neighbors is described in Subsection 3.4.3.

The while cycle between lines 5 and 10 runs until  $w$  is the last neighbor of  $u$  that has to be processed. Points  $u, v, w$  are connected to each other by edges on line 6. Then, point  $v$  becomes point  $w$  on line 7. Point  $w$  is temporarily saved to a variable *temp* on line 8 and next neighbor of *temp* is assigned to  $w$  on line 9.

Finally, line 12 returns the triangulated point cloud.

**Input:** point cloud *cloud*

**Output:** triangulated point cloud *triangulatedCloud*

```

1. setPoints(triangulatedCloud, cloud)
2. for all points  $u \in cloud$  do
3.    $v \leftarrow downNeighbor(u)$ 
4.    $w \leftarrow getNextNeighbor(u, v)$ 
5.   while  $w$  isn't last neighbor of  $u$  do
6.     connectPoints(u, v, w)
7.      $v \leftarrow w$ 
8.      $temp \leftarrow w$ 
9.      $w \leftarrow getNextNeighbor(u, temp)$ 
10.  end while
11. end for
12. return triangulatedCloud

```

**Scenario 3:** Pseudocode of point cloud triangulation.

### 3.4.3 Neighbor selection

Every point in a 2D grid has 8 neighboring points but only 3 neighbors are needed for triangulation purposes. These points are located down, right-down and right from the original point (see Figure 3.3). So, the method *downNeighbor(u)* on line 3 in Scenario 3 assigns the neighbor located down from point  $u$  to variable  $v$ .



Figure 3.3: Triangulation of a 2D grid where the red dot represents the original point and the blue dots represent its neighbors during neighbor selection.

Scenario 4 presents the pseudocode of getting a next neighbor of point  $u$  with regard to previously selected neighbor  $v$ . It is basically just if-else command. When  $v$  is a neighbor located down from the original point  $u$  on line 1 then the neighbor located right-down from  $u$  is assigned to  $w$  on line 2. Otherwise, the neighbor located right from  $u$  is assigned to  $w$  on line 4. Finally  $w$  is returned on line 6.

**Input:** current point  $u$ , neighboring point  $v$

**Output:** neighboring point  $w$

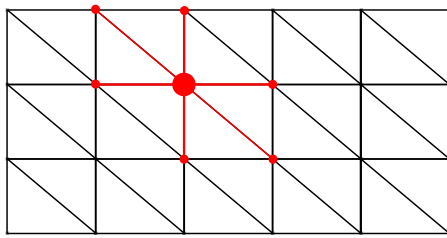
1. **if**  $v$  is down neighbor of  $u$  **then**
2.      $w \leftarrow \text{rightDownNeighbor}(u)$
3. **else**
4.      $w \leftarrow \text{rightNeighbor}(u)$
5. **end if**
6. **return**  $w$

**Scenario 4:** Pseudocode of getNextNeighbor process.

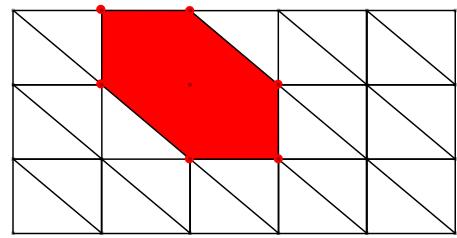
## 3.5 Polygon reduction

A polygonal domain is created after point triangulation described in the previous section. The domain consists of the same points as the original point cloud and polygons made of these points. The polygonal domain can contain hundreds of thousands of points and polygons which has large impact on computational complexity of any further operations. Therefore it is wise to reduce the number of points and polygons. One way of achieving this is to give some meaningful value to every point representing how much a certain point contributes to the shape of the whole polygonal domain. The point with the lowest contribution value is removed along with every polygon this point participated in.

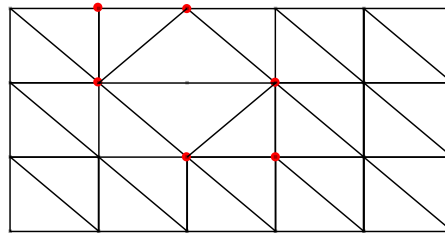
However, removal of this point causes creation of an undesired hole which must be addressed. Therefore points on the boundary of this hole are re-triangulated and the hole is sealed (Figure 3.4). Contribution value of re-triangulated points has changed by sealing the hole, so their



(a) Point with the lowest contribution value and its topological neighbors.



(b) Hole created by point removal. Neighbors of original point lie on the hole's boundary.



(c) Re-triangulated hole.

Figure 3.4: Process of re-triangulation.

contribution values need to be reevaluated. Re-triangulation is again based on 2D grid as described in previous section and uses triangulation of monotone polygons. This process is repeated many times until only the most relevant points remain.

A couple of methods were developed to evaluate the contribution value of these points. These methods are described in Subsections 3.5.2, 3.5.3, 3.5.4 and 3.5.5.

### 3.5.1 Reduction outline

Scenario 5 presents the pseudocode that reduces the number of polygons in a polygonal domain. The algorithm assumes the input in form of a triangulated point cloud and outputs reduced version of this triangulation.

Contribution values are computed from triangulated point cloud  $tria$  on line 1 and are inserted to some kind of priority queue  $Q$ . In this paper, binary heap was used for its computational efficiency. More about binary heaps can be found in Section 5.3. How to compute the contribution value of points is described below this section.

The reduction process itself takes place in a while cycle between lines 2 and 10. Point  $u$  with the lowest contribution value is recovered and deleted from  $Q$  on line 3. Topological neighbors of  $u$  are identified on line 7 and are assigned to a set of points  $P$ . On line 8,  $P$  is then used to re-triangulate the hole created by removal of point  $u$ . On the same line, edge connections between points of  $P$  and  $u$  are updated in  $tria$ . Process of re-triangulation of polygons is

described in Section 3.6. After re-triangulation, contribution values of points in  $P$  has changed, so on line 9 their contribution values are computed once again and  $Q$  is accordingly updated.

The while cycle between lines 2 and 10 terminates if  $Q$  doesn't contain any more points or when contribution value of  $u$  is so great that removal of this point would drastically degrade the shape of the whole polygonal domain. The second condition is located between lines 4 and 6.

When reducing the number of polygons, the contribution value of points on the boundary of the polygonal domain would possibly be the lowest. So, when polygons are reduced, only points inside the 2D grid are evaluated. Points on the boundary of the 2D grid are reduced on line 11 and this process is described in Section 3.7.

Finally, on lines 5 and 12 the triangulation with reduced polygons is returned.

**Input:** triangulated point cloud  $tria$

**Output:** triangulated point cloud with reduced number of polygons  $tria$

1.  $Q \leftarrow initializeContValues(tria)$
2. **while**  $count(Q) > 0$  **do**
3.      $u \leftarrow getPointWithLowestContValue(Q)$
4.     **if**  $contValue(u) > limit$  **then**
5.         **return**  $tria$
6.     **end if**
7.      $P \leftarrow getNeighbors(u)$
8.      $retriangulate(P, u, tria)$
9.      $Q \leftarrow updateContValues(P)$
10. **end while**
11.  $reduceLines(tria)$
12. **return**  $tria$

**Scenario 5:** Pseudocode of a polygon reduction.

### 3.5.2 Centroid method

This method measures the distance between evaluated point and a centroid which is computed from topologically neighboring points of the evaluated point. The distance is used directly as the contribution value.

First, sums have to be computed:

$$x_c = \sum_{i=1}^k x_{n_i}, \quad y_c = \sum_{i=1}^k y_{n_i}, \quad z_c = \sum_{i=1}^k z_{n_i} \quad (3.1)$$

where  $x_c, y_c, z_c$  are coordinates of the centroid  $C$ ,  $x_{n_i}, y_{n_i}, z_{n_i}$  are coordinates of  $i$ -th neighboring point and  $k$  is the number of neighboring points.

$$contValue = \|P - C\| \quad (3.2)$$

where  $P$  is the evaluated point with coordinates  $x_P, y_P, z_P$ . The symbol  $\|\cdot\|$  represents the Euclidean norm.

### 3.5.3 LSQ method

This method is slightly similar to the previous method. Again, distance is directly used as the contribution value of evaluated point. A plane is approximated by neighboring points using the least squares approach. Distance is then measured between the evaluated point and the approximated plane.

First, write the matrix equation  $Ax = b$ , where

$$A = \begin{bmatrix} \sum_{i=1}^k x_{n_i}^2 & \sum_{i=1}^k x_{n_i}y_{n_i} & \sum_{i=1}^k x_{n_i} \\ \sum_{i=1}^k x_{n_i}y_{n_i} & \sum_{i=1}^k y_{n_i}^2 & \sum_{i=1}^k y_{n_i} \\ \sum_{i=1}^k x_{n_i} & \sum_{i=1}^k y_{n_i} & n \end{bmatrix}, \quad b = \begin{bmatrix} \sum_{i=1}^k x_{n_i}z_{n_i} \\ \sum_{i=1}^k y_{n_i}z_{n_i} \\ \sum_{i=1}^k z_{n_i} \end{bmatrix} \quad (3.3)$$

Next, solve for  $x$  which represents coefficients of least square fit plane  $z = ax + by + d$ :

$$x = [a \quad b \quad d]^T \quad (3.4)$$

Now, a concrete value of  $z$  is got by substituting a given point  $P$  to parametric plane equation  $z = ax + by + d$ :

$$z = ax_P + by_P + d \quad (3.5)$$

Usually, there would be an absolute value in Equation 3.5, but used stereo camera (MS Kinect) always generates non-negative  $z$  values.

Finally, the contribution value of point  $P$  is computed:

$$contValue = |z_P - z| \quad (3.6)$$

### 3.5.4 Shape decomposition

Another method comes from [4]. The article deals with shape decomposition based on discrete contour convolution in 2-dimensional space. The idea is to take two consecutive line segments and determine whether their common endpoint can be removed. How is the decision made is explained in Section 3.7 which deals with line reduction.

It could be possible to use this method in 3D environment where the common endpoint of all possible participating polygons should be considered for removal. However, this method wasn't implemented in this thesis.

### 3.5.5 vtkDecimation

The VTK library provides several methods of point/polygon reduction. These methods differ in the way the point reduction is done. For example method `vtkQuadricClustering` uses clustering of vertices to bins and computing quadric error. Their main disadvantage, however, is that they don't guarantee preserving the topology and position of input points [5].

Another point against using VTK algorithms is that they are more general and don't respect the 2D grid representation of points which makes them relatively slow compared to centroid or LSQ methods.

### 3.5.6 Summary

The LSQ method and the centroid method are used in this thesis. Their advantage lies in respecting the 2D grid representation and their simplicity.

## 3.6 Re-triangulation

As was said in Section 3.5, point removal generates a hole as can be seen in Figure 3.4. In 3D, re-triangulation of this hole could be problematic, but in 2D this process becomes a lot easier. Points on the boundary of the hole are still stored in 2D grid and one can take advantage of this additional information as in Section 3.5. In 2D, these points form a single polygons which needs to be re-triangulated. There are several methods of doing so.

### 3.6.1 Ear Clipping

One of the most basic methods of polygon triangulation is ear clipping. A diagonal is found between two vertices which divides the polygon in two. Another diagonal is found by recursion and polygon is divided once again. This goes on and on until there are no more diagonals which divide the polygon [6].

### 3.6.2 Triangulation of monotone polygons

Monotone polygons are one of the easiest polygons to triangulate and the decomposition of more complicated polygons to monotone ones is the basis of many fast triangulation methods. Here are some definitions.



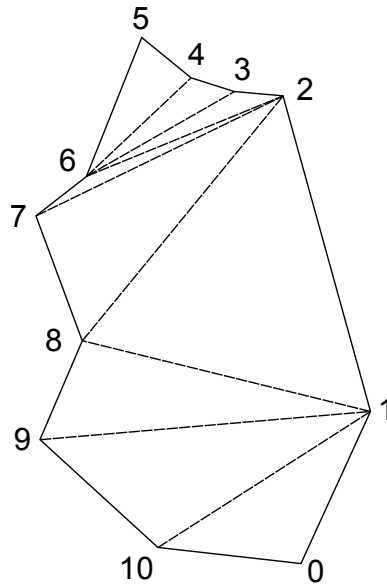


Figure 3.5: Triangulation of a monotone polygon.

**Definition 1** A polygonal chain (or curve)  $C$  is called **monotone polygonal chain** with respect to  $L'$  if every line  $L$  orthogonal to  $L'$  intersects  $C$  in at most one point [6].

**Definition 2** A polygon  $P$  is called **monotone polygon** with respect to a line  $L$  if  $P$  can be split into two monotone polygonal chains  $A$  and  $B$  with respect to  $L$  [6].

For example, a polygon on Figure 3.5 is monotone with respect to the vertical line (such polygon is sometimes also called  $y$ -monotone).

### Triangulation

Scenario 6 presents the pseudocode of triangulation of monotone polygons. The algorithm assumes the input in form of a monotone polygon  $poly$  and outputs diagonals  $diag$  between points from which the triangulation of polygon consists. The algorithm can be easily modified to return a set of triangles if needed.

First, on line 1, the vertices of the monotone polygon  $poly$  are sorted by their  $x$  or  $y$  coordinate (according to the monotonicity of the polygon) and are assigned to a set of vertices  $Q$ . In a for cycle between lines 2 and 10, each point  $u$  of  $Q$  is assigned to a traversed set of points  $P$ .

In a for cycle between lines 3 and 8, each point  $v$  of  $P$  is tested for visibility between vertices  $u$  and  $v$  on line 4. If the vertex  $v$  is visible from  $u$  then, on line 5, a diagonal is created between  $u$  and  $v$  and now unnecessary vertex  $v$  is removed from  $P$  on line 6. Finally, a set of diagonals  $diag$  is removed on line 11.

**Input:** polygon  $poly$   
**Output:** diagonals  $diag$

1.  $Q \leftarrow sortVertices(poly)$
2. **for all** points  $u \in Q$  **do**
3.     **for all** points  $v \in P$  **do**
4.         **if**  $visible(u, v)$  **then**
5.              $diag \leftarrow makeDiag(u, v)$
6.              $P \leftarrow P \setminus \{v\}$
7.         **end if**
8.     **end for**
9.      $P \leftarrow u$
10. **end for**
11. **return**  $diag$

**Scenario 6:** Triangulation of a monotone polygon.

### Example

To present a concrete example, assume that the algorithm is currently processing vertex no. 6 in Figure 3.5. The set of points  $P$  contains vertices 5, 4, 3, 2. Although vertex 5 is technically visible from 6 a diagonal is not considered to be added as they both lie on the same line segment. However, vertices 4, 3 and 2 are visible from 6 and diagonals are added to  $diag$ .

### Simplification

The Scenario 6 can be simplified in the sense, that no visibility check is required. If one can identify monotone chains  $A$  and  $B$  then it can be seen, that vertices of  $A$  are connected to  $B$  and vice versa. Vertices on one chain are never connected to any vertex on the same chain. Therefore, visibility test on line 4 can be replaced by condition testing whether the vertices lie on the opposite chains.

In example on Figure 3.5, one chain is made from vertices 5, 6, 7, 8, 9, 10, 0 and the other is made from vertices 5, 4, 3, 2, 1, 0.

### 3.6.3 Trapezoidalization

A technique called trapezoidalization is used to decompose general polygons into monotone ones so they can be easier triangulated. Trapezoidalization is done by using a sweep line running across the polygon which stops at its vertices. The vertices represent events which are evaluated. If a vertex has a clear line of sight on both sides of a polygon, then the vertex is called an interior cusp. Cusps are further divided to upward cusps and downward cusps. Upward cusp is connected to opposing vertex in the upper trapezoid. Downward cusp is connected to opposing vertex in

the lower trapezoid. For more information on trapezoidalization see [6]. Figure 3.6 represents decomposition of a polygon to monotone polygons by trapezoidalization.

The decomposition of polygons to monotone polygons by doing trapezoidalization is used in this thesis.

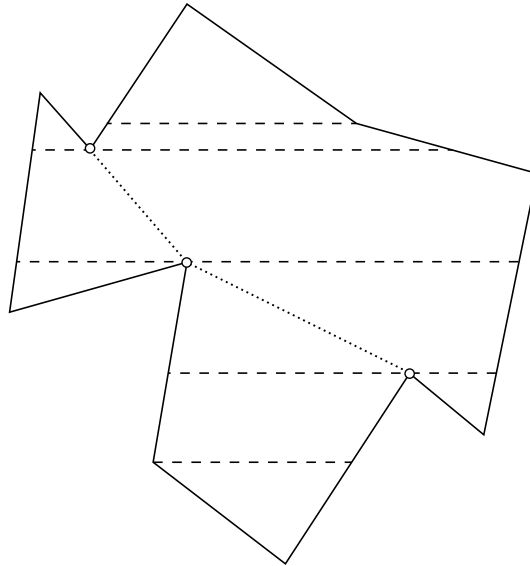
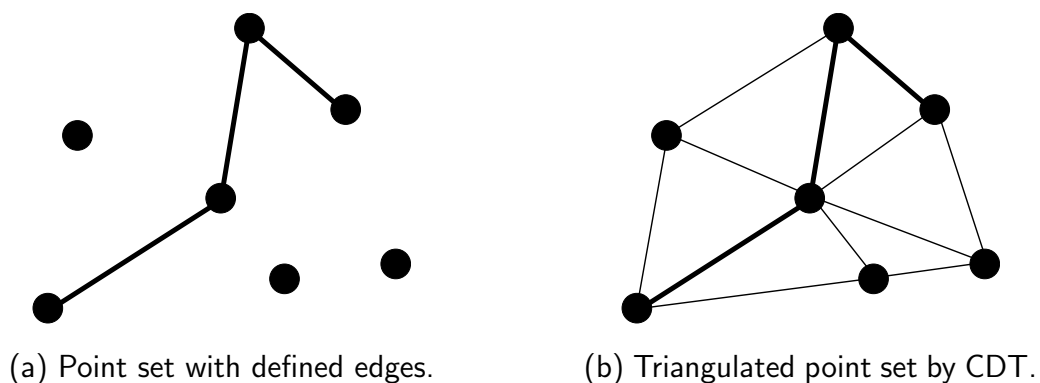


Figure 3.6: Trapezoidalization of a polygon.

### 3.6.4 Constrained Delaunay triangulation

Constrained Delaunay triangulation is a type of triangulation which takes into account edges between input vertices and preserves them. At the same time, it is trying to be as similar as possible to the classic Delaunay triangulation described in Subsection 3.4.1 but often fails to do so [7]. Example of this triangulation can be seen on Figure 3.7.



(a) Point set with defined edges.

(b) Triangulated point set by CDT.

Figure 3.7: Constrained Delaunay triangulation or CDT.

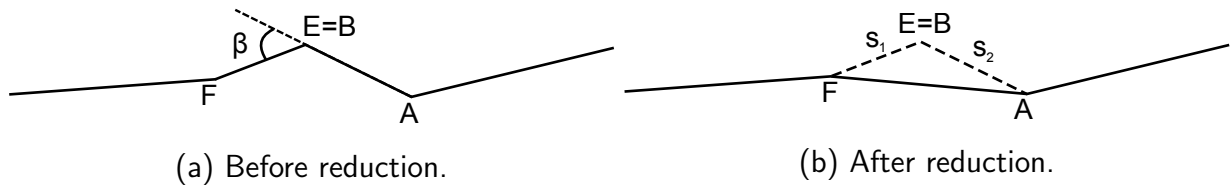


Figure 3.8: Line reduction.

### 3.6.5 Summary

The method of trapezoidalization of polygons into monotone polygons and their following triangulation is used in this thesis. Although the constrained Delaunay triangulation might give visually better triangulation of polygons, its disadvantage lies in higher computational complexity compared to trapezoidalization/triangulation of monotone polygons.

## 3.7 Line reduction

As was said in Subsection 3.5.1, the contribution value of points on the boundary of the polygonal domain would be the lowest among all points. Therefore, the reduction of the boundary points takes place in a separate process.

The algorithm of line reduction is very similar to Scenario 5 with two exceptions. There is no need for step on line 11 anymore and a different method of computing the contribution value of points is used on lines 1 and 9. This method is described right below in Subsection 3.7.1.

### 3.7.1 Contribution value

Calculation of the contribution value for reduction of boundary points is based on article [4]. Two topological neighbors of point  $P$  lying on the boundary of topological domain are selected. The contribution value is then calculated like:

$$\text{contValue} = \frac{\beta(s_1, s_2)l(s_1)l(s_2)}{l(s_1) + l(s_2)} \quad (3.7)$$

where  $s_1 = AB$  and  $s_2 = EF$  are two consecutive line segments of curve  $C$  and  $B = E$  is their common endpoint (see Figure 3.8).  $\beta(s_1, s_2)$  is the turn angle, i.e.  $\beta(s_1, s_2) = \text{angle}(EF) - \text{angle}(AB)$ .  $l(s_1)$  and  $l(s_2)$  represents the length of line segments  $s_1$  and  $s_2$ .

For more information about the reasoning behind this calculation, the reader is encouraged to read article [4].

## 3.8 Creation of a polyhedron

After reducing the number of polygons the process of creating the polyhedron can begin. Scenario 7 presents the pseudocode of creation of the polyhedron. The algorithm assumes the input in form of a triangulated point cloud *tria* and outputs a polyhedron *poly*.

Polygons from *tria* are transferred to *poly* on line 1. Then, on line 2, the origin *O* is added to the polyhedron, without connecting it by edges to any other point. Coordinates of the origin in 3D space usually take form of  $O = (0, 0, 0)$ . Boundary points *P* are identified on line 3. First point of *P* is assigned to *firstP* and *prevP* on lines 4 and 5, so it is known what point was first taken into account. Boundary neighbor of *firstP* is assigned to variable *v* on line 6.

In a while cycle between lines 7 and 12 every two neighboring boundary points *prevP* and *v* are connected to origin *O*, thus creating a new polygon in *poly* on line 8. On line 10 a new boundary neighbor is found and the process repeats until *v* takes form of the first point of *P*.

On line 13, the final polygon is created and the algorithm returns the polyhedron *poly* on line 14.

Figure 3.9 shows an example of a triangulated point cloud being converted into a polyhedron.

**Input:** triangulated point cloud *tria*

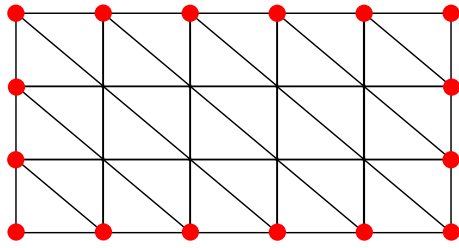
**Output:** polyhedron *poly*

1.  $poly \leftarrow getPointsAndPolygons(tria)$
2.  $O \leftarrow addOrigin(poly)$
3.  $P \leftarrow getBoundaryPoints(tria)$
4.  $firstP \leftarrow getPoint(P)$
5.  $prevP \leftarrow firstP$
6.  $v \leftarrow getOneOfBoundaryNeighbor(firstP)$
7. **while**  $v \neq firstP$  **do**
8.      $createPolygon(prevP, v, O, poly)$
9.      $temp \leftarrow v$
10.     $v \leftarrow getOneOfBoundaryNeighbor(v, prevP)$
11.     $prevP \leftarrow temp$
12. **end while**
13.  $connectToOrigin(prevP, v, poly)$
14. **return** *poly*

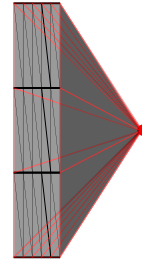
**Scenario 7:** Creation of a polyhedron.

## 3.9 Frames transformation

This section deals with the transformation of coordinates of a point in polyhedron from camera's coordinate system (or frame) to world's coordinate system. This is necessary for further



(a) Triangulated point cloud in which red dots represent boundary points.



(b) Created polyhedron. Image is slightly rotated to allow polygons connecting boundary points to  $O$  to be better visible.

Figure 3.9: Polyhedron.

work with polyhedrons which includes union of multiple polyhedrons described in Section 5.6 and path planning described in Chapter 4.

As was said earlier, the coordinates of points in the point cloud and the polyhedron are relative to a coordinate system of the stereo camera which recorded the scene. To pass these coordinates from one frame to another, rotation matrix and translation vector are needed. Let  $C$  be a camera frame and  $W$  be a world frame. To express location of point  $P$  captured by the camera (frame  $C$ ) in world frame  $W$ , translation from  $W$ 's origin to the  $C$ 's origin needs to be applied first. Then, a rotation of  $C$ 's coordinate axis in  $W$  is applied. Equation 3.8 represents this transformation:

$$P_W = T_{C_W} + R_{C_W} P_C \quad (3.8)$$

where  $P_W$  is a point  $P$  in world's coordinate system,  $P_C$  is a point  $P$  in camera's coordinate system,  $T_{C_W}$  is a translation vector of  $C$  in world's coordinate system and  $R_{C_W}$  is a rotation matrix of  $C$  in world's coordinate system.

How to transform points between different frames in ROS environment is described in Chapter 5.

# Chapter 4

## From polyhedron to path

This chapter deals with path planning on a polyhedron. This is useful for exploring an unknown 3-dimensional environment which requires navigation through explored parts of the world to frontiers lying between known and unknown areas.

### 4.1 Chapter outline

Scenario 8 presents the pseudocode that constructs a path from some starting point to one of goals called frontiers. The algorithm assumes the input in form of a polyhedron and outputs a path to the frontier.

Frontiers are identified on line 1. For more about frontiers see Section 4.2. On line 2, a dual graph of polyhedron is constructed to allow easier path construction. The process of constructing the dual graph is described in Section 4.3. Line 3 presents an incorporation of frontiers into dual graph of the polyhedron, this process is described in Section 4.4. On line 4, the path is planned from starting position (e.g. position of the exploring robot) to one of the frontiers. Section 4.5 contains more information about path planning.

**Input:** polyhedron *poly*

**Output:** *path*

1. *frontiers*  $\leftarrow$  *getFrontiers(poly)*
2. *dualGraph*  $\leftarrow$  *createDualGraph(poly)*
3. *addFrontiers(dualGraph, frontiers)*
4. *path*  $\leftarrow$  *planPath(dualGraph)*
5. **return** *path*

**Scenario 8:** Pseudocode of path planning.

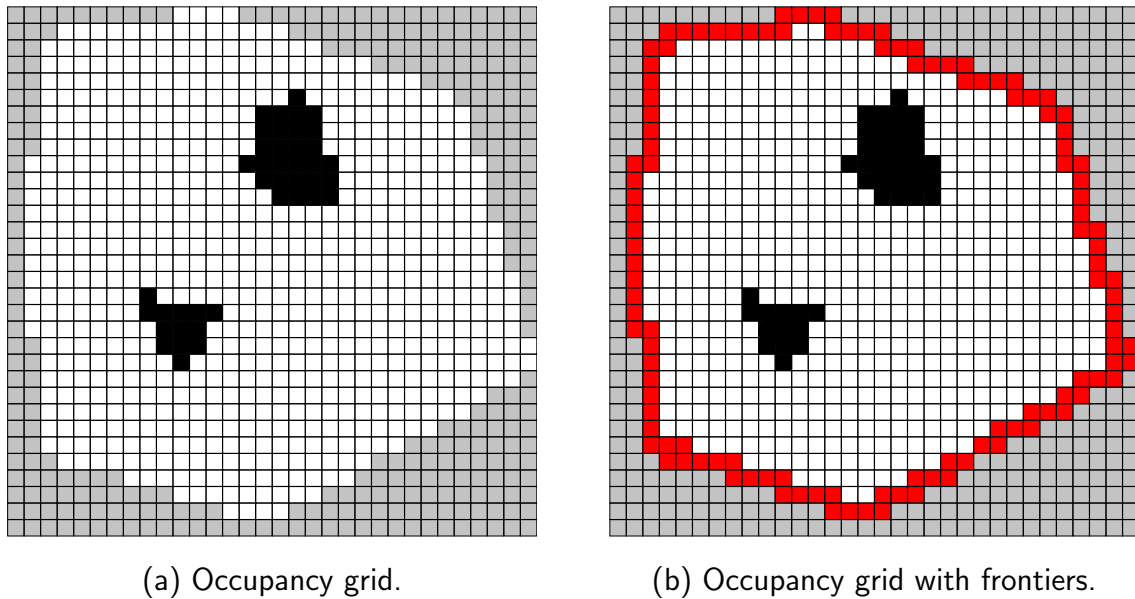


Figure 4.1: Occupancy grid where black elements represent obstacles, white elements represent free space and gray elements represent unknown environment.

## 4.2 Frontiers

During the exploration of an unknown environment the exploring robot should know which place would be best to explore next. The robot chooses this location from a collection of places called frontiers, plans a path to it and goes exploring it.

For better understanding, frontiers in 2D are described first, then follows a description of frontiers in 3D.

### 4.2.1 Frontiers in occupancy grid

Occupancy grid is a form of world representation; every element of the grid indicates, with certain probability, that the corresponding space is either an obstacle or a free space. There is also a third state which is associated with an unknown space. An example of the occupancy grid can be seen on Figure 4.1.

Frontiers are elements separating unknown space from free space. They also indicate that beyond these elements lies something more worth investigating. In exploration, frontiers are regarded as points of interest. So, these are the points to where the path is planned [1].



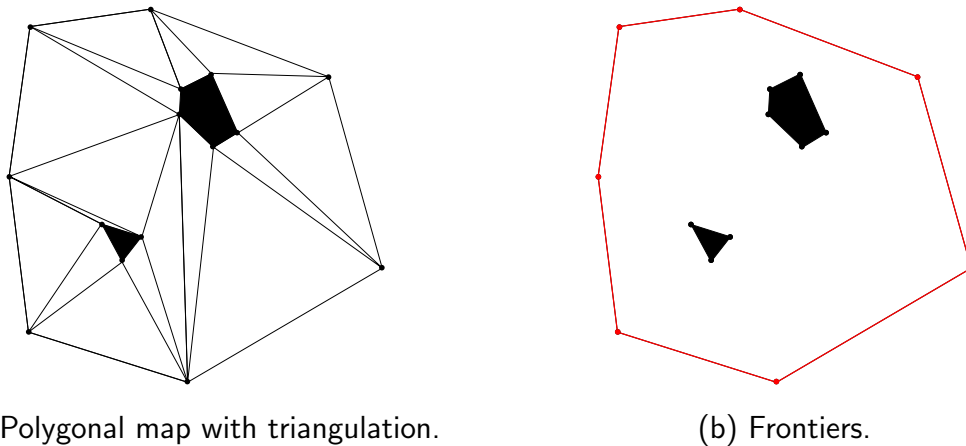


Figure 4.2: Polygonal domain where black segments represent triangulation of explored environment with obstacles, red segments represent frontiers.

### 4.2.2 Frontiers in polygonal domain

The concept of frontiers was outlined in the previous subsection. Now, this concept is extended to work on polygonal domain which is constructed by doing triangulation of points. The process of triangulation was described in Chapter 3.

One can view a polygonal domain as a map consisting of a set of polygons (Figure 4.2). Free space is represented by inner space of polygons, obstacles are represented by "holes" in polygons and unknown space is anything else. If a union of these polygons was computed, one big polygon would be acquired and its outer edges would represent the frontiers.

### 4.2.3 Frontiers in polyhedral domain

The only difference between frontiers in 2D with regard to polygonal domain and frontiers in 3D with regard to polyhedral domain is that in 2D they are represented by polygon's edges while in 3D they are represented by the faces of a polyhedron.

## 4.3 Dual graph

Planning a path on an occupancy grid is relatively easy. Every element of the grid represents a vertex which is connected to its neighbors. However, on polygonal or polyhedral domain, things are a bit different.

The inner side of polygons contains an infinite number of points, therefore the path cannot be planned across them. The solution is to pick one point of the polygon that represents it. Dual graphs can serve well for this type of conversion.

Dual graphs are first described on a polygonal domain, before changing focus to the polyhedral case.

### 4.3.1 Polygonal domain

#### Voronoi diagram

Voronoi diagram is a dual graph to Delaunay triangulation. To better describe the Voronoi diagram one can imagine solving a problem known as the post office problem.

Let  $S$  be a set of  $n$  points representing  $n$  post offices. When an arbitrary new point (e.g. a residence) is picked it is desirable to know which post office is closest to the new point. Voronoi diagram determines the locus of points for each point  $n$  called Voronoi region  $V(p)$  that are closer to  $n$  than any other point (Figure 4.3).

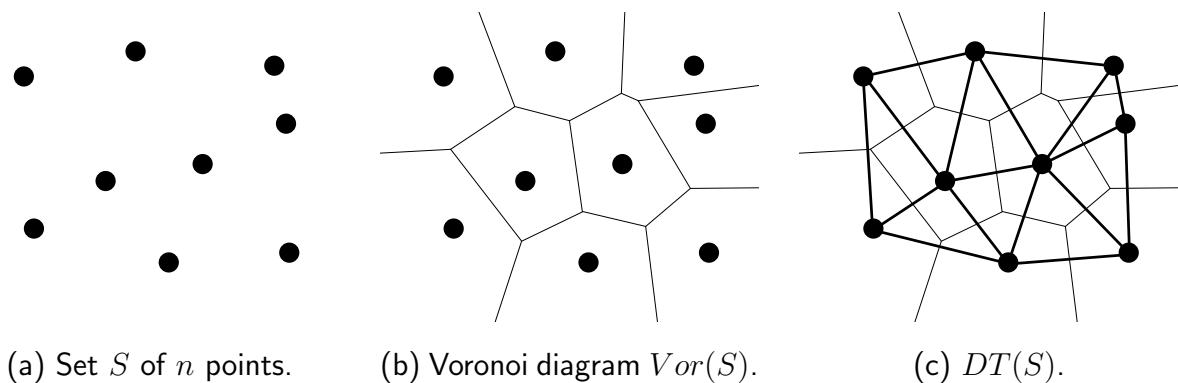
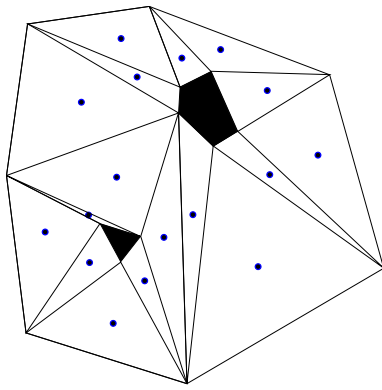


Figure 4.3: Voronoi diagram and its connection to Delaunay triangulation.

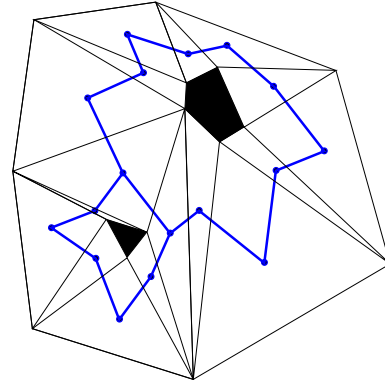
The following properties are stated to show that the Voronoi diagram is indeed dual to Delaunay triangulation ( $DT(s)$ ):

- If  $p$  is the nearest neighbor of  $q$  then Voronoi regions  $V(p)$  and  $V(q)$  are adjacent. This implicates that  $p$  and  $q$  are connected by edge in  $DT(s)$
- It was already said in Chapter 3, that points  $p, q, r$  form a triangle in  $DT(s)$  iff the circle through  $p, q, r$  has no points in its interior

For better description of Voronoi diagram and its connection to Delaunay triangulation see [8] or [6].



(a) Polygons with their centers of gravity.



(b) Edges connecting centers of gravity represent the dual graph.

Figure 4.4: Dual graph without frontiers where black segments represent triangulation of explored environment with obstacles, blue dots represent centers of gravity of polygons and blue lines represent connections between adjacent polygons in dual graph.

### Dual graph to other forms of triangulation

Now to the case, when a set of points  $S$  isn't triangulated by Delaunay triangulation. It can be instead triangulated e.g. by a method described in Subsection 3.4.2. Luckily, some kind of dual graph to this type of triangulation still exists.

In this paper, the center of gravity of polygons (created by triangulation) is computed as a mean to construct the dual graph. In case that polygon has the form of a triangle it can be computed like this:

$$x_c = \frac{x_{p_1} + x_{p_2} + x_{p_3}}{3}, \quad y_c = \frac{y_{p_1} + y_{p_2} + y_{p_3}}{3} \quad (4.1)$$

where  $C = (x_c, y_c)$  is the center of gravity of the polygon,  $P_1 = (x_{p_1}, y_{p_1})$ ,  $P_2 = (x_{p_2}, y_{p_2})$  and  $P_3 = (x_{p_3}, y_{p_3})$  are the points forming the polygon.

Next, the adjacency of the polygons is identified so edges could be created in the dual graph. When two polygons share an edge, a new edge in dual graph is created by connecting centers of gravity of these two polygons. By doing so for every polygon, the dual graph of triangulation is constructed (see Figure 4.4).

### 4.3.2 Polyhedral domain

Back to a 3-dimensional space and a polyhedron. A polyhedron was constructed from initial set of points in Chapter 3. As in 2D case on polygonal domain, it is necessary to plan a path from some starting position to one of several frontiers to properly explore the environment.

The problem is, that the interior of the polyhedron is empty, therefore many vertices are potentially missing to plan the shortest path and construction of the dual graph directly from polyhedron is impossible. One way of filling the interior is to use a method called constrained Delaunay tetrahedralization. Then, the dual graph can be finally constructed.

### Constrained Delaunay tetrahedralization

In Subsection 3.6.4, a basic description of constrained Delaunay triangulation was provided. The method called constrained Delaunay tetrahedralization is similar, except it deals with 3D objects. This method is used to generate a mesh. The mesh is later converted to a dual graph upon which the planning algorithm is used.

To briefly outline formal aspects of constrained Delaunay tetrahedralization, let  $X$  be a polyhedron with vertices and faces. A tetrahedralization  $T$  is called constrained tetrahedralization of  $X$  if  $T$  and  $X$  have the same vertices and the tetrahedra in  $T$  cover the triangulation domain. Triangulation domain is the region of space being tetrahedralized. For more information about constrained Delaunay tetrahedralization see [9].

	CD triangulation	CD tetrahedralization
<b>Preserves (constrains)</b>	edges	faces
<b>Creates triangle/tetrahedron if no vertex lies in a</b>	circumcircle	circumsphere

Table 4.1: Quick comparison between constrained Delaunay (CD) triangulation and CD tetrahedralization. Constrained versions of Delaunay triangulation/tetrahedralization sometimes break the second property in favor of the first property.

### Dual graph in a polyhedral domain

After obtaining a tetrahedralization of a polyhedron, a dual graph in a polyhedral domain can be constructed. Similarly to 2-dimensional case, centers of gravity of tetrahedrons are computed.

$$x_c = \frac{x_{p_1} + x_{p_2} + x_{p_3} + x_{p_4}}{4}, \quad y_c = \frac{y_{p_1} + y_{p_2} + y_{p_3} + y_{p_4}}{4}, \quad z_c = \frac{z_{p_1} + z_{p_2} + z_{p_3} + z_{p_4}}{4} \quad (4.2)$$

where  $C = (x_c, y_c, z_c)$  is the center of gravity of the tetrahedron,  $P_1 = (x_{p_1}, y_{p_1}, z_{p_1})$ ,  $P_2 = (x_{p_2}, y_{p_2}, z_{p_2})$ ,  $P_3 = (x_{p_3}, y_{p_3}, z_{p_3})$  and  $P_4 = (x_{p_4}, y_{p_4}, z_{p_4})$  are the points forming the tetrahedron.

After identifying the adjacency of tetrahedrons, the dual graph can be constructed by connecting the centers of gravity of all adjacent tetrahedrons.

## 4.4 Frontiers in a dual graph

Previous section described the process of converting the polygons in a polygonal domain and polyhedrons in a polyhedral domain to a dual graph. Unfortunately, there is no information about frontiers in the dual graph after this conversion. This section describes the process of adding the frontiers to the dual graph, which are regarded as goals in path planning.

This section is, again, explained first on the polygonal case for easier description. Then, the process of adding the frontiers to the dual graph on a polyhedral domain is explained.

### 4.4.1 Polygonal domain

Frontiers in a polygonal domain are represented by lines bounded by two points. To add frontiers to the dual graph, the polygon in which the particular frontier is contained needs to be identified first. Then, the frontier's center is computed:

$$x_f = \frac{x_{p_1} + x_{p_2}}{2}, \quad y_f = \frac{y_{p_1} + y_{p_2}}{2} \quad (4.3)$$

where  $F = (x_f, y_f)$  are coordinates of frontier's center,  $P_1 = (x_{p_1}, y_{p_1})$  and  $P_2 = (x_{p_2}, y_{p_2})$  are the points defining the frontier.

Next, frontier's line center  $F$  is connected to the center of gravity  $C$  of the particular polygon. It is desirable to also connect  $F$  with neighboring frontiers. This can be easily accomplished since it is known which points are used in which frontiers. The whole conversion can be seen on Figure 4.5.

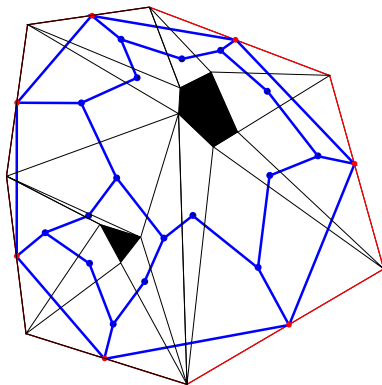
### 4.4.2 Polyhedral domain

The process of adding frontiers to the dual graph in a polyhedral domain is, again, very similar to the polygonal case. The exception is that in this case, frontiers are represented by faces of a polyhedron. To proceed, the centers of gravity of frontiers (i.e. triangle faces) are computed:

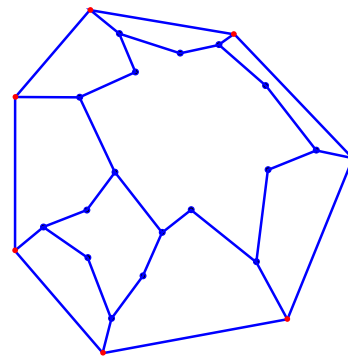
$$x_f = \frac{x_{p_1} + x_{p_2} + x_{p_3}}{3}, \quad y_f = \frac{y_{p_1} + y_{p_2} + y_{p_3}}{3}, \quad z_f = \frac{z_{p_1} + z_{p_2} + z_{p_3}}{3} \quad (4.4)$$

where  $F = (x_f, y_f, z_f)$  is the center of gravity of the frontier,  $P_1 = (x_{p_1}, y_{p_1}, z_{p_1})$ ,  $P_2 = (x_{p_2}, y_{p_2}, z_{p_2})$  and  $P_3 = (x_{p_3}, y_{p_3}, z_{p_3})$  are the points defining the frontier.

Then, the frontiers center's  $F$  are connected to center's  $C$  of tetrahedrons where did the frontiers previously belong to. Next, the adjacency of frontiers to other frontiers is determined and their  $F$ 's are connected.



(a) Centers of frontiers connected to centers of gravity of relevant polygons.



(b) The resulting dual graph containing frontiers.

Figure 4.5: Dual graph with frontiers where black segments represent triangulation of explored environment with obstacles, blue dots represent centers of gravity of polygons, blue lines represent connections between adjacent polygons in dual graph, red lines represent frontiers and red dots represent center of gravity of frontiers.

Although this approach may seem sufficient for every polygon, it may not hold true for very large ones. It is better to represent such big polygons by multiple points spread across its surface. This work, however, doesn't focus on this particular topic.

## 4.5 Path planning

Path planning is one of the fundamental problems in robotics. A 2D map can be seen as a graph made of vertices that are connected by edges. The path planning problem deals with finding the shortest path from a start vertex to a goal vertex.

Some of path planning method are described in following paragraphs.

### 4.5.1 Shortest path

The term shortest or optimal path should be defined before describing various methods of graph planning.

Let  $G$  be a graph with a set of vertices  $V$  and set of weighted edges  $E$ . The shortest path is such path that connects two vertices in a way which minimizes the sum of weights of edges needed to connect these two vertices.

### 4.5.2 Breadth-first search

This method uses FIFO (first-in first-out) queue to select vertices to be processed. First, only neighboring vertices of *start* are added to the queue, other vertices follow when the FIFO queue starts to be traversed. This causes the graph to be searched uniformly or to the breadth first, hence the name of the algorithm.

The algorithm ensures it delivers the path from a *start* to a *goal* using the least number of steps. On the other hand it is not guaranteed that the path will be optimal [10].

### 4.5.3 Depth-first search

In comparison to breadth-first search, this method uses LIFO (last-in, first-out) queue to select vertices to be processed. This is quite aggressive method, as it dives quickly into the graph and prefers to traverse longer routes first.

Although this algorithm may find the path from a *start* to a *goal* very quickly it is also not guaranteed to find the optimal path.

### 4.5.4 Dijkstra search algorithm

Dijkstra designed a graph search algorithm for solving the shortest path problem for a graph with nonnegative edge weights.

Let  $G$  be a graph with a set of vertices  $V$  and a set of weighted edges  $E$ . First step is to assign  $\infty$  to  $g(v)$  of every vertex. The cost value  $g(v)$  represents the lowest cost of getting from vertex *start* to vertex  $v$ . Therefore the value  $g(\textit{start})$  is then set to 0. Next, a set of vertices  $V$  is assigned to set  $Q$ . In a while, vertex  $v$  with the lowest value  $g(v)$  is taken from set  $Q$ . Simultaneously  $v$  is removed from set  $Q$  to mark  $v$  as a visited vertex.

The while loop continues until every vertex in  $Q$  is traversed. In a for cycle every neighbor  $u$  of vertex  $v$  is processed. If the cost value  $g(v)$  is greater than the summed value  $g(u)$  and the distance between vertices  $u$  and  $v$ , then the summed value is assigned to  $g(u)$ .

As mentioned, the algorithm stops when every vertex is processed (which implies that the path to the *goal* has not been found) or when currently processed vertex is the *goal*. When this happens, it is desirable to reconstruct the shortest path from *start* to *goal* consisting only of necessary vertices. Previous vertex is assigned to every processed vertex for this purpose. By doing this, every vertex "knows" which way to go when heading to the *start* [11].

The pseudocode of Dijkstra search algorithm can be seen in Scenario 9.

#### Path reconstruction

The shortest path can be reconstructed by recursion, as can be seen in Scenario 10. In the beginning, there is a set of vertices *path* containing only vertex *goal*. In a while loop, previous

**Input:** *start*, *goal*, set of nodes  $V$

**Output:** *path*

1. **for all**  $v \in V$  **do**
2.      $g(v) \leftarrow \infty$
3.      $prev(v) \leftarrow undefined$
4. **end for**
5.  $g(start) \leftarrow 0$
6.  $Q \leftarrow V$
7. **while**  $Q \neq \emptyset$  **do**
8.      $u \leftarrow \min(g(\forall v \in Q))$
9.     **if**  $u = goal$  **then**
10.          $path \leftarrow getPath(u)$
11.         **return**  $path$
12.     **end if**
13.     **for all**  $v \in neighbors(u)$  **do**
14.         **if**  $g(v) > g(u) + dist\_between(u, v)$  **then**
15.              $g(v) \leftarrow g(u) + dist\_between(u, v)$
16.              $prev(v) \leftarrow u$
17.         **end if**
18.     **end for**
19. **end while**

**Scenario 9:** Pseudocode of Dijkstra search algorithm.

vertex of *goal* is added to *path*. Simultaneously, this previous vertex is set as a currently processed one. The while loop continues until the vertex *start* is reached (which has undefined previous vertex). After this, the algorithm ends.

**Input:** *goal*

**Output:** *path*

1.  $u \leftarrow goal$
2.  $path \leftarrow u$
3. **while**  $prev(u) \neq undefined$  **do**
4.      $path \leftarrow path \cup prev(u)$
5.      $u \leftarrow prev(u)$
6. **end while**
7. **return**  $path$

**Scenario 10:** Pseudocode of path reconstruction.



### 4.5.5 A\* search algorithm

The A\* search algorithm is a search algorithm that relies on heuristic estimate  $h$  representing the cost of getting from vertex  $v$  to *goal*. The heuristic estimate is used to lower the number of visited nodes.

As the word "estimate" suggests, the true value of getting from vertex  $v$  to *goal* is not known but it can be easily approximated. For example, if graph vertices have coordinates  $(x, y)$ , Manhattan or Euclidean distances between vertices and *goal* can be computed. These types of heuristics underestimate the true cost of getting from  $v$  to *goal*. Otherwise the algorithm couldn't guarantee to find the shortest possible path.

The difference between A\* search algorithm and Dijkstra search algorithm is in the computation of the cost value  $f$ . In Dijkstra, the cost function  $g$  represented the cost of getting from vertex  $v$  to *start*. In A\*, the cost function  $f$  consists of the same value  $g$  plus heuristic estimate  $h$ . If heuristics is set to zero  $h = 0$  in A\* then it would imitate the function of Dijkstra search algorithm. The pseudocode of A\* can be seen in Scenario 11.

As in Dijkstra search algorithm, the A\* terminates when every vertex has been processed (traversed) or when *goal* has been found. The shortest path is obtained by the algorithm described in Section 4.5.4 [10].

### 4.5.6 Planning to multiple frontiers (goals)

As might be noticed, the start vertex represents e.g. a position of the robot and frontiers represent goal vertices.

In this paper, when a path is planned by an algorithm described in Section 4.5 it is planned from start vertex to the nearest or best accessible frontier. In other words e.g. in A\* search algorithm it is not checked whether the currently processed vertex  $u$  is a *goal*, but it is checked whether it is a frontier. By doing so, the world is explored step by step as the robot move from one frontier to another.

**Input:** *start, goal*, set of nodes  $V$

**Output:** *path*

1. **for all**  $v \in V$  **do**
2.      $prev(v) \leftarrow undefined$
3. **end for**
4.  $closedset \leftarrow \emptyset$
5.  $openset \leftarrow start$
6.  $g(start) \leftarrow 0$
7.  $f(start) \leftarrow g(start) + h(start, goal)$
8. **while**  $openset \neq \emptyset$  **do**
9.      $u \leftarrow \min(f(\forall v \in openset))$
10.    **if**  $u = goal$  **then**
11.        $path \leftarrow getPath(u)$
12.       **return**  $path$
13.    **end if**
14.    **for all**  $v \in neighbors(u)$  **do**
15.        $g_{new} \leftarrow g(u) + dist\_between(u, v)$
16.        $f_{new} \leftarrow g_{new} + h(v, goal)$
17.       **if**  $v \notin closedset \wedge v \notin openset$  **then**
18.            $g(v) = g_{new}$
19.            $f(v) = f_{new}$
20.            $prev(v) = u$
21.            $openset \leftarrow openset \cup v$
22.       **else if**  $f_{new} < f(v)$  **then**
23.            $g(v) = g_{new}$
24.            $f(v) = f_{new}$
25.            $prev(v) = u$
26.       **end if**
27.    **end for**
28.     $closedset \leftarrow openset \cup u$
29.     $openset \leftarrow openset \setminus \{u\}$
30. **end while**

**Scenario 11:** Pseudocode of A\* search algorithm.

# Chapter 5

## Framework

This chapter describes the implementation details of the created framework. The framework was written in C++ language and it uses ROS environment [12] to provide additional functionalities, such as convenient data manipulation and communication between various framework parts.

ROS environment is described in Section 5.1. Section 5.2 talks about used 3rd party libraries which were utilized. Before moving to deeper details of the implementation, a binary heap data structure is described in Section 5.3 as it is used in many parts of the framework. Details of point triangulation are described in Section 5.4. A look inside of polygon reduction is provided by Section 5.5. Section 5.6 describes the implementation details of polyhedral operations, such as union of two polyhedrons and their transformation between coordinate systems.

### 5.1 ROS

The Robot Operating System or ROS is an open-source collection of tools and libraries. It aims to simplify creation and usage of robot oriented software. ROS supports various languages like C++, Python and others. Processes in ROS are represented by nodes and they serve as basic elements in ROS.

Communication between nodes in ROS is done by sending and receiving messages. Node can send messages under topics (in ROS called rostopics) that serve as broadcasters so other nodes listening to this topic can receive sent message. This behavior is used in this paper to receive messages from a sensor containing actual point cloud data, transformation matrix and several others [12].

## 5.2 Used 3rd party libraries

This section introduces various 3rd party libraries used for easier development of the whole framework. Some of these libraries were slightly altered and reasons for this are given at appropriate sections.

### VTK

Visualization Toolkit (VTK) is an open-source library with focus on scientific data visualization, computer graphics and image processing. Among other languages it supports C++ and python. It provides capabilities such as Delaunay triangulation or polygon reduction which were used in this work for comparison with similar methods. VTK was also used as the main visualization interface [5].

### Polypartition

Polypartition is a C++ library for polygon triangulation which consists of several methods of triangulation. It requires input polygons not to be self-intersecting and in counter-clockwise order [13].

PolyPartition library is heavily used for hole re-triangulation after point removal which occurs during process of polygon reduction. This is described in Chapter 3 and here in Section 5.5.

### Carve

Carve is a C++ library for boolean operations on polygonal meshes including polyhedrons. Carve uses a modeling technique called CSG or Constructive Solid Geometry which is a technique by which complex objects can be created. In this paper, Carve's CSG is used to create a world map by computing a union of two polyhedrons.

As said, Carve implements boolean operations such as union, intersection and symmetric and asymmetric difference. It also retains object's properties like texture during the CSG operations. Carve's ability to retain object's properties is used to easily preserve information about frontiers [14].

Carve is used during union of polyhedrons which is described in Section 5.6.

### TetGen

TetGen is a C++ library for generating tetrahedral meshes on polyhedral domain. It is used in this work for constrained Delaunay tetrahedralization which is useful for path planning algorithms in 3D. Same as Carve, TetGen also retains object's properties during tetrahedralization by interpolation and therefore the information about frontiers isn't lost [15].

	Insert	Delete	Update <sup>2</sup>
<b>Average</b>	$O(1)$	$O(\log n)$	$O(1)$
<b>Worst case</b>	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 5.1: Time complexity of a binary heap.

Deeper insight into usage of TetGen is provided by Section 5.7.3.

## 5.3 Binary heap

Several methods used in this work rely on fast extraction of nodes ordered by their value (e.g. extraction of points with the lowest contribution value for polygon reduction). Binary heap is a form of a tree data structure which provides just that functionality.

Two types of binary heap exist: max-heap or min-heap. Every node has two children, with the exception of the lowest nodes, and every node has one parent, again with exception of the top most node. When talking about max-heap, every parent has greater or equal value than its children and every child has lower or equal value than its parent. This is called the heap property and it ensures that a node with the greatest value is always on top of the heap.

By maintaining the heap property one can achieve very fast node manipulation [16].

For time complexity of operation on binary heap see Figure 5.1.

## 5.4 Point triangulation

When working with point clouds in this paper, the 3D points are stored in 2D grid. Every element of this a 2D grid represents a unique 3D point in space. These points are triangulated in a way described in Section 3.4. There are several problems which need to be taken care of, such as missing data or duplicate points.

### 5.4.1 Duplicate data

Some point clouds acquired by a stereo camera may contain points with duplicate values. For example 500 points can have the same  $x$ ,  $y$  and  $z$  coordinates. Such point clouds are discarded as unusable in this thesis. The other option would be to identify their position in the 2D grid and treat them like points with missing values. This procedure is described in following subsection.

---

<sup>2</sup>The general algorithm of binary heap is slightly altered to support quick search for particular node. This ability is useful for updating the node's value. After updating it must be ensured that the heap property is maintained by doing similar operations as in insertion or deletion of a node.

### 5.4.2 Missing data

Point clouds can also contain points with missing values. These points may represent either a point lying far beyond sensor range or it is a sign of an object with surface of unusual reflectance.

In both cases missing data are treated the same as being faraway points. With knowledge of intrinsic parameters of used stereo camera the position of this point can be recalculated. This procedure has only one caveat and that is it has to be given the desired distance of this point:

$$x_P = \frac{x - c_x}{f_x} d, \quad y_P = \frac{y - c_y}{f_y} d, \quad z_P = d \quad (5.1)$$

where  $x_P, y_P, z_P$  are desired coordinates of the missing point,  $x, y$  are coordinates of point stored in 2D grid,  $d$  is the desired distance from the camera,  $c_x, c_y$  is the optical center of the camera and  $f_x, f_y$  are the focal lengths of the camera.

The default value of the parameter  $d$  is set to be 8 meters but it can be set to any other value as a command line parameter. Values of parameters  $c_x, c_y$  and  $f_x, f_y$  are loaded from `rostopic /CameraInfo`.

### Cropping

Some point clouds have large amount of missing data on the boundaries of the grid. Approximate coordinates of these elements can be computed by using procedure described above in this subsection or the point cloud can just be cropped and unnecessary data can be thrown away. Cropping of the point cloud brings one advantage – saving some of the computational time without losing any relevant information.

How many elements of the grid are to be cropped can be set as a command line parameter (see Appendix B for a list of command line switches).

### Ordered vertices

Every point is contained in several polygons after triangulation. When a point is removed during polygon reduction, a hole is created. This hole is represented by a new polygon that consists of points to which was connected the original point. To quickly retrieve this polygonal hole it is useful to have neighbors of the original point ordered in clockwise or counter-clockwise fashion.

For this purpose, a binary heap data structure is used for every 3D point. The heap contains neighboring points ordered by an angle which is calculated between every neighbor and the original point.

As in Chapter 3, things can be simplified by taking advantage of the 2D grid and calculate the angle in 2D space. Sorting takes place during triangulation and it is used during other phases of the algorithm, such as the re-triangulation part.

## 5.5 Polygon reduction

Polygon reduction is used to lower the number of polygons in a polygonal mesh. The process of reduction is explained in Section 3.5 and some of its more important implementation details are listed below.

### 5.5.1 Binary heap

The process of polygon reduction after point cloud triangulation consists of removing a point which has the least impact on the shape of the whole polygonal domain. This impact is represented by contribution value which assigns a number to every point of the cloud that represents how much does certain point influence the polygonal shape. Means to calculating the contribution value are given in Section 3.5.

It is desirable to remove points with the lowest contribution value first, therefore a binary heap data structure is used to quickly retrieve stored contribution values of points.

### 5.5.2 Re-triangulation

Re-triangulation is used to triangulate "leftover" points after point removal. For this purpose the PolyPartition library is used. It provides means to triangulate points using several different methods. Summary of these methods is listed in Table 5.2.

Method called "Triangulation by partition into monotone polygons" is used to re-triangulate created hole after point removal. Although PolyPartition library provides visually better looking methods of polygon triangulation than the used one, the method "Triangulation by partition into monotone polygons" has the lowest time complexity among other methods. This is its huge advantage because there is a need to re-triangulate polygon hundred thousands times during processing of one point cloud which is very time consuming.

## 5.6 Polyhedral operations

Polyhedron is created after polygon reduction and it serves as a 3D representation of an explored part of the world. How to create a polyhedron from a point cloud is described in Section 3.8.

Polyhedrons are created to allow construction of a world map. To do so, the union of several polyhedrons has to take place alongside with transformation of points in point clouds from sensor's coordinate system to world's coordinate system.

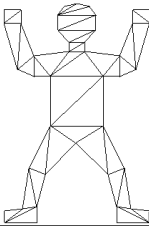
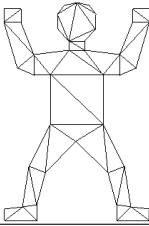
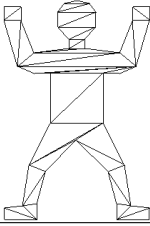
	<b>Time complexity</b>	<b>Space complexity</b>	<b>Example Image</b>
<b>Triangulation by ear clipping</b>	$O(n^2)$	$O(n)$	
<b>Optimal triangulation in terms of edge length using dynamic programming algorithm</b>	$O(n^3)$	$O(n^2)$	
<b>Triangulation by partition into monotone polygons</b>	$O(n \cdot \log(n))$	$O(n)$	

Table 5.2: Summary of triangulation methods provided by PolyPartition library (taken from [13]).

### 5.6.1 Polyhedron transformations between frames

Transformation between frames is required to allow points to be transformed from sensor's coordinates to world's coordinates which is needed to properly create a world map.

Transformation in ROS environment is done by using `rostopic /tf` which contains information about various coordinate frames. It also provides means to easily transform points from one coordinate system to another.

### 5.6.2 Union of polyhedrons

The union of polyhedrons (or any object, in that matter) is done by using Constructive Solid Geometry (CSG) which was briefly described in Subsection 5.2. Union, intersection and others are boolean operations that allow modeling of complex objects.

In this paper, union of polyhedrons was accomplished by using a C++ library called Carve. Carve is designed to perform boolean operations between polygonal meshes. It is also great in a sense that it allows polygons in meshes to keep their original "color property". This is useful for further processing as frontier and non-frontier polygons can still be easily distinguished.



## 5.7 Planner

This section provides an insight into programmatic part of path planning on polyhedral map. Theoretical part can be found in Chapter 4.

### 5.7.1 Frontier identification in polyhedron

Frontiers are an important part of path planning in robotics. They represent locations in the world map that the robot should explore. More about frontiers can be found in Section 4.2. The identification of frontiers in a polyhedron can be divided into two parts.

#### Frontiers in point cloud

Frontiers in point cloud are points which coordinates are initially missing. Their coordinates are later computed as is shown in Section 5.4.2. These points are marked as frontiers before the computation of their coordinates. After point triangulation in Section 5.4 and polygon reduction in Section 5.5, every polygon that contains any of the frontier points is also regarded as frontier.

The frontier points are no longer important, so in the end only frontiers in form of polygons are further considered.

#### Frontiers during polyhedron creation

Identification of frontiers created during construction of a polyhedron is easy. Every new polygon that is created during construction of polyhedron is regarded as a frontier.

### 5.7.2 Path planning

The A\* path planning algorithm was used to plan a path on a dual graph of tetrahedralized polyhedron. The A\* algorithm is described in Subsection 4.5.5.

In the early development phases, only one goal vertex was considered, so the heuristics parameter  $h$  was set to represent Euclidean distance to goal from every vertex. After considering multiple goals by adding the concept of frontiers,  $h$  was set to 0, otherwise there would be a problem of determining the closest frontier to each vertex.

Although it might make sense to use Dijkstra's path planning algorithm instead of the A\* algorithm, the latter was kept for possibility of future its extension to plan a path to any location on the acquired map. This can be useful e.g. for collision avoidance.

### 5.7.3 Tetrahedralization

A polyhedron has to be tetrahedralized before proceeding to planning a path to frontier as described in Subsection 4.3.2. For this purpose, C++ library TetGen was used. The behavior of TetGen can be adjusted by using several of console-like switches. In this paper, switch "pznnfY" was used. Feel free to read TetGen documentation for a description of these switches [15].

# Chapter 6

## Experiments

This chapter aims to provide an insight into how efficiently the implemented algorithms work and how does the created world map look like. Section 6.1 deals with quality of polygon reduction while Section 6.2 presents execution times of polygon reduction. Section 6.3 deals with creation of a world map of an unknown environment. Discussion on accomplished results is provided in Section 6.4

### 6.1 Quality of polygon reduction

Two experiments have been realized to test the quality of polygonal domain after performing polygon reduction. ROS bag files usable in the ROS environment containing sequence of point clouds upon which is the polygon reduction executed come from [17].

#### 6.1.1 Scenes

Point cloud from dataset "freiburg3\_structure\_notexture\_near" provided by [17] was used for the first experiment. Figure 6.1 represents a triangulated scene on which was the polygon reduction executed. Comparison of both implemented methods of polygon reduction (LSQ and centroid method) can be seen on Figure 6.2 during various stages of reduction. The experiment was set up with following parameters: the image was cropped by 50 px and missing data were recomputed so they would lie in the distance of 8 meters.

Point cloud from dataset "freiburg3\_long\_office\_household" provided by [17] was used for the second experiment. Figure 6.3 again represents a triangulated scene on which was the polygon reduction executed. Comparison of both implemented methods of polygon reduction can be seen on Figure 6.4 during various stages of reduction. The experiment was set up with following parameters: the image was cropped by 100 px and missing data were recomputed so they would lie in the distance of 3 meters

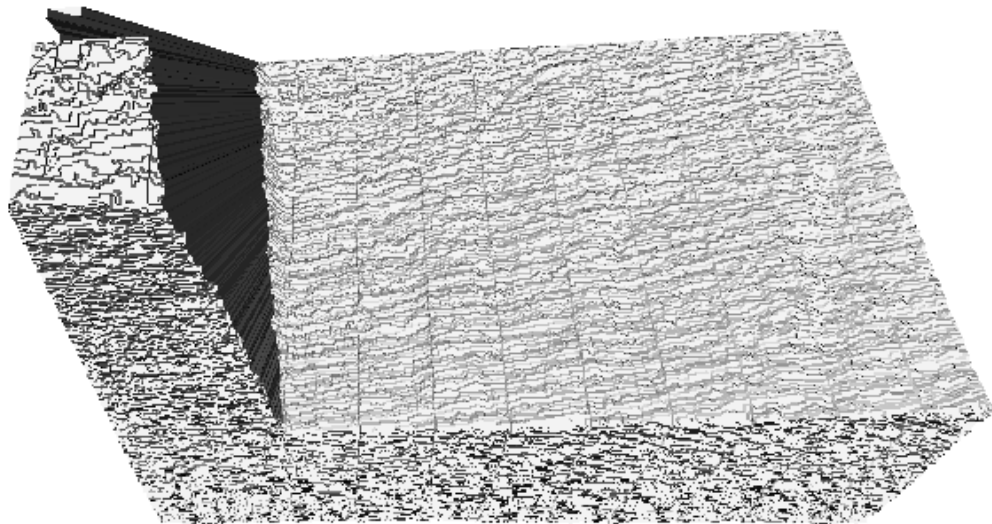
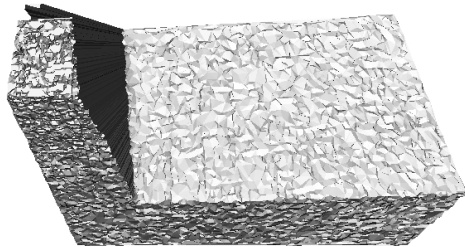
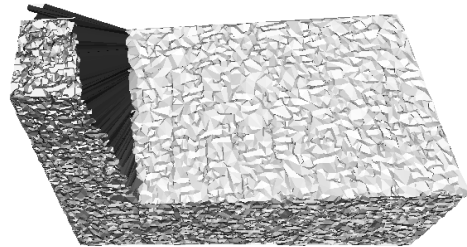


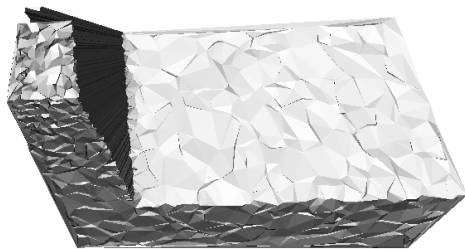
Figure 6.1: Scene 01 with 410400 polygons and 206121 points.



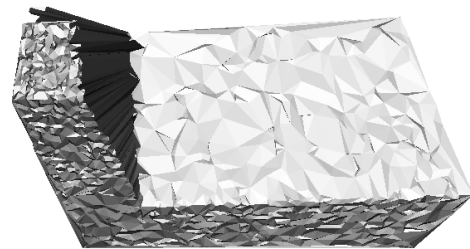
(a) Centroid Method (20518/10306).



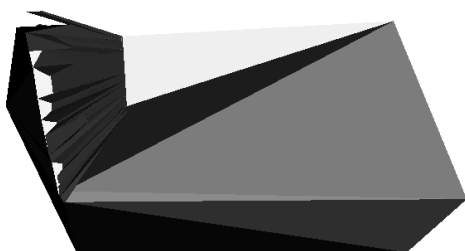
(b) LSQ Method (20518/10306).



(c) Centroid Method (4100/2060).



(d) LSQ Method (4100/2060).



(e) Centroid Method (222/113).



(f) LSQ Method (222/113).

Figure 6.2: Comparison of two polygonal reduction methods on Scene 01. Numbers in brackets represents number of polygons/points contained in the scene.

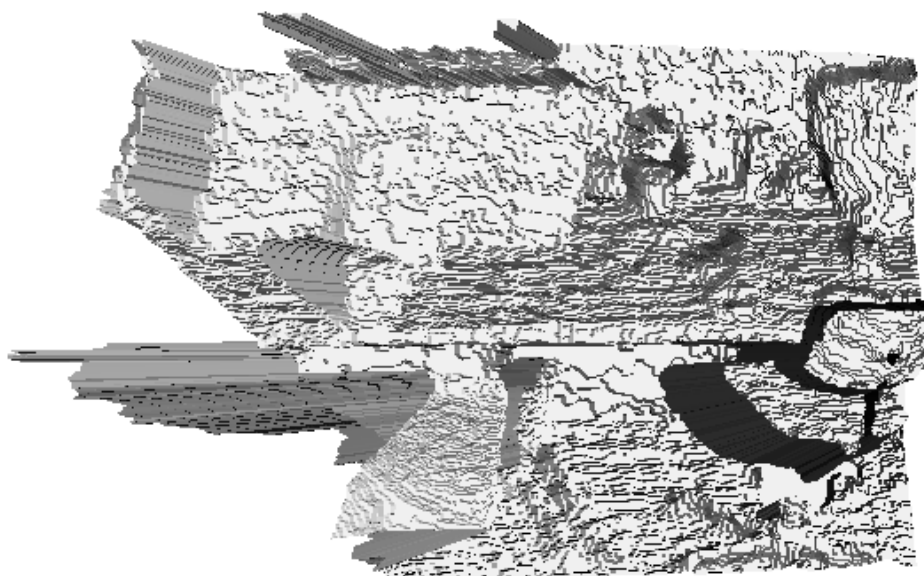
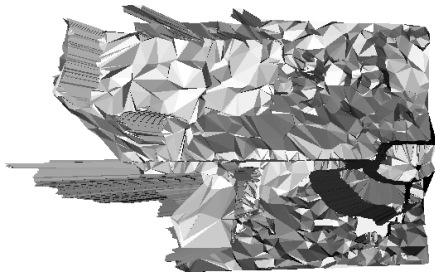
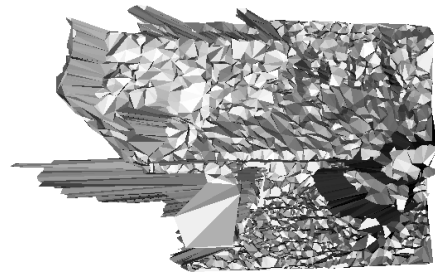


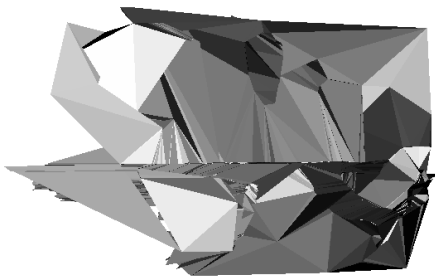
Figure 6.3: Scene 02 with 246400 polygons and 123921 points.



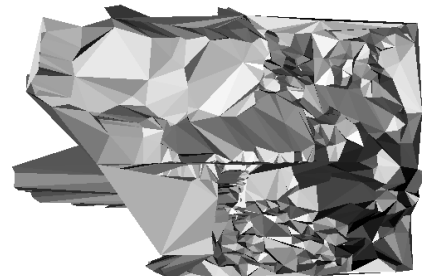
(a) Centroid Method (12318/6196).



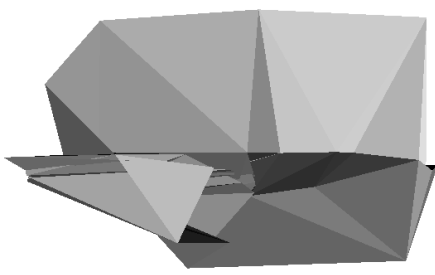
(b) LSQ Method (12318/6196).



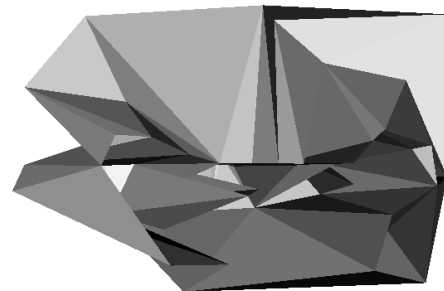
(c) Centroid Method (2457/1238).



(d) LSQ Method (2457/1238).



(e) Centroid Method (129/71).



(f) LSQ Method (129/71).

Figure 6.4: Comparison of two polygonal reduction methods on Scene 02. Numbers in brackets represents number of polygons/points contained in the scene.

	Centroid method	LSQ method
Time [s]	$1.458 \cdot 10^1$	$1.445 \cdot 10^1$

Table 6.1: Execution times of polygon reduction methods.

## 6.2 Execution time of polygon reduction

Execution times were measured for both implemented polygon reduction methods – LSQ and centroid. Measurements are represented by Table 6.1.

The experiment was made on a hardware with an Inter(R) Core(TM) i7-3630QM processor at 2.40GHz x4, 4GB RAM running 64-bit Linux Ubuntu 13.04, ROS version Groovy. Each experiment was run 10 times and the average was computed from all runs. Point cloud from dataset "freiburg3\_structure\_notexture\_near" was used without any initial cropping and missing data were recomputed so they would lie in the distance of 8 meters. Initial point cloud contained 307200 points and 6112162 polygons. Reduced point cloud contained 10 points and 8 polygons.

## 6.3 Quality of map creation

Point clouds used for map creation of an unknown environment again come from dataset "freiburg3\_long\_office\_household". The LSQ method of polygon reduction was applied on every point cloud before making a union of them. Every triangulated point cloud after polygon reduction contains 1294 polygons and 684 points.

Figure 6.5 represents used scenes after polygon reduction. Figure 6.6 contains front and back views of created world map with frontiers after incorporating particular scene to the world map.

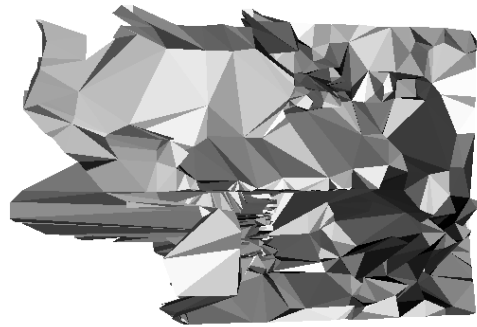
## 6.4 Discussion

Quality of implemented polygon reduction methods was tested on two point clouds. Although it might seem that the LSQ method dominates above the centroid method, it can be noticed that the LSQ method compared to centroid method degrades the quality during the early phases of polygon reduction. This is especially visible along sharp edges. This behavior is a result of the nature of the least square methods.

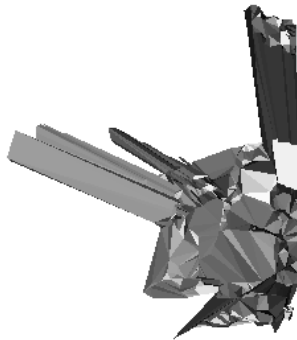
There is not clear winner between these two reduction methods. When it is desired to obtain a triangulated point cloud with low amount of polygon reduction, it is wise to choose the centroid method for the reduction. When it is desired to obtain only a contour of a scene using high amount of polygon reduction, then the LSQ method should be used.

Execution times were measured for both implemented polygon reduction methods – LSQ and centroid. Both methods behave almost the same when comparing their execution times.

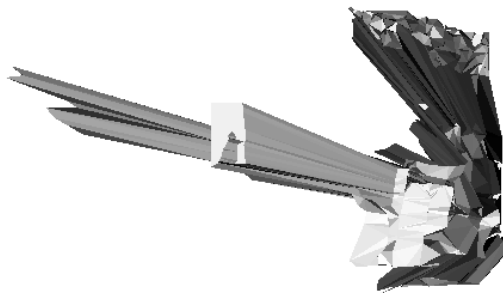




(a) Scene 01.

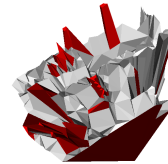
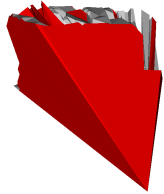


(a) Scene 02.



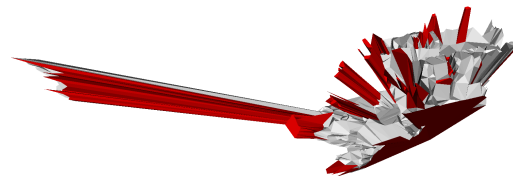
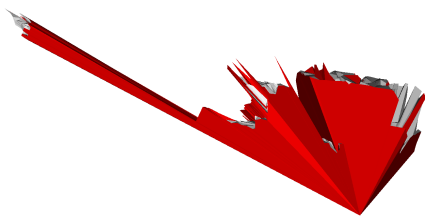
(a) Scene 03.

Figure 6.5: Scenes used for map creation after polygon reduction using LSQ method. Each scene contains 1294 polygons and 684 points.



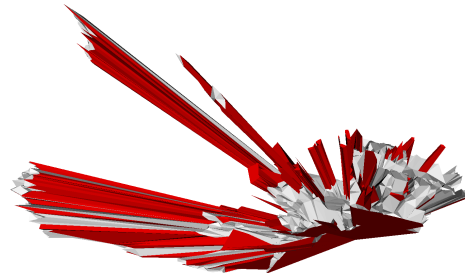
(a) Front view of world map with Scene 01.

(b) Back view of world map with Scene 01.



(c) Front view of world map with Scene 02.

(d) Back view of world map with Scene 02.



(e) Front view of world map with Scene 03.

(f) Back view of world map with Scene 03.

Figure 6.6: Procedural world map creation of an unknown environment where red polygons represent frontiers and white polygons represent obstacles.

The main "bottleneck" seems to be in the part that deals with re-triangulation although one of the fastest polygon triangulation algorithms was used. The execution time of re-triangulation process is more than 1/3 of the whole execution time. On the other hand, when reducing the number of polygons from several thousands to 10, one may expect longer execution times. Cropping the point cloud which is described in Section 5.4 can be considered to speed up the algorithm.

After making union of multiple point clouds for the purpose of creating a world map of an unknown environment, there exist a large number of points and polygons in a polyhedron which represents the map. This isn't a big problem when working with a small number of point clouds. It can be a problem, however, when working with hundreds or thousands of point clouds. Big number of points and polygons in a map could lead to slow computational speed. The solution is to use high amount of polygon reduction on original point clouds. The other way could be to make a polygon reduction on a polyhedron. Either way, both solutions can potentially degrade the quality of a world map.

# Chapter 7

## Conclusion

This thesis took Yamauchi's and Juchelka's approach of exploring an unknown environment in a polygonal domain using frontiers described in [1, 2] and took it to a polyhedral domain. Chapter 2 outlined the problem of exploration in 3-dimensional environment. The process of converting a point cloud to a polyhedron was described in Chapter 3. Chapter 4 dealt with path planning on a polyhedral domain while Chapter 5 gave implementation details on created framework which was written in ROS environment using C++ language. Finally, Chapter 6 presents made experiments.

The development in the ROS environment was chosen to ensure the possibility of future extension of this thesis to incorporate functionality of working with real robots. Several 3rd party libraries were used for the framework to properly function. The libraries took care of union of polyhedrons for the purpose of map creation and tetrahedralization which was used to plan a path on the polyhedral map.

There's no way to tell which of the implemented reduction methods is better. They all depend on trade off between quality of a reduced polygonal domain and a desired number of polygons in the domain.

The map creation of an unknown 3D environment using polyhedrons proved to be a viable solution to 3D exploration problem. It can for example represent large room using very few points and polygons as well as table with books and cups. The main drawback of this method is in its dependency on accuracy of points in the provided point cloud and the quality of polygon reduction.

In the future, this thesis can be improved by development of a polygon reduction method that does better job in preservation of the shape of the polygonal domain. Another issue that should be addressed is time complexity of the polygon reduction method. The quality and usability of the framework could get better by improving said features.

The implemented components of the exploration framework have not been currently integrated into the exploration framework and tested with a real robot or in a simulator. This integration and testing were not possible mainly due to time needed to get acquainted with

functionalities of a robotic simulator like Gazebo or V-Rep. However, the exploration capabilities were successfully tested on a real world dataset. These tests showed that the implemented modules of the exploration framework are ready to be integrated into the overall exploration framework. This integration is a subject of a further work.

## Bibliography

- [1] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings of the IEEE International Symposium on Computational Intelligence, Robotics and Automation*, pages 146–151, 1997.
- [2] T. Juchelka. Exploration algorithms in a polygonal domain. Master's thesis, Czech Technical University in Prague, 2013.
- [3] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.
- [4] L. J. Latecki and R. Lakämper. Convexity rule for shape decomposition based on discrete contour evolution. *Computer Vision and Image Understanding*, 73(3):441–454, 1999.
- [5] Inc. Kitware. Visualization toolkit (vtk). <http://www.vtk.org/>, April 2014.
- [6] J. O'Rourke. *Computational geometry in C - 2nd ed.* Cambridge University Press, 1998.
- [7] L. Paul Chew. Constrained Delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.
- [8] R. Graham and F. Yao. A whirlwind tour of computational geometry. *American Mathematical Monthly*, 97(8):687–701, 1990.
- [9] J. R. Shewchuk. Constrained delaunay tetrahedralizations and provably good boundary recovery. In *Eleventh International Meshing Roundtable (Ithaca, New York)*, pages 193–204, September 2002.
- [10] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.
- [11] D. Joyner, M. V. Nguyen, and N. Cohen. *Algorithmic Graph Theory*. 0.7-r1843 edition, May 2011.
- [12] Robot operating system. <http://www.ros.org/>, April 2014.
- [13] I. Fratric. Polypartition. <https://code.google.com/p/polypartition/>, April 2014.
- [14] T. Sargeant. Polypartition. <https://code.google.com/p/carve/>, April 2014.

- [15] Weierstrass Institute for Applied Analysis and Stochastics. Tetgen. <http://wias-berlin.de/software/tetgen/>, April 2014.
- [16] S. S. Sane, N. A. Deshpande, and A. A. Puntambekar. *Data Structures And Algorithms*. Technical Publications, 2006.
- [17] Technische Universität München. Computer vision group. <http://vision.in.tum.de/>, April 2014.

# Appendices



# Appendix A

## CD Content

The names of all root directories on CD are listed in Table A.1.

<b>Directory name</b>	<b>Description</b>
pdf	Directory containing a digital copy of this thesis
source	Source files of the framework

Table A.1: CD Content.

## Appendix B

### Command line switches

Command line switches of the framework are listed in Table B.1.

Switch	Description
-t	Use TetGen for path planning
-c	Crop image by $x$ pixels
-o	Don't connect points to origin; don't create a polyhedron
-f	Set distance of frontiers to $x$ mm
-m	Set method of polygon reduction – 0 for LSQ method or 1 for Centroid method

Table B.1: Command line switches.