Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Visual 3D Terrain Mapping by a Robotic Helicopter

Tomáš Tichý

Supervisor: Ing. Jan Chudoba
May 2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Tichý Tomáš**	Personal ID number:	**474617**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Cybernetics and Robotics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Visual 3D Terrain Mapping by a Robotic Helicopter

Bachelor’s thesis title in Czech:

Vizuální 3D mapování terénu robotickou helikoptérou

Guidelines:

The thesis topic aims to application of visual 3D terrain mapping „structure from motion“ methods for purpose of robotic multi-rotor helicopter navigation. The result of the work should be a system able to
• create map of unknown environment in a convenient representation,
• geo-reference the map to global coordinate system (GNSS provided position) and
• plan safe low-altitude trajectory to an arbitrary chosen destination position.
Existing available methods may be used for 3D map reconstruction. The results will be experimentally evaluated either in simulated or real environment.
Detailed tasks include
• do a research of existing available 3D mapping methods and choose suitable candidates for implementation,
• make yourself familiar with basics of multi-rotor UAV navigation,
• make a design of the mapping and navigation planning system, and do the necessary implementations,
• experimentally evaluate system performance and precision in a simulator or using real helicopter data.

Bibliography / sources:

[1] J. Zienkiewicz, A. Tsiotsios, A. Davison and S. Leutenegger, "Monocular, Real-Time Surface Reconstruction Using Dynamic Level of Detail," 2016 Fourth International Conference on 3D Vision (3DV), Stanford, CA, 2016, pp. 37-46, doi: 10.1109/3DV.2016.82.
[2] L. Doitsidis, A. Renzaglia, S. Weiss, E. Kosmatopoulos, D. Scaramuzza and R. Siegwart, "3D surveillance coverage using maps extracted by a monocular SLAM algorithm," 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Francisco, CA, 2011, pp. 1661-1667, doi: 10.1109/IROS.2011.6094460.
[3] T. Suzuki, Y. Amano, T. Hashizume, and <. Suzuki, “3D Terrain Reconstruction by Small Unmanned Aerial Vehicle Using SIFT-Based Monocular SLAM,” J. Robot. Mechatron., Vol.23, No.2, pp. 292-301, 2011.

Name and workplace of bachelor’s thesis supervisor:

Ing. Jan Chudoba, Intelligent and Mobile Robotics, CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **15.09.2020** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **19.02.2022**

Ing. Jan Chudoba	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank my supervisor Ing. Jan Chudoba for his patience and indispensable advising. I am also grateful for the support my parents, along with my significant other provided throughout my whole studies.

Declaration

Abstract

The subject of this thesis is the visual 3D reconstruction of terrain using a robotic helicopter for the purpose of navigation. Publicly available programs are used to recover 3D data from images generated with the AirSim simulator. The data is processed into a more convenient 2D heightmap and occupancy grid. The navigation is executed by the Theta* algorithm. Subsequently, an experiment is conducted to compare three open-source photogrammetry programs. The best candidate is used to construct a navigation system, whose performance is experimentally evaluated with satisfactory results.

Keywords: photogrammetry, terrain mapping, UAV navigation, AirSim

Supervisor: Ing. Jan Chudoba
CIIRC,
Jugoslávských partyzánů 3,
Praha 6

Abstrakt

Předmětem této práce je vizuální 3D rekonstrukce terénu s využitím robotické helikoptéry za účelem navigace. Jsou použity veřejně dostupné programy k získání 3D dat z fotografií vygenerovaných za pomocí AirSim simulátoru. Tato data jsou následně zpracována do vhodnější reprezentace pro navigaci v podobě 2D výškové mapy a mřížky obsazenosti. Navigace je provedena pomocí algoritmu Theta*. Dále jsou porovnány tři veřejně dostupné implementace fotogrammetrie. Vybraný program je použit pro sestavení navigačního systému, jehož účinnost je experimentálně ověřena s uspokojivými výsledky.

Klíčová slova: fotogrammetrie, mapování terénu, navigace UAV, AirSim

Překlad názvu: Vizuální 3D mapování terénu robotickou helikoptérou

Contents

1 Introduction **1**
1.1 Context 1
1.2 Specification 2

2 The theory behind photogrammetry **3**
2.1 Mathematical apparatus 3
2.1.1 Notation 3
2.1.2 Transformation of coordinates 3
2.1.3 Homogeneous coordinates 4
2.1.4 Vector product 5
2.1.5 Convolution on matrices 5
2.2 Modeling the camera 5
2.2.1 Pinhole camera model 6
2.2.2 Normalized image coordinates 8
2.2.3 Lens distortion 9
2.3 Depth estimation 10
2.3.1 Stereo depth estimation 10
2.3.2 Triangulation 11
2.4 Challenges 12
2.4.1 Camera calibration 12
2.4.2 Correspondence problem 12
2.4.3 Pose estimation 14
2.5 Photogrammetry pipeline 14
2.5.1 Structure-from-Motion 15
2.5.2 Multi-view Stereo 15

3 Geographic coordinate systems **17**
3.1 ECEF 17
3.2 WGS 17
3.3 ENU 17

4 Used software **19**
4.1 External libraries 19
4.1.1 OpenDroneMap 19
4.1.2 openMVG + openMVS 20
4.1.3 COLMAP 21
4.2 Simulator 21
4.2.1 AirSim 22
4.2.2 Unreal Engine 23

5 Implementation **27**
5.1 AirSim interface 27
5.1.1 airsim_env.py 27
5.1.2 Point cloud from AirSim 29
5.1.3 Simulating lens distortion . . . 30
5.2 Photogrammetry data processing 30
5.2.1 COLMAP 31
5.2.2 OpenDroneMap 31
5.2.3 openMVG + openMVS 32
5.3 Preprocessing for path planning 32
5.3.1 Heightmap generation 32
5.3.2 Occupancy grid generation . . 33
5.4 Path planning 35
5.4.1 Theta* 36

6 Experiments **39**
6.1 Comparison of methods 39
6.1.1 Experiment without distortion 40
6.1.2 Experiment with distortion . . 45
6.1.3 Comparison discussion 49
6.2 Final navigation system 49
6.2.1 Performance evaluation 50

7 Conclusions **53**
7.1 Summary 53
7.2 Future work 54

A Bibliography **55**

B DVD content **57**

Figures

2.1 Coordinate frame transformation . . . 4
2.2 Pinhole camera at the origin 6
2.3 General pinhole camera 8
2.4 Stereo depth estimation 11
2.5 Epipolar geometry 14

3.1 Geographic coordinate systems . 18

4.1 AirSim simulation 22

5.1 Point cloud from AirSim 30
5.2 Heightmap and its grown version 33
5.3 Absolute height occupancy 34
5.4 Gradient magnitudes and corresponding occupancy grid 35
5.5 Combined occupancy and its grown version 36
5.6 Path generated by Theta* 36
5.7 Theta* pseudo-code 37
5.8 Update node method pseudocode 38

6.1 Error histogram without distortion 41
6.2 Error distribution without distortion 42
6.3 Heightmaps without distortion . 43
6.4 Error map without distortion . . . 44
6.5 Error histogram with distortion . 46
6.6 Error distribution with distortion 46
6.7 Heightmaps with distortion 47
6.8 Error map with distortion 48
6.9 Navigation system scheme 50
6.10 Generated heightmap 51
6.11 Generated path between two points 51

B.1 DVD content 57

Tables

2.1 Used symbolic notation 3

4.1 OpenDroneMap binary calls in order 20
4.2 COLMAP binary calls in order . 22
4.3 Unreal Engine LOD disabling commands 25

5.1 Coordinate frames used by the photogrammetry implementations . 31
5.2 PDAL Pipeline arguments 33

6.1 Model aligner parameters 40
6.2 Heuristics without distortion . . . 41
6.3 Error statistics without distortion 41
6.4 Heuristics with distortion 45
6.5 Error statistics with distortion . . 45

Chapter 1 Introduction

1.1 Context

Robotics are becoming a cheaper and more effective tool for all sectors of economy. Major part of presently employed robotic systems are stationary machines in factories performing repetitive tasks. With recent advancement and increased accessibility of electronics, software and artificial intelligence, mobile robotics are becoming more prevalent most notably in the form of drones (robotic helicopters). These have great potential for use in many fields of profession including agriculture, civil engineering, archeology, law enforcement or entertainment. Their use however often requires a human supervisor, which somewhat restricts their efficiency. Making drones navigate autonomously is therefore highly desirable in order to enable their adoption by all industries that could use them.

To navigate through the environment, a model of the surrounding obstacles has to be first obtained, in which a collision-free path can be searched for using path finding algorithms. To construct this model, various sensor types can be used in practice such as LIDAR or an ordinary RGB camera. The camera is a popular choice for this task thanks to its low cost, low weight, small footprint and large range. However, whereas LIDAR can measure distance to obstacles directly, the camera provides only 2D images of the scene. From these images, the 3D information needs to be recovered using 3D reconstruction algorithms. There are in general two approaches to solving this problem: simultaneous localisation and mapping (SLAM) and photogrammetry. Both SLAM and photogrammetry have undergone vast advancement in the past 20 years due to rising accessibility of digital cameras and computing power. Whereas SLAM focuses on processing the data from the camera in real-time and providing a sparse and often inaccurate representation of the surroundings, photogrammetry focuses on dense and accurate reconstruction at the expense of processing rate.

1.2 Specification

The subject of this thesis is the use of visual terrain 3D reconstruction methods for drone navigation purposes. The scene is captured utilizing an RGB camera mounted on an aerial vehicle with precisely known position. An accurate and complete dense map of the environment is desired, making photogrammetry techniques preferable over less accurate SLAM. There are several publicly available photogrammetry implementations demanding an experimental comparison of suitable candidates to be conducted in order to select the best option. An appropriate method for processing the 3D model produced by the photogrammetry software must be devised to transform it into a convenient representation for latter navigation. Finally a navigation system will be implemented to enable save low-altitude flight above the mapped terrain. Performance of the whole system will be experimentally evaluated using a photo-realistic drone simulator.

Chapter 2
The theory behind photogrammetry

2.1 Mathematical apparatus

2.1.1 Notation

Unless otherwise stated, the notation described in table 2.1 is used throughout this thesis. One exception is the $\mathbf{X}$ symbol used to designate a 3D object point (instead of a matrix) in sections 2.2, 2.3 and 2.4. This notation was adopted from [1] and is commonly used in computer vision. This small exception to the otherwise standard notation (in table 2.1) improves readability.

Symbol	**Meaning**
x, X	scalar
O_i	coordinate frame i or its origin depending on context
$\mathbf{x}$	column vector
$\mathbf{x}^T$	row vector
$\mathbf{x}^i$	$\mathbf{x}$ with respect to O_i
$\mathbf{A}$	matrix A
$\mathbf{A}^{-1}$	inverse matrix
$\mathbf{A}^T$	matrix transpose
$\mathbf{A}^{-T}$	matrix inverse transpose
$\mathbf{A} := \mathbf{A}^i_j$	renaming the $\mathbf{A}^i_j$ matrix for readability
$\mathbf{A}(1,2)$	element of $\mathbf{A}$ on the first row and second column

Table 2.1: Used symbolic notation

2.1.2 Transformation of coordinates

Suppose we have two coordinate system O_j, O_i and a vector $\mathbf{x}$ with coordinates $\mathbf{x}^j \in \mathbb{R}^3$ as illustrated in figure 2.1. The relative pose of O_j with respect to O_i can be described using a rotation matrix $\mathbf{R} := \mathbf{R}^i_j \in \mathbb{R}^{3\times3}$ and a translation vector $\mathbf{t} := \mathbf{t}^i_j \in \mathbb{R}^3$.

The transformation of the vector coordinates $\mathbf{x}^j$ from O_j to O_i can than

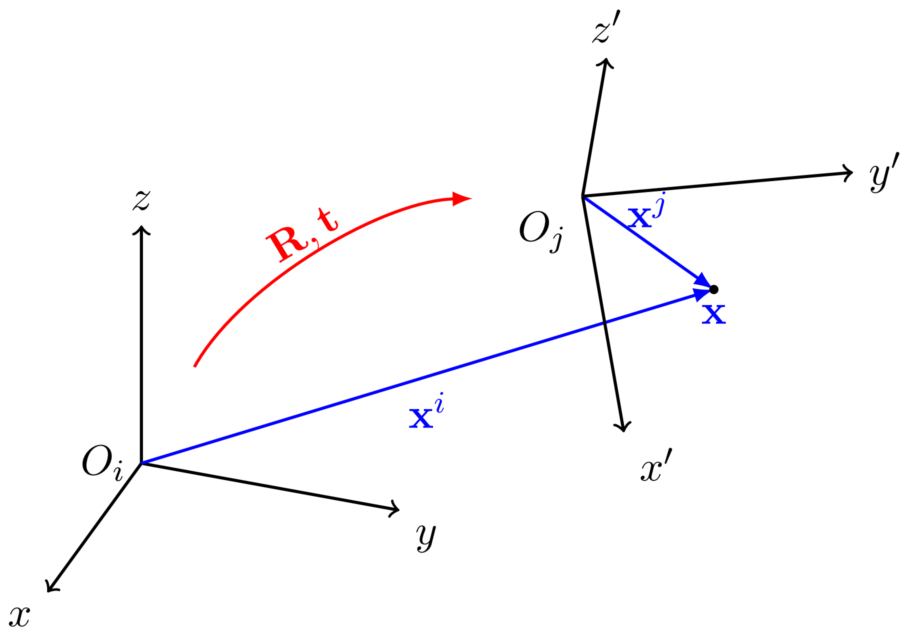


Figure 2.1: Coordinate frame transformation

be written as:

$$\mathbf{x}^i = \mathbf{R}\mathbf{x}^j + \mathbf{t}\,. \tag{2.1}$$

2.1.3 Homogeneous coordinates

The expression (2.1) can be simplified if we introduce homogeneous coordinates. The notation $\tilde{\mathbf{x}}$ is used to denote the homogeneous coordinates of vector $\mathbf{x}$. Let $\mathbf{x} = [x, y, z]^T \in \mathbb{R}^3$, then $\tilde{\mathbf{x}} = [wx, wy, wz, w]^T$, where $w \in \mathbb{R} \setminus \{0\}$ and is commonly chosen to be $w = 1$. We can use following relations to transform between homogeneous and non-homogeneous coordinates of vectors in $\mathbb{R}^3$:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \to \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \to \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix} \tag{2.2}$$

and similarly for vectors in $\mathbb{R}^2$:

$$\begin{bmatrix} u \\ v \end{bmatrix} \to \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \qquad \begin{bmatrix} u \\ v \\ w \end{bmatrix} \to \begin{bmatrix} u/w \\ v/w \end{bmatrix}\,. \tag{2.3}$$

From this follows, that two homogeneous representations of a vector that differ up to scale $\tilde{\mathbf{x}}_1 = \lambda\tilde{\mathbf{x}}_2$, $\lambda \in \mathbb{R} \setminus \{0\}$ can be considered equivalent and represent the same non-homogeneous coordinates. This fact is denoted using the notation: $\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{x}}_2$. Using homogeneous coordinates, we can rewrite (2.1) as:

$$\tilde{\mathbf{x}}^i = \mathbf{T}^i_j \tilde{\mathbf{x}}^j \,, \tag{2.4}$$

where

$$\mathbf{T}^i_j = \begin{bmatrix} \mathbf{R}^i_j & \mathbf{t}^i_j \\ \mathbf{0} & 1 \end{bmatrix} \tag{2.5}$$

is called homogeneous transformation matrix. In this way homogeneous coordinates allow us expressing a change of coordinate frame using a simple matrix multiplication.

2.1.4 Vector product

The vector product of two vectors $\mathbf{x} = [x_1, x_2, x_3]^T$, $\mathbf{y} = [y_1, y_2, y_3]^T$ in $\mathbb{R}^3$ is defined as:

$$\mathbf{x} \times \mathbf{y} = \begin{bmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_1 y_2 - x_2 y_1 \end{bmatrix} \,, \tag{2.6}$$

the same result can also be achieved by left-multiplying $\mathbf{y}$ by the following skew-symmetric (anti-symmetric) matrix:

$$[\mathbf{x}]_\times = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \tag{2.7}$$

2.1.5 Convolution on matrices

The two dimensional convolution on two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{k \times l}$, where $m \leq k \wedge n \leq l$ is defined as a new matrix $\mathbf{C} \in \mathbb{R}^{(m-k+1) \times (n-l+1)}$ whose elements are obtained as follows:

$$\mathbf{C}(u, v) = \sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{A}(i, j) \mathbf{B}(r + c - 1, c + j - 1) \,. \tag{2.8}$$

This discrete version of convolution on matrices is commonly used in computer vision for image processing, where the matrix $\mathbf{A}$ is often referred to as the **kernel**, $\mathbf{B}$ is the input image matrix and $\mathbf{C}$ is the output image matrix.

2.2 Modeling the camera

In this section the basics of the pinhole camera model and how the image is formed will be covered.

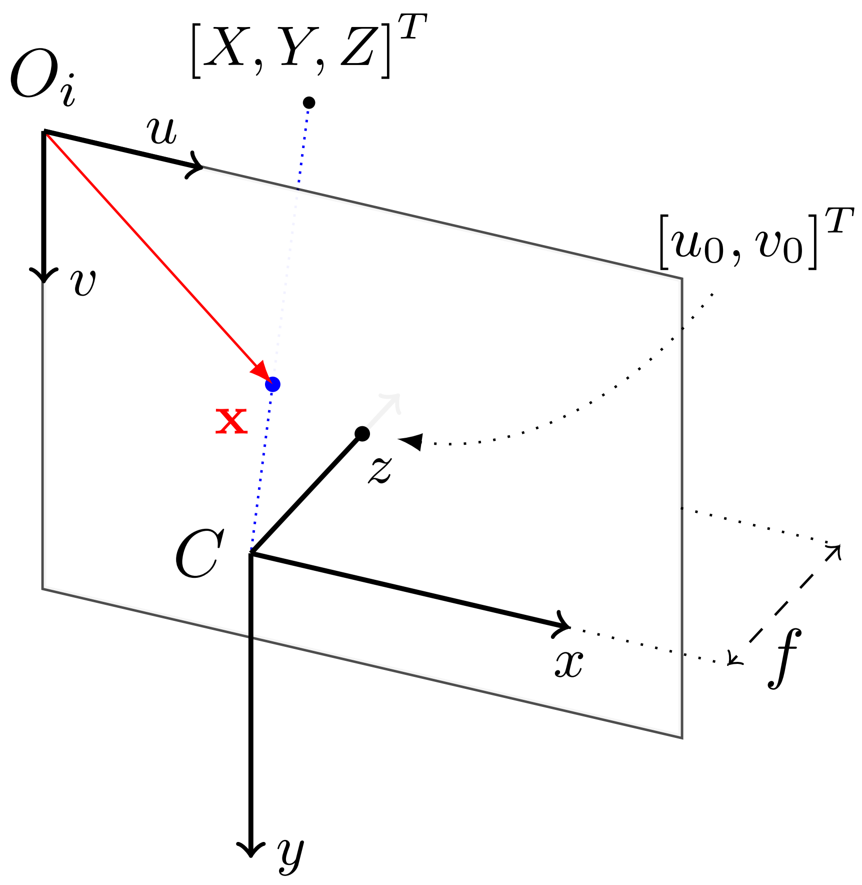


Figure 2.2: Pinhole camera at the origin

2.2.1 Pinhole camera model

Pinhole camera model is the simplest way to model the projection of 3D points into the so called image plane. The image points will be represented by the two-dimensional coordinates $[u, v]^T$ unless stated otherwise. These coordinates are any real numbers in theory, but in practice are restricted to whole numbers and represent pixel coordinates in an image. A camera is always associated with a coordinate frame, with which it shares its center and the image plane lies in the xy-plane of this coordinate frame.

Camera at the origin

Suppose we have a pinhole camera C with its center at the origin and a point $\mathbf{X} = [X, Y, Z]^T$ as depicted in figure 2.2. The image coordinates $\mathbf{x} = [u, v]^T$ (the image point) of the projection of $\mathbf{X}$ are obtained as follows:

$$u = f_x \frac{X}{Z} + u_o\,, \qquad v = f_y \frac{Y}{Z} + v_o\,, \tag{2.9}$$

where f_x and f_y are the focal lengths for x and y axis respectively. The $[u_0, v_0]^T$ represent the coordinates of the **principal point**. The equation

(2.9) can be written in terms of homogeneous coordinates of the image point as:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \sim \underbrace{\begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} , \tag{2.10}$$

where $\mathbf{K}$ is known as the **camera intrinsic matrix** or as the calibration matrix. Sometimes the image coordinate axes are not perfectly square and that can be modelled by adding a skew term s to the intrinsic matrix:

$$\mathbf{K} = \begin{bmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} , \tag{2.11}$$

though the term s is usually zero in modern cameras and is thus omitted. Moreover the focal lengths f_x and f_y tend to be equal in magnitude and the principal point $[u_0, v_0]^T$ is often in the center of the image. These facts then justify the most commonly used form of the intrinsic matrix:

$$\mathbf{K} = \begin{bmatrix} f & 0 & w/2 \\ 0 & f & h/2 \\ 0 & 0 & 1 \end{bmatrix} , \tag{2.12}$$

where the f is called the **focal length** and w and h denote the width and height of the image respectively. Unless stated otherwise, the intrinsic matrix of the form 2.12 will be considered.

General camera orientation

Suppose we have a 3D point $\mathbf{X}^w = [X, Y, Z]^T$, with O_w being the world coordinate system and a camera C with its orientation defined by a rotation matrix $\mathbf{R}^c_w \in \mathbb{R}^{3\times3}$ and its center $\mathbf{c}^w = [c_x, c_y, c_z]^T$ as illustrated in figure 2.3. We need to first transform the coordinates of $\mathbf{X}^w$ into the cameras coordinate frame and then apply the pinhole camera model (2.10):

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \sim \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} \left(\mathbf{R}^c_w \mathbf{X}^w - \mathbf{R}^c_w \mathbf{c}^w\right) . \tag{2.13}$$

We can again simplify the equation (2.13) by using homogeneous coordinates $\tilde{\mathbf{X}}^w$ and by introducing a new vector $\mathbf{t}^c_w = -\mathbf{R}^c_w \mathbf{c}^w$ that represents the coordinates of world origin in the camera's coordinate frame:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \sim \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \underbrace{\begin{bmatrix} \mathbf{R}^c_w & \mathbf{t}^c_w \\ \mathbf{0} & 1 \end{bmatrix}}_{\mathbf{T}^c_w} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} , \tag{2.14}$$

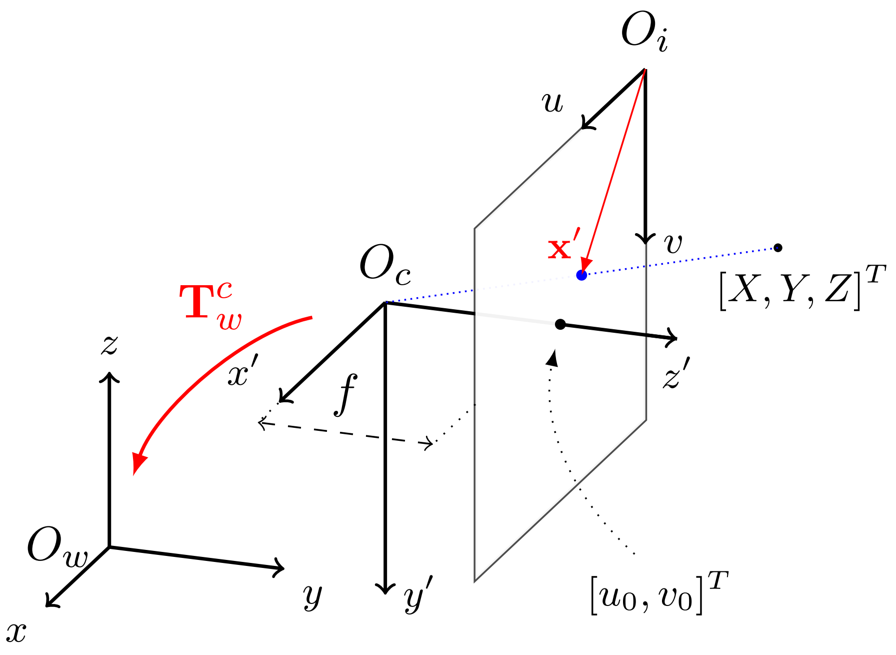


Figure 2.3: General pinhole camera

where $\mathbf{T}_w^c$ is called the **extrinsic camera matrix**. The three matrices in equation (2.14) can be multiplied together to form a so-called camera matrix:

$$\mathbf{P} = \mathbf{K} \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{T}_w^c \,, \tag{2.15}$$

which finally enables us to formulate the pinhole camera model as a simple linear transformation between homogeneous coordinates of the image point $\tilde{\mathbf{x}}$ and the 3D point $\tilde{\mathbf{X}}^w$:

$$\tilde{\mathbf{x}} = \mathbf{P}\tilde{\mathbf{X}}^w \,. \tag{2.16}$$

2.2.2 Normalized image coordinates

Suppose we have a point $\mathbf{X}^1$ seen by two cameras C_1 and C_2, with the first camera at origin and the second camera's pose described by a homogeneous transformation matrix $\mathbf{T}_1^2$. Suppose that the cameras have intrinsic matrices $\mathbf{K}_1$ and $\mathbf{K}_2$ respectively. Then the image coordinates $\tilde{\mathbf{x}}_1 = [u_1, v_1, 1]^T$ and $\tilde{\mathbf{x}}_2 = [u_2, v_2, 1]^T$ of point $\mathbf{X}^1$ captured by cameras C_1 and C_2 respectively can be obtained using (2.10) and (2.13) as follows:

$$\begin{aligned} \tilde{\mathbf{x}}_1 &\sim \mathbf{K}_1 \mathbf{X}^1 \,, \\ \tilde{\mathbf{x}}_2 &\sim \mathbf{K}_2 \left(\mathbf{R}_1^2 \mathbf{X}^1 + \mathbf{t}^2 \right) \,. \end{aligned} \tag{2.17}$$

We then define the homogeneous normalized image coordinates of $\tilde{\mathbf{x}}_1$ and $\tilde{\mathbf{x}}_2$ as:

$$\begin{aligned}\tilde{\mathbf{y}}_1 &= \mathbf{K}_1^{-1}\tilde{\mathbf{x}}_1\,,\\ \tilde{\mathbf{y}}_2 &= \mathbf{K}_2^{-1}\tilde{\mathbf{x}}_2\,.\end{aligned} \tag{2.18}$$

The normalized coordinates represent how the cameras would see the point $\mathbf{X}^1$, if both cameras had intrinsic matrices equal to unity i.e. unit focal length and the principal point at $[0,0]^T$. In fact substituting (2.17) into (2.18) yields:

$$\begin{aligned}\tilde{\mathbf{y}}_1 &\sim \mathbf{X}^1\,,\\ \tilde{\mathbf{y}}_2 &\sim \left(\mathbf{R}_1^2\mathbf{X}^1 + \mathbf{t}^2\right)\,.\end{aligned} \tag{2.19}$$

The equation (2.9) can be rewritten using the normalized coordinates (substituting $f_x = f_y = f = 1$ and $[u_0, v_0]^T = \mathbf{0}$) as follows:

$$\tilde{\mathbf{y}} = \frac{1}{Z}\begin{bmatrix}X\\Y\\Z\end{bmatrix} = \frac{1}{Z}\mathbf{X}\,, \tag{2.20}$$

where $\tilde{\mathbf{y}}$ are the homogeneous normalized image coordinates and $\mathbf{X} = [X,Y,Z]^T$ are the 3D point coordinates in the cameras reference frame.

2.2.3 Lens distortion

For real-world cameras the pinhole camera model is often not accurate enough, because in reality cameras use lenses that are not infinitely small as the pinhole model assumes. These lenses distort the image, which can be accounted for and corrected in software using the Brown-Conrady [2] model[1]. Suppose that $\mathbf{y}_d = [u_d, v_d]^T$ are the normalized image coordinates in the distorted image and $\mathbf{y} = [u,v]^T$ are the normalized image coordinates in the undistorted image corresponding to the same 3D object point. That the lens distortion can be modelled as follows:

$$\begin{aligned}u_d &= u + (u-u_c)(K_1r^2 + K_2r^4) + \\ &\quad + \left[P_1(r^2 + 2(u-u_c)^2) + 2P_2(u-u_c)(v-v_c)\right]\,,\\ v_d &= v + (v-v_c)(K_1r^2 + K_2r^4) + \\ &\quad + \left[2P_1(u-u_c)(v-v_c) + P_2(r^2 + 2(v-v_c)^2)\right]\,,\end{aligned} \tag{2.21}$$

$$\text{with } r^2 = u^2 + v^2$$

where $\mathbf{y}_c = [u_c, v_c]$ is the distortion center. The parameters K_1 and K_2 are referred to as **radial distortion coefficients** and P_1 and P_2 are called **tangential distortion coefficient**. Using the equation (2.21) it is possible to re-sample a distorted image into an image that would be captured by an ideal pinhole camera. This process is referred to as image undistortion.

[1]Higher order approximation than (2.21) can be used and thus a more accurate model can be obtained, but two coefficient usually suffice for practical applications.

Intrinsic parameters

The distortion coefficients together with the parameters in the intrinsic matrix $\mathbf{K}$ are called the **intrinsic parameters** of the camera and if known, the camera is said to be **calibrated**.

2.3 Depth estimation

Depth estimation refers to the process of determining the depth of a 3D point, whose image we have captured. For depth estimation to work, it is necessary for the point to be seen by at least two cameras with non-zero relative translation.

2.3.1 Stereo depth estimation

Suppose we have two cameras C_1 and C_2 that share a common orientation, focal length f and are only translated with respect to one another by distance B as illustrated in figure 2.4. Let $\mathbf{x}_1 = [u_1, v_1]^T$, $\mathbf{x}_2 = [u_2, v_2]^T$ denote the image coordinates of projections of point $\mathbf{X}$ into the image planes of C_1, and C_2, then we define the **disparity** d of $\mathbf{X}$ as:

$$d = u_1 - u_2\,. \tag{2.22}$$

The distance of point $\mathbf{X}$ from the xy-plane of the two cameras can be obtained using similarity of triangles shown on figure 2.4 as follows:

$$Z = \frac{Bf}{d}\,, \tag{2.23}$$

where Z is depth of the point $\mathbf{X}$. With known depth the X and Y coordinates in the C_1 coordinate frame can be deduced:

$$\begin{aligned} \frac{X}{Z} &= \frac{u_1}{f} \quad \implies \quad X = Z\frac{u_1}{f} = \frac{Bu_1}{d} \\ \frac{Y}{Z} &= \frac{v_1}{f} \quad \implies \quad Y = Z\frac{v_1}{f} = \frac{Bv_1}{d} \end{aligned} \tag{2.24}$$

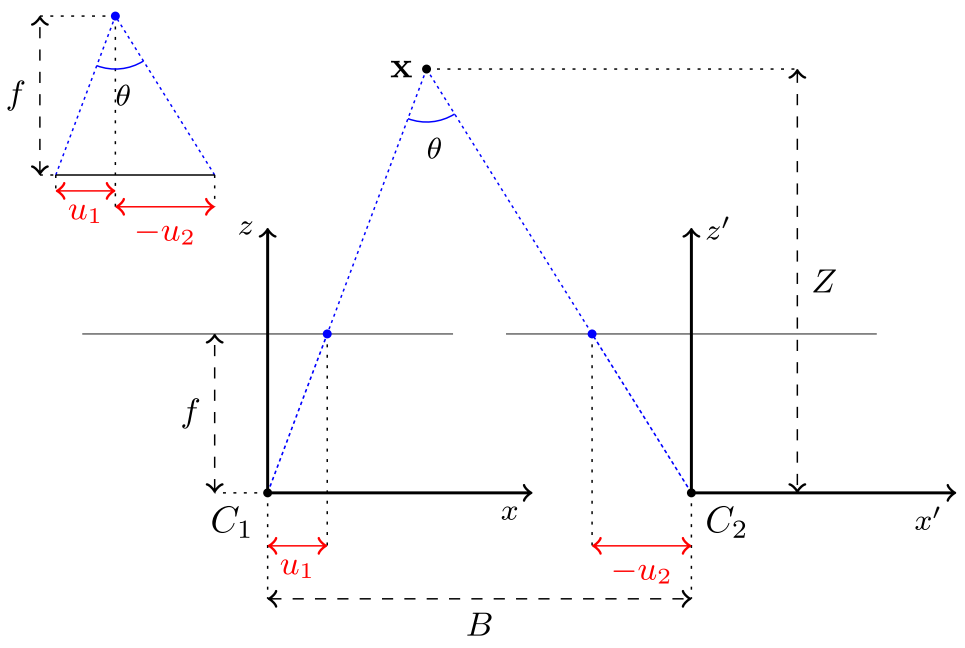


Figure 2.4: Stereo depth estimation

2.3.2 Triangulation

General camera configuration

Consider the situation from section 2.2.2 with:

$$\mathbf{T}_1^2 = \begin{bmatrix} \mathbf{R}_1^2 & \mathbf{t}^2 \\ \mathbf{0} & 1 \end{bmatrix}, \qquad \mathbf{R}_1^2 = \begin{bmatrix} \mathbf{r}_1^T \\ \mathbf{r}_2^T \\ \mathbf{r}_3^T \end{bmatrix}, \qquad \mathbf{t}^2 = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}.$$

Then we can express the u_2 coordinate of $\tilde{\mathbf{y}}_2$ using (2.20):

$$u_2 = \frac{\mathbf{r}_1^T \mathbf{X}^1 + t_1}{\mathbf{r}_3^T \mathbf{X}^1 + t_3} = \frac{\mathbf{r}_1^T \tilde{\mathbf{y}}_1 + t_1/Z}{\mathbf{r}_3^T \tilde{\mathbf{y}}_1 + t_3/Z}. \tag{2.25}$$

Rearranging this expression:

$$\begin{aligned} & u_2 \mathbf{r}_3^T \tilde{\mathbf{y}}_1 + u_2 t_3/Z = \mathbf{r}_1^T \tilde{\mathbf{y}}_1 + t_1/Z\,, \\ \implies \quad & \frac{1}{Z}(u_2 t_3 - t_1) = \mathbf{r}_1^T \tilde{\mathbf{y}}_1 - u_2 \mathbf{r}_3^T \tilde{\mathbf{y}}_1\,, \end{aligned} \tag{2.26}$$

which after solving for the Z coordinate of $\mathbf{X}^1$:

$$Z = \frac{u_2 t_3 - t_1}{\mathbf{r}_1^T \tilde{\mathbf{y}}_1 - u_2 \mathbf{r}_3^T \tilde{\mathbf{y}}_1}, \tag{2.27}$$

enables us to recover the original 3D point $\mathbf{X}^1$ coordinates using (2.20) as:

$$\mathbf{X}^1 = Z \tilde{\mathbf{y}}_1\,. \tag{2.28}$$

Rectified camera pair

If we consider the so called rectified camera configuration where:

$$\mathbf{R}_1^2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{t}^2 = \begin{bmatrix} -B \\ 0 \\ 0 \end{bmatrix}. \tag{2.29}$$

the equation (2.27) has a particularly simple form:

$$Z = \frac{B}{d}, \tag{2.30}$$

that we have already seen in section 2.3.1. This rectified configuration is typical for stereo camera rigs. The formula (2.30) can be used even when the original cameras are not in this rectified configuration. Assuming the relative pose of the cameras is known, it is possible to warp the images into the rectified form in a process called **camera rectification**. The details of this technique are beyond the scope of this work.

2.4 Challenges

What enabled the recovery of a 3D point seen by the camera in section 2.3.2 were known intrinsic and extrinsic parameters of both cameras. Furthermore the corresponding image points $\tilde{\mathbf{x}}_2$ and $\tilde{\mathbf{x}}_1$ belonging to the same 3D point were assumed to be known. These conditions are however usually not fulfilled in the real world and present the main problem for photogrammetry.

2.4.1 Camera calibration

The intrinsic parameters are usually obtained using camera calibration techniques. This is commonly done by capturing several images of 3D points with known location while varying the camera's pose. Using non-linear optimization it is then possible to solve for the camera's intrinsic parameters.

2.4.2 Correspondence problem

Suppose we have two cameras C_1 and C_2 with camera matrices $\mathbf{P}_1$ and $\mathbf{P}_2$ respectively and we wish to find the image coordinates $\tilde{\mathbf{x}}_2$ for given image coordinates $\tilde{\mathbf{x}}_1$ such that:

$$\tilde{\mathbf{x}}_1 = \mathbf{P}_1 \tilde{\mathbf{X}}^w \qquad \wedge \qquad \tilde{\mathbf{x}}_2 = \mathbf{P}_2 \tilde{\mathbf{X}}^w, \tag{2.31}$$

where $\tilde{\mathbf{X}}^w$ are homogeneous coordinates of some 3D point satisfying this relation. This is known as the correspondence problem and is the key challenge in computer vision. It is usually solved by comparing small patches of the two images and finding ones which look similar enough. Though it seems like we have to search through the whole image to find $\tilde{\mathbf{x}}_2$, this problem can actually be reduced to a line search in the second image using epipolar geometry constraints.

Epipolar geometry

Suppose two cameras C_1 and C_2 with relative pose defined by $\mathbf{R} := \mathbf{R}_2^1$ and $\mathbf{t} := \mathbf{t}_1^2$ as depicted in figure 2.5. The vectors $\mathbf{X} := \mathbf{X}^1$, $\mathbf{X}' := \mathbf{X}^2$ and $\mathbf{t}$ lie in the same plane in 3D space, which can be expressed as:

$$(\mathbf{t} \times \mathbf{X})^T (\mathbf{R}\mathbf{X}' + \mathbf{t}) = 0 . \tag{2.32}$$

Now using skew-symmetric matrix to reformulate the vector product we gain:

$$([\mathbf{t}]_\times \mathbf{X})^T (\mathbf{R}\mathbf{X}' + \mathbf{t}) = \mathbf{X}^T [\mathbf{t}]_\times^T \mathbf{R}\mathbf{X}' = 0 . \tag{2.33}$$

We can rewrite this using the normalized image coordinates $\tilde{\mathbf{y}}$ and $\tilde{\mathbf{y}}'$ representing the projections of $\mathbf{X}$ and $\mathbf{X}'$ respectively using the relation (2.20):

$$\tilde{\mathbf{y}}^T \mathbf{E} \tilde{\mathbf{y}}' = 0 , \tag{2.34}$$

where the matrix $\mathbf{E} = [\mathbf{t}]_\times^T \mathbf{R} \in \mathbb{R}^{3\times3}$ is called the **Essential matrix** and the equation (2.34) represents the so called epipolar constraint for normalized coordinates. Suppose the cameras C_1 and C_2 have intrinsic matrices $\mathbf{K}_1$ and $\mathbf{K}_2$ respectively, then using (2.18) we can express this equation for the image coordinates $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{x}}'$ as follows:

$$\begin{aligned} &\left(\mathbf{K}_1^{-1}\tilde{\mathbf{x}}\right)^T \mathbf{E} \left(\mathbf{K}_2^{-1}\tilde{\mathbf{x}}'\right) = 0 \\ \Longrightarrow \qquad &\tilde{\mathbf{x}}^T \mathbf{K}_1^{-T} \mathbf{E} \mathbf{K}_2^{-1} \tilde{\mathbf{x}}' = 0 , \end{aligned} \tag{2.35}$$

which we can rewrite as

$$\tilde{\mathbf{x}}^T \mathbf{F} \tilde{\mathbf{x}}' = 0 , \tag{2.36}$$

where the matrix $\mathbf{F} = \mathbf{K}_1^{-T} \mathbf{E} \mathbf{K}_2^{-1} \in \mathbb{R}^{3\times3}$ is called the **Fundamental matrix** and the equation (2.36) represents the epipolar constraint for image coordinates. Let $\tilde{\mathbf{y}} = [u, v, 1]^T$ represent a fixed point in the first camera's image. Then denoting $\mathbf{l}^T := \tilde{\mathbf{y}}^T \mathbf{E}$ in the equation (2.34) yields:

$$\mathbf{l}^T \tilde{\mathbf{y}}' = \begin{bmatrix} l_1 & l_2 & l_3 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = l_1 u + l_2 v + l_3 = 0 \tag{2.37}$$

for the normalized image coordinates (or similarly for $\mathbf{l}^T := \tilde{\mathbf{x}}^T \mathbf{F}$ in 2.36). The equation (2.37) constraints the location of the corresponding image point $\tilde{\mathbf{y}}'$ ($\tilde{\mathbf{x}}$) to a line in the second image. Such a line is called the **epipolar line**.

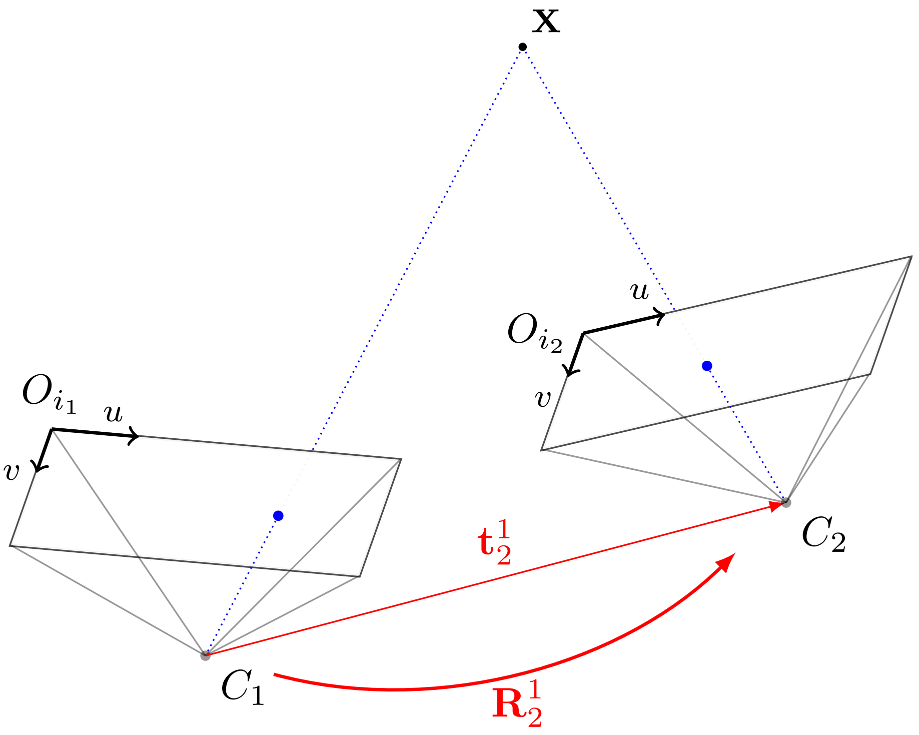


Figure 2.5: Epipolar geometry

2.4.3 Pose estimation

Without diving too deep into the pose estimation problem, we will provide an outline of a method of recovering relative pose of the two cameras C_1 and C_2. First at least some correspondences need to be established between the images (usually 8). At this point, the epipolar geometry can not be exploited, because the pose is unknown. This is why the search often is reduced to only features - interesting, well distinguishable points in the image, where we can robustly estimate the correspondences. With the corresponding points we can estimate the epipolar or the fundamental matrix depending on whether the cameras are calibrated or not. From the epipolar or fundamental matrix, it is then possible to extract the relative orientation of the two cameras up to scale. A more in-depth explanation can be found in [3, 1].

2.5 Photogrammetry pipeline

Photogrammetry is the process of taking a set of unordered images and creating a 3D model of the captured scene. Most of the available implementations output this 3D model in the form of a point cloud, i.e. a set of 3D points that represent the real world scene being captured. Nearly all of the present state-of-the art methods can be divided into two separate stages: **Structure-from-Motion** (SfM) and **Multi-View Stereo** (MVS). Sometimes there is also a third stage that deals with creating a possibly textured triangular mesh from this point cloud as a post-processing step. However as we are only interested in the generated point cloud, this stage will not be considered further.

2.5.1 Structure-from-Motion

Structure from motion tries to estimate mainly the intrinsic and extrinsic parameters of cameras in the scene. In the process it also reconstruct a sparse point cloud of the environment, which is however often too sparse for the desired application. It can be divided into a sequence of three steps:

1. **Feature detection and description**

   First of all, every image is searched for **features** - groups of pixels that are well distinguishable from other parts of the image. For each of these features a so called **descriptor** is computed, which serves as a signature of the feature, so it can later be found in other images and is stored together with the feature coordinates. In the present the most popular method for selecting good features is SIFT [4].

2. **Feature matching and filtering**

   In this step the features are matched against the features in other images to form **correspondences** based on some similarity measure of their descriptors. After the correspondences are established, the outliers are usually filtered out using **RANSAC** (**ran**dom **sa**mple **c**onsensus) [5] scheme, where the largest set of features satisfying an epipolar constraint (Sec. 2.4.2) is searched for in the corresponding image pair and the rest is discarded. The result are geometrically consistent pairs of features in the images, which are likely to correspond to the same 3D points.

3. **Solving for SfM model and bundle adjustment**

   In this step the relative poses of all the cameras are estimated along with the intrinsic parameters and are then iteratively improved in a process called bundle adjustment[2]. The most popular approach for this step is called **Incremental SfM**, where first an initial image pair is selected and its pose estimated, than more images are incrementally added to the reconstruction and bundle adjusted one by one.

2.5.2 Multi-view Stereo

The MVS stage takes in the images together with their cameras' poses and intrinsics and outputs a dense point cloud representation of the scene. MVS is usually a lot slower than the SfM stage and accounts for most of the processing time. In the MVS stage, using the intrinsics estimated in SfM, the

[2]Non-linear optimization problem that is usually solved using sparse Levenberg-Marquardt algorithm

distortion (Sec. 2.2.3) is removed from the images. On the undistorted images it then runs a dense correspondence algorithm i.e. correspondence search for every pixel in the image. The correspondence problem can be reduced into a line search problem thanks to the epipolar constraint (Sec. 2.4.2) since the pose is already known. The images can also be pairwise rectified into the stereo configuration (Sec. 2.3.1) in order to exploit a more memory-efficient horizontal line search in the images. Using the correspondence it is then possible to triangulate (Sec. 2.3.2) the points in the image.

Chapter 3

Geographic coordinate systems

This section is a brief introduction to coordinate systems commonly used for expressing the geographic position (position on Earth).

3.1 ECEF

ECEF (**E**arth **C**entered **E**arth **F**ixed) refers to a cartesian coordinate system with its origin at the Earth's center of mass. The z axis coincides with the Earth's geodetic north, the x axis intersects the Earth at its equator and prime meridian. The y axis completes a right-handed cartesian coordinate system.

3.2 WGS

The **WGS** (**W**orld **G**eodetic **S**ystem) is a spherical coordinate system with its origin at the Earth's center of mass. The three coordinates longitude, latitude and altitude define position with respect to a reference ellipsoid that approximates shape of the Earth. Latitude denotes the angle between a point and the Earth's equator, longitude denotes the angle between a point and the Earth's prime meridian and altitude is the distance from Earth's surface. This coordinate system is used for example by the GPS system.

3.3 ENU

The **ENU** (**E**ast **N**orth **U**p) is a cartesian coordinate system with its origin at the Earth's surface, the x axis pointing to the east, y axis pointing to the north and the z axis pointing upwards. It is a useful coordinate frame for small areas, where the Earth's curvature does not have much of an effect and is commonly used for example in aviation. Note that to describe the ENU coordinates one must first specify where is the origin on the Earth's surface. This origin can be described using the ECEF or WGS coordinates.

Illustration

The figure 3.1 depicts the ENU, ECEF and WGS coordinate systems as situated on the Earth. The x, y, z axes belong to the ECEF, the *lon* and *lat* angles denote the longitude and latitude of the WGS system and the East, North, Up axes denote the ENU coordinate system. The altitude coordinate of the WGS system is the same as the Up coordinate in the ENU.

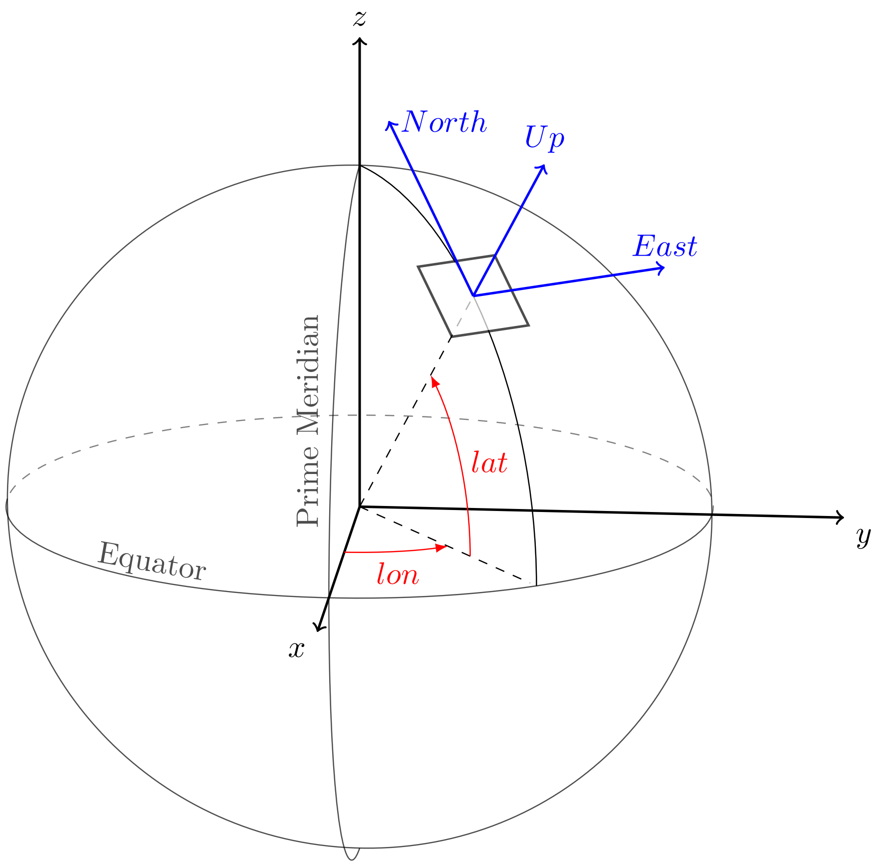


Figure 3.1: Geographic coordinate systems

Chapter 4

Used software

4.1 External libraries

As there are many open-source implementations of photogrammetry software available, it is difficult to select one over the others, since it is not clear which one is most suitable for the task of terrain mapping. Therefore three of the most popular libraries in photogrammetry community [6, 7] were selected for latter comparison.

4.1.1 OpenDroneMap

About

OpenDroneMap [8] (ODM) is a popular photogrammetry tool designed specifically for terrain mapping using a UAV equipped with a camera. After creating the 3D reconstruction, it also by default creates a textured mesh, orthophoto, digital elevation model and can even geo-reference these[1] using GPS data from the drone. These functions can be useful for creating maps of the environment, but we will utilize only the generated point cloud. ODM is written in Python which is mostly used here as a scripting language and the computationally expensive tasks are performed by external programs that are mainly written in faster C++.

Usage

Once the external libraries are compiled, the main Python script *run.py* that takes care of the whole reconstruction process can be run. User only has to specify the project path, where the images are stored and the output will be saved. It also offers many options to customize the reconstruction process, the most important are shown in table 4.1.

[1]The coordinate system relation to GPS coordinates is stored along with the data.

Argument	Function
--project-path	determines the project directory, which must include a directory "images" with the images for reconstruction
--feature-quality	determines feature extraction quality
--pc-quality	determines by how much are the images scaled down for the MVS stage
--cameras	enables using a JSON[2] file containing camera intrinsics
--geo	enables using a text file storing GPS data
--split	enables splitting the images into groups before reconstruction to save memory
--matcher-neighbours	how many nearest cameras (with respect to GPS coordinates) to use for feature pre-matching

Table 4.1: OpenDroneMap binary calls in order

4.1.2 openMVG + openMVS

About

Open Multi-View Stereo (OpenMVS) [9] is one of the most popular photogrammetry implementations available to public, thanks to its great performance and speed. Besides the point cloud generation it also includes the functionality of textured mesh generation. It however includes only the MVS stage of photogrammetry and needs a full SfM reconstruction completed by a different program. A popular choice in the community for its SfM complement is the OpenMVG [10] library, which is what we are going to use. Both OpenMVG and OpenMVS are written in C++.

Usage

Once compiled, both of these libraries can be used via convenient binaries, one for each step of the reconstruction pipeline. This provides users great flexibility to tailor the program for their needs. The most important OpenMVG binaries are:

1. **openMVG_main_SfMInit_ImageListing**

   Takes care of loading the images database and camera intrinsics.

2. **openMVG_main_ComputeFeatures**

   Computes features and descriptors.

3. **openMVG_main_ComputeMatches**

   Creates a list of corresponding matches.

4. **openMVG_main_IncrementalSfM**

   Reconstructs the cameras' poses incrementally.

5. **openMVG_main_ChangeLocalOrigin**

   Changes the scale and origin of the SfM reconstruction, so that it is better handled by the MVS stage.

6. **openMVG_main_geodesy_registration_to_gps_position**

   Outputs transformation parameters from the SfM coordinates into the specified GPS coordinates. This can be later used to georeference the MVS output.

7. **openMVG_main_openMVG2openMVS**

   Exports the SfM reconstruction into a format OpenMVS can use.

From the OpenMVS library, only one binary will be used and that is **DensifyPointCloud**. The most important argument of this binary is --*resolution-level*, which determines how much the images are scaled down before the reconstruction process.

4.1.3 COLMAP

About

COLMAP [11, 12] is one of two most popular open-source photogrammetry programs, that make use of a GPU[3] to accelerate the reconstruction process. The other such program being MeshRoom [13], which however does not as of yet enable the use of GPS coordinates to make the terrain map usable for navigation purposes. This library is written in C++ and CUDA. Once compiled, there is a single main binary, through which Colmap can be used. It can be accessed via a graphical user interface, which however does not enable automation of the process, and via a command line interface (CLI).

Usage

The CLI can be used in two ways, the first option is calling it with the *automatic_reconstructor* argument which handles the full reconstruction by itself. It however is not very flexible and does not enable specifying known intrinsic parameters or the camera positions. The second option, which we are going to use, is manually running COLMAP via the CLI for each stage of the photogrammetry process separately. This approach enables fine-tuning of parameters and most importantly specifying the intrinsic parameters and poses of the cameras easily. The individual calls one has to make for the complete reconstruction are shown on table 4.2 in order of execution.

4.2 Simulator

The performance of selected photogrammetry implementations will be evaluated using artificial dataset generated inside a simulator. Although simulation

[3]Graphics processing unit

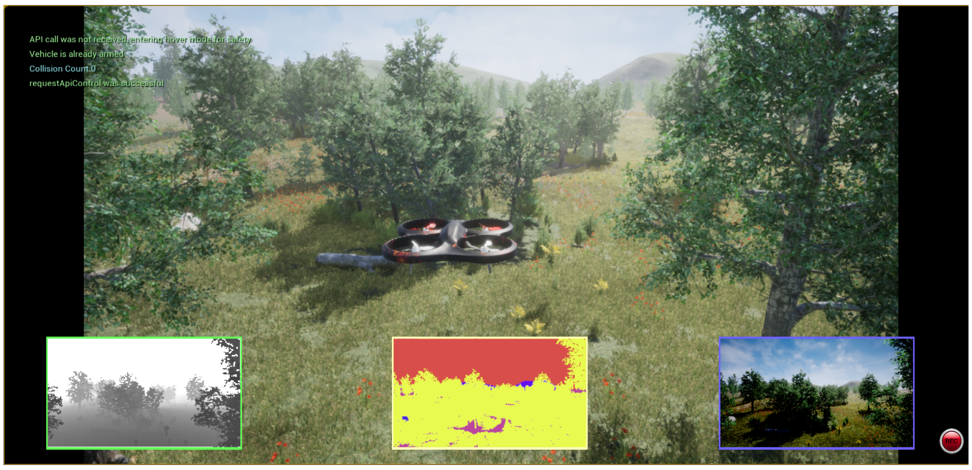
Argument	Function
feature_extractor	feature extraction and description
exhaustive_matcher	feature matching
mapper	incremental SfM
model_aligner	align sparse model with xyz coordinates
image_undistorter	removing distortion
patch_match_stereo	depth map reconstruction
stereo_fusion	fusion of depth maps

Table 4.2: COLMAP binary calls in order

can never fully replace real-life experiments, with present technology it comes close and offers many advantages. Experiments can be done at any time and any place independent of the weather outside, or the current pandemic related restrictions. Also real-life drones have limited battery charge and can get damaged if things go wrong. Using simulation instead (if feasible) alleviates all these troubles and thus accelerates research and development.

4.2.1 AirSim

AirSim [14] (**A**erial **I**nformatics and **R**obotics **Sim**ulation) is an open-source simulator intended for robotics and AI research. It is not a standalone program but rather a plugin for Unreal Engine [15], which is a 3D rendering engine. Together they form a photorealistic simulation platform, which is particularly useful for visually based AI and robotics research. A screenshot from the simulation can be seen in figure 4.1.



```
API call was not received, entering hover mode for safety
Vehicle is already armed
Collision Count:0
requestApiControl was successful
```

Figure 4.1: AirSim simulation

The AirSim project enables taking any virtual world made in the Unreal Engine and spawning a vehicle (car or a drone and more to be added) inside it. It is possible to control this vehicle and retrieve sensor data through Python or C++ API. The vehicle's movement is simulated using a physically based model independent of the visual simulation in order to be maintain steady response rate. Many types of sensor can be attached to the vehicle including accelerometer, barometer, GPS, cameras[4] and LIDAR. To provide realistic behaviour of the sensors, AirSim uses a physics based model for some of them.

Usage

After building the AirSim library, a plugin folder is simply moved into any Unreal Engine project, which then needs to be recompiled. That is all it takes to set up the simulation. After launching the AirSim simulator for the first time, a *settings.json* file is generated, in which the basic parameters for the simulation are set. These setting determine which type of vehicle is used, what sensors are simulated, camera intrinsic parameters, camera noise levels and more. The simulated drone can be controlled using both low-level and high-level commands ranging from control of individual motor voltages to setting desired location of the vehicle and letting the in-built AirSim flight controller to do the rest of the work.

4.2.2 Unreal Engine

The Unreal Engine (UE) is a free-to-use rendering engine, whose primary purpose was to provide a development platform for computer games, but today is used in many industries thanks to its photo-realistic visuals and ease of use. As it is primarily focused on gaming and entertainment industry, its use for computer vision research comes with some caveats. In this chapter, some basic concepts of UE are covered along with some of the issues that were encountered and how they were handled.

Actors

The basic elements of every environment in UE are **actors**. Each actor corresponds to an object (e.g. drone, terrain, lights, vegetation), that can be placed inside the environment, move around and generate collision events with other actors. Most actors have an associated 3D triangular mesh together with a 2D texture that define the actor's shape and appearance.

Camera actor

To get a visual output from the environment, a camera actor is placed inside the world, which captures images as modelled by the pinhole camera model. The camera in UE comes with many adjustable parameters, the most important being width and height of an image and focal length. More advanced settings such as motion blur, lens flares, or auto-exposure can be also used to mimic a real-life camera's behaviour. It does not unfortunately provide lens distortion simulation (Sec. 2.2.3), which needs to be added to the images manually.

[4]AirSim provides RGB, depth and infrared cameras.

Lighting

Lighting (and thus shadows) is unsurprisingly an important concept when it comes to producing a realistic looking virtual world. A **light** in UE is an actor used to illuminate the scene. There are many types of lights available in UE made for different purposes such as a point light source, directional light source, etc. During our experiments we use two types of lights: directional light to mimic direct sunshine and skylight to mimic the light coming from the sky. Atmospheric fog is also being used to simulate how light scatters in the atmosphere.

Every light needs to be configured to one of three light mobility modes:

- **Static**

  Uses precomputed lighting, which does not enable lighting changes during gameplay and is therefore not optimal when dealing with moving objects in the scene. The light building also takes a long time to complete and has to be redone every time the scene changes. The advantage is that precomputed lighting is more efficient during the simulation and increases frame rate.

- **Movable**

  Generates completely dynamic lighting and shadows, that however slow down the scene rendering especially for large scale environments. The advantage is that moving objects effect the lighting during the simulation, which enables moving objects having dynamic shadows, daytime changes during the simulation and simulating cloud shadows.

- **Stationary**

  This is a hybrid between the static and movable lighting enabling partially precomputed and partially dynamic lighting and shadows. It enables using more efficient static lighting for immovable objects and scene illumination, and use dynamic shadows for moving objects such as the vehicle, vegetation, or the clouds. It is however a bit more complicated to set up than the movable or static variants.

Stationary lighting is used in the experiments as the precomputed lighting looks more realistic in the static parts of the environment and the dynamic lighting and shadows can be useful for moving vegetation and clouds, which often complicate visual terrain reconstruction in real-life scenarios.

Level of detail

By default, many actors that represent 3D objects inside UE come with a built-in level of detail (LOD) system. An actor with LOD has more than one 3D mesh and texture associated with it, each with different level of detail - hence the name. When the camera is near the actor, the most detailed mesh and texture are used and as the camera moves further, the mesh and texture are replaced by a less detailed one in order to decrease the computational load. This is a useful feature for real-time use cases, where faraway objects take up little space on screen and the extra rendering speed is welcome. However for visual fidelity, an object changing its geometry during the simulation is not optimal. The LOD system needs to be therefore disabled, which can be achieved by passing in few commands (table 4.3) into the UE command line.

Command	Function
r.forceLOD 0	forces the highest detail mesh to be used for object rendering
r.forceLODShadow 0	forces the highest detail mesh to be used for shadow casting
foliage.forceLOD 0	forces the highest detail mesh to be used for foliage rendering

Table 4.3: Unreal Engine LOD disabling commands

Constraints

The Unreal Engine is well suited for creation of realistic looking environments. However, it has some limitations that need to be taken into account when using this tool, which somewhat restricts how we can use it.

- **Transparent materials**

  Transparent materials such as glass or water surfaces are not seen by the AirSim's depth camera. These materials need to be set to be opaque or removed entirely from the scene to assure consistency of the recovered depth maps and images.

- **Level loading**

  When moving the camera too fast through the environment, sometimes the level does not load fast enough and as a result corrupt data can be obtained from the rgb and depth camera.

- **Shadow distance**

  Dynamic shadows disappear when flying too far or too high with the drone as a result of underlying optimization in UE. One reason for this is the capped shadow draw distance, which can be increased up to 200 meters. The other is that if an object takes too little space on the screen, its dynamic shadow will not be drawn.

To mitigate these problems in our experiments, transparent materials are avoided and the drone moves slowly at relatively low altitude. Additionally dynamic shadows are disabled for small actors to avoid shadow flickering.

Chapter 5

Implementation

The implemented code is described in this chapter. The programming language of choice is Python because it is versatile and enables fast development. One instance, where Python would limit usability was the path searching algorithm, which was therefore rewritten in Cython[1] to gain two orders of magnitude faster execution time when compared to plain Python implementation.

5.1 AirSim interface

5.1.1 airsim_env.py

The *airsim_env.py* file contains the interface to the AirSim simulator in the form of three classes AirSimBase, AirSimCV and AirSimUAV.

AirSimBase

The first implemented class is the AirSimBase, which wraps the most basic and general methods such as:

- **__init__**

  Pulls camera intrinsics from the AirSim simulator and computes the Q reprojection matrix (section (5.2)).

- **get_rgb**

  Returns an RGB image taken from camera on the UAV.

- **get_disp**

  Returns a 2D floating point number matrix representing the disparities of individual pixels.

- **disable_lods**

  Disables the Unreal Engines level of detail system for higher consistency of subsequent photos.

[1]An extension of Python that enables typed variables and compilation into faster C code.

AirSimCV

The second class AirSimCV is intended for use with the AirSim Computer Vision mode. This mode does not simulate a vehicle and instead creates a weightless camera in the environment. This camera can be moved around to capture images without having to deal with the vehicle's dynamics. The AirSimCV class inherits all of the methods from the AirSimBase class and has some in addition such as:

- **set_pose**

  Takes 3D coordinates and pitch, roll and yaw[2] angles and uses these to set the camera pose.

- **get_cloud**

  Takes a list of camera poses and an output file path. It then loops through these poses and at each one, reconstructs the 3D scene seen by the camera and registers it into the world coordinate frame. The reconstructions from all the poses are saved into a point cloud file. Detailed description of this method follows in section 5.1.2.

- **save_rgbs_gps**

  Takes a list of poses of the camera and at each one captures an image along with GPS coordinates generated using an artificial WGS coordinate system.

AirSimUAV

The third class AirSimUAV is intended for use with the AirSim multirotor mode. As the name suggests, it enables the control of a UAV in the simulation. Some of the implemented methods are:

- **get_collision_info**

  Returns whether a collision has occurred since the start of the simulation.

- **move_to**

  Moves the drone to a specified location using AirSim's in-built flight controller.

- **get_xyz**

  Returns the current position of the drone.

- **get_gps**

  Returns the latitude, longitude and altitude GPS coordinates with respect to set GPS coordinates of the world origin.

[2]Yaw, pitch and roll are used to represent the subsequent rotation about x, y and z axes respectively.

- **move_on_path**

  Takes a list of 3D waypoints and flies through them using the AirSims in-build flight controller.

- **survey**

  Flies the drone through a list of 3D waypoints while taking pictures at each one and saves them into a folder together with their GPS coordinates.

5.1.2 Point cloud from AirSim

To get the ground-truth 3D point cloud of the to-be reconstructed 3D scene from photos, the disparity camera provided by AirSim API is used via the *get_disp* method. This method returns a *disparity* matrix, where each pixels stores the disparity of the corresponding pixel seen by the camera. We have already discussed how the point location can be obtained from disparity information in 2.3.1. We however do not need to implement this functionality as it is already included in the optimized open-source C++ library for computer vision OpenCV [16] in the form of method *reprojectImageTo3D*. This method accepts a matrix Q and a 2D matrix of disparities and returns a 2D matrix of 3D points. It actually performs just a simple matrix multiplication:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = Q \begin{bmatrix} x \\ y \\ disparity(x,y) \\ 1 \end{bmatrix} . \tag{5.1}$$

To obtain the results as discussed in (2.23) and (2.24) the Q matrix needs to be constructed as follows:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -w/2 \\ 0 & 1 & 0 & -h/2 \\ 0 & 0 & 0 & f \\ 0 & 0 & 1/B & 0 \end{bmatrix} , \tag{5.2}$$

where w and h are the images' width and height, f is the cameras focal length and B is the baseline, which in AirSim is natively set to 0.25 [cm]. But we can exploit this method further by left-multiplying the Q matrix with the homogeneous transform $\mathbf{T}_c^w$ that encodes the coordinate transform from the camera reference frame to world reference frame:

$$Q_{new} = \mathbf{T}_c^w Q \, . \tag{5.3}$$

Now this method can be used to directly acquire an accurate 3D point cloud ground-truth data from the AirSim environment. This functionality is implemented in the *airsim_env.py* file as the *get_cloud* method. The resultant point cloud representation of the environment can be seen in figure 5.1.

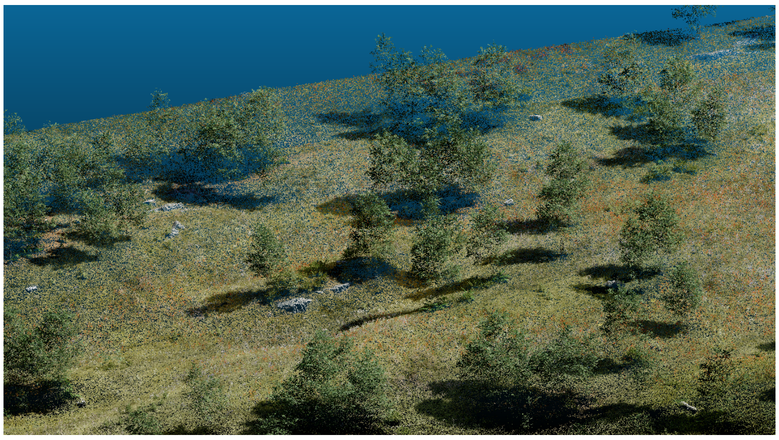


Figure 5.1: Point cloud from AirSim

5.1.3 Simulating lens distortion

Since most real world cameras can not be modelled by a simple pinhole camera, it is useful to have lens distorted images generated from the simulation. To distort the ideal pinhole images from the simulation, the brown camera model (Sec. 2.2.3) can be used. However the equations (2.21) only allow us to calculate, where a specific ideal image point is mapped into the distorted image. We, on the other hand, need to know where a specific distorted image point has its pre-image in the pinhole camera image. Therefore we need an inverse relationship to (2.21), for which no analytical form is known. It is therefore necessary to invert this formula for each pixel on the distorted image individually using non-linear optimization. The resulting ideal image points' coordinates can then be used to resample the ideal image into the distorted one using interpolation.

Luckily, the OpenCV library already provides this functionality in the *undistortPointsIter* method. This method accepts the intrinsic matrix, distortion coefficients and the distorted image points list. It then outputs a list of corresponding ideal image points, which can be used to create a map (look-up table) that can be supplied to the OpenCV *remap* method along with the ideal image. This method performs the resampling and finally outputs the distorted image. Note that as a result of the interpolation, some detail in the images can be lost. A solution to this would be implementing the distortion directly in the Unreal Engine. The distortion functionality is implemented in the *distortion.py* file via the *Distorter* class.

5.2 Photogrammetry data processing

While the used photogrammetry software is implemented in ready-to-use external libraries, the data that goes in and comes out of them needs to be correctly handled. Since resultant 3D models are used for navigation, it is necessary to geo-reference them, that is, they need to be related with the coordinate system used for navigation. The implementations offer this functionality assuming the cameras' coordinates are provided. Unfortunately, they are not completely consistent in the format they use for the input cameras' coordinates or the output 3D models' coordinates. Coordinate systems used by the individual pipelines are listed in table 5.1. Since our simulated world already comes with its own cartesian coordinate system, it is natural to use this coordinate system for navigation as well. The simulation origin is assigned with some arbitrary point in the WGS coordinate system, which enables transforming the simulation x, y, z coordinates into the geographic coordinate frames used by the implementations.

Method	**Input coordinates**	**Output coordinates**
COLMAP	ENU	ENU
ODM	WGS	ENU
OpenMVG + OpenMVS	WGS	ECEF

Table 5.1: Coordinate frames used by the photogrammetry implementations

5.2.1 COLMAP

COLMAP accepts the camera poses as a simple text file with rows containing the image names and their x, y, z coordinates with respect to some cartesian coordinate frame. The extraction of these coordinates from the simulator is implemented in the *airsim_env.py* file. In a real-life scenario sometimes only the WGS latitude, longitude and altitude coordinates are available and need to be transformed into an ENU coordinate system first in order to be used by COLMAP. This functionality is implemented in the file *geo.py* borrowed from another open-source SfM project OpenSfM [17]. The COLMAP pipeline is executed using the Python script *colmap_pipeline.py*. It simply executes each stage of the photogrammetry pipeline (table 4.2). Most of the parameters are set on default values. The intrinsic parameters can be easily specified in the first stage of the pipeline by two arguments for the camera model used and its parameters. When done processing, COLMAP outputs a point cloud which is referenced in the same cartesian coordinate frame as the cameras and can be then directly used for further processing.

5.2.2 OpenDroneMap

ODM automatically searches the images' EXIF[3] tags for GPS data in the WGS format and if found, uses those to align the resultant point cloud with the GPS coordinate system. To transform x, y, z coordinates from the simulation into GPS coordinates, the method *topocentric_from_lla* in the *geo.py* file is used. The insertion of GPS data in the WGS format into the EXIF tags is implemented in the *save_image_gps* method in *data_extraction.py* file.

[3]EXIF (Exchangeable image file format) is the metadata inserted directly into images. It can store information about the GPS coordinates, date or camera settings.

The output point cloud of ODM is already in a the desired ENU format, but its origin is located at the mean coordinates of all the cameras at 0 altitude. It needs to be shifted in order to align it with our worlds ENU coordinate frame. This can be easily done by applying a homogeneous transform to the point cloud.

5.2.3 openMVG + openMVS

The openMVG, just like ODM, automatically searches for GPS coordinates in the WGS format in the EXIF tags of the supplied images. The output point cloud is however created in the ECEF coordinate system. To transform between the ECEF and ENU frames, a simple homogeneous transformation can be used. The *ecef_from_topocentric_transform*method in *geo.py* computes the matrix for opposite transformation, so we only need to invert it and apply to the output point cloud to align it with our ENU frame.

5.3 Preprocessing for path planning

It is possible to use the acquired point cloud directly for path planning, though it would be very computationally expensive and would be feasible only for small scale environments. The 3D data structure is also redundant, since we capture the scene using a camera from far above and thus can only recover the surface of the environment. It is therefore preferable to use a simpler two dimensional heightmap[4].

5.3.1 Heightmap generation

A suitable library for heightmap generation is the Point Data Abstraction Library (PDAL) [18]. PDAL is written in C++ and we are going to access its functionality through the Python API it offers. We are going to exploit the *PDAL.Pipeline* class which is capable of creating a raster (grid of pixels) from a point cloud file. All it requires is the location of the input point cloud file and parameters for the raster generation (shown in table 5.2). It works by dividing the point cloud into a grid of cells with respect to the specified axis. Then for each cell it computes its value based upon which type of output is chosen. All these values are then stored into an output TIFF image file, which enables storing floating point numbers with minimal loss of precision. We will use the max raster, which for each grid cell takes the maximum height value of all the points inside it. This approach avoids underestimating the terrain height during navigation, which could lead to a collision.

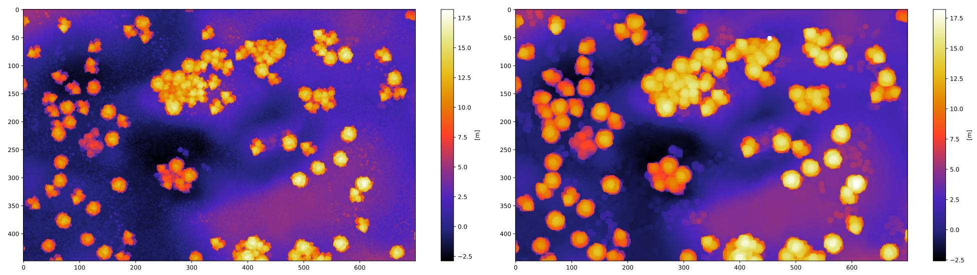
Growing the heightmap

If only the elevation directly below the drone would be taken into account, there would be a risk of the drone crashing due to its non-zero dimensions.

[4]Heightmap is a grid of height values usually in the form of a 2D floating point array.

Argument	Function
output_type	determines how the value for the raster is obtained (e.g. min, max, average,...)
resolution	determines the tile size
origin_x & origin_y	determine the raster's origin world coordinates
height & width	determine the number of rows and columns of the raster

Table 5.2: PDAL Pipeline arguments



(a) : Original heightmap **(b) :** Grown heightmap

Figure 5.2: Heightmap and its grown version

Therefore neighbouring values some well chosen radius r need to be considered during the path planning. We can easily incorporate this right into the heightmap by iterating over the tiles and for each one, assigning the maximum value of its neighbours in the given radius r into its new value. While planning on this "grown" heightmap, we can consider the drone to be of zero size. The footprint of the drone we use in our simulation is a square with side length of 1 meter. A good radius to choose for heightmap growing is therefore $r = \sqrt{0.5}$ meters as this is the radius of a circumscribed circle of this square. To convert this radius into tile radius, we divide it by the resolution of the heightmap. Since the point cloud obtained with photogrammetry tends to be noisy, a larger value can be used in order to suppress noise at the cost of lost information. An example of a heightmap and its grown version is shown in figure 5.2.

5.3.2 Occupancy grid generation

Using the heightmap we could in theory navigate across the terrain very easily using a straight line between two points in the xy-plane and simply fly higher than all the obstacles in the way. However there can be some constraints to the path we want to take. The information about where the agent can or cannot go is commonly encoded in so called **occupancy grid**. Occupancy grid is a 2D or 3D binary grid, which represents a map of obstacles in the navigated environment. For our purpose it will be a 2D raster of the same size as the heightmap and will have a value of 1, where there is an obstacle and a value of 0, where there is not. This functionality is implemented in the file *dem_handling.py*.

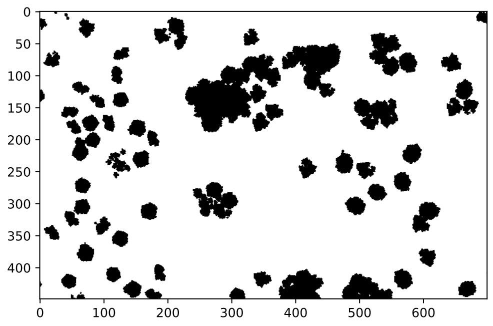


Figure 5.3: Absolute height occupancy

Absolute height restriction

The most basic method to build the occupancy grid is to specify a minimum and a maximum possible terrain elevation, that we want the UAV to have access to. Our obstacle grid has value 1 for every position on the grid where the elevation is not inside these bounds. En example of such occupancy grid is shown in the figure 5.3, where the obstacles are shown in black. When comparing this occupancy grid to the heightmap on figure 5.2, we can see that the obstacles are generated at the highest points of the terrain, which in this example are trees that are taller then the chosen threshold.

Relative height restriction

Another criterion that can be used for the occupancy grid generation is relative height difference between adjacent cells of the grid. Obstacles are placed where the terrain changes too rapidly to avoid large jumps in the terrain. To generate the relative height information we can approximate the gradient of the terrain height with respect to u, v[5] coordinates using the Sobel filter [19]. To implement it, we first need to construct two normalized Sobel kernels for the u and v direction:

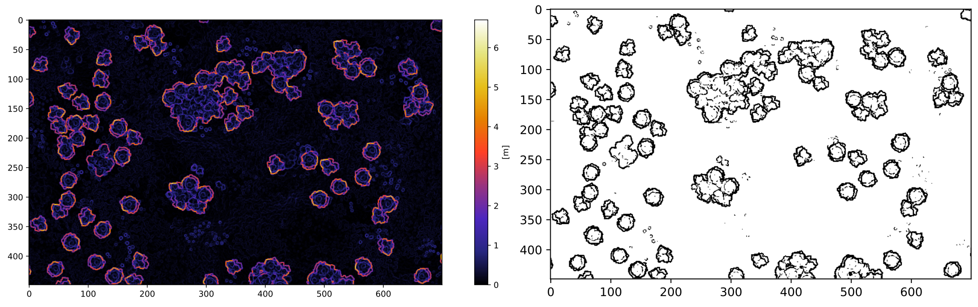
$$\mathbf{K}_u = \frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad \mathbf{K}_v = \frac{1}{8}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}. \tag{5.4}$$

These two kernels can than be used to compute the horizontal and vertical gradient components using 2D convolution with the heightmap:

$$\begin{aligned} \mathbf{G}_u &= \mathbf{K}_u * \mathbf{I}\,, \\ \mathbf{G}_v &= \mathbf{K}_v * \mathbf{I}\,, \end{aligned} \tag{5.5}$$

where $\mathbf{I}$ denotes our heightmap image, here treated as a matrix. $\mathbf{G}_u$ and $\mathbf{G}_v$ denote the matrices that store the horizontal and vertical components of the gradient for each image point respectively. The magnitude of the gradient

[5]right and down respectively as per the image coordinate convention



(a) : Gradient magnitude **(b) :** Relative height occupancy

Figure 5.4: Gradient magnitudes and corresponding occupancy grid

vector at position (u, v) on the heightmap can be computed as follows:

$$|\nabla \mathbf{I}(u,v)| = \sqrt{\mathbf{G}_v^2(u,v) + \mathbf{G}_u^2(u,v)}\,, \tag{5.6}$$

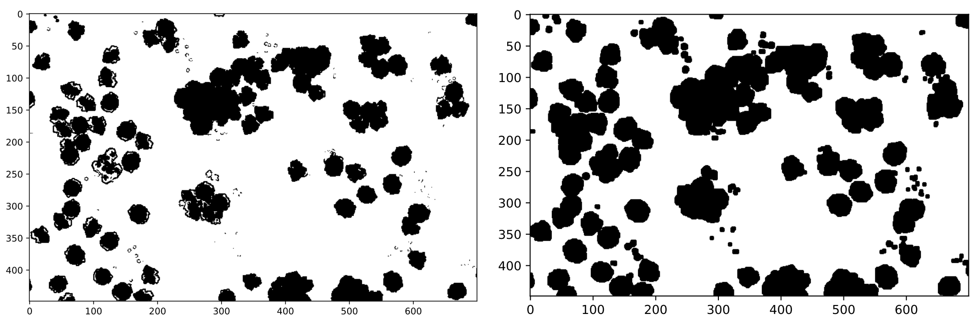
where the notation $\nabla \mathbf{I}(u,v)$ denotes the gradient approximation of the heightmap $\mathbf{I}$ at the coordinates (u, v). Note that since the kernels (5.4) are 3 by 3 matrices, we cannot compute the gradient values at the edges of our heightmap and we need to fill these in with zeros so we get a matrix of the same size as the heightmap. Once we have this gradient magnitude matrix computed, we can create another occupancy grid, where we put the value 1 (an obstacle) wherever the gradients magnitude exceeds specified threshold. The resultant gradient magnitudes and the generated occupancy map are shown in figure 5.4.

Combined occupancy

The absolute and relative thresholds need to be empirically set to some reasonable values and the resultant occupancies can them be combined together by a position-wise OR. It is possible to avoid using these occupancy grids in the path planning, but they are useful in that they can help us impose some restrictions into the path taken by the drone and speed up the navigation process. The resultant occupancy grid obstacles should also be grown similarly to 5.3.1 to enable neglecting the drone's size. We use the same radius for growing occupancy grid as for the heightmap. The combined occupancy grid and its grown version are shown in figure 5.5.

5.4 Path planning

The path is planned using the heightmap and occupancy grids, where only the positions at the centres of the tiles are considered. The planning algorithm takes the goal and start tiles as arguments and returns a set of coordinates of tiles through which the found path leads. These tile coordinates along with corresponding heightmap values can be easily transformed into world 3D point coordinates, which can be used for flight.



(a) : Combined Occupancy **(b) :** Grown combined occupancy

Figure 5.5: Combined occupancy and its grown version

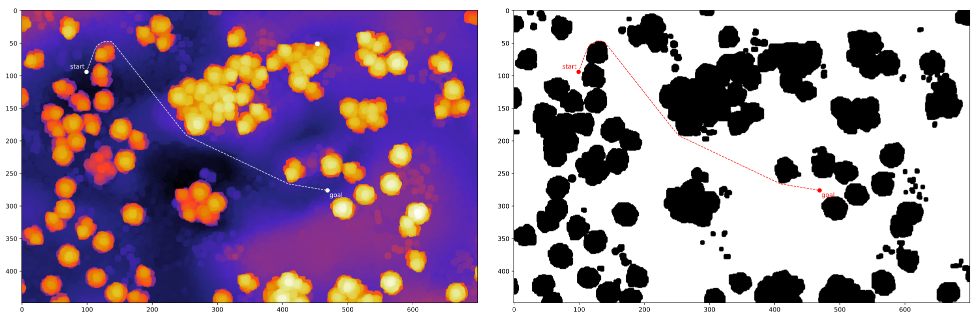


Figure 5.6: Path generated by Theta*

5.4.1 Theta*

Theta* [20] is an A*-based any-angle grid search algorithm. Any-angle meaning the final path is not restricted to the 45° angle increments as a typical A* on a grid would be. Instead it produces piece-wise straight trajectory through the environment, which is shorter, simpler and thus better suited for aerial vehicles. The Python pseudo-code 5.7 depicts the used Theta* algorithm implemented in the *thetastar.pyx* file. An example of a path generated by this algorithm between two points is shown in the figure 5.6.

```
def thetastar(Start, Goal):
    if Start == Goal:
        return None
    minheap.push(Start, 0) # push onto the heap
    while minheap is not empty > 0:
        current = minheap.pop()
        current.closed = True   # final visit
        if current == Goal:
            return path(current)  # goal has been reached
        for neighbour in neighbours(current):
            if neighbour.closed:  # skip
                continue
            if neighbour not in minheap:
                neighbour.gscore = Inf
            update_node(current, neighbour)
```

Figure 5.7: Theta* pseudo-code

The node objects used in this pseudo-code (*Start*, *Goal*, *current* and *neighbour*) represent the nodes of the search graph, or in our case tiles on a grid. Each node contains information about the smallest found cost (distance from start) *gscore*, the *parent* node (node from which this node was reached with the smallest cost) and whether it is *closed* (the cost is final). The *minheap* object is a binary min heap , with *push* method adding the specified node onto the heap with specified value and *pop* method returning the node with the smallest value and removing it from the heap. The *neighbours* method returns a list of neighbouring nodes of the specified node (8 neighbours in our case).

At first glance, the algorithm 5.7 looks just like regular A*, but the main difference lies in the *update_node* method which first checks for line of sight between the *current* node's parent and the *neighbour* node. If there is line of sight, than the algorithm moves on a straight line between the *neighbour* node and the *parent* of the *current* node. In this way, Theta* avoids moving in a zig-zag pattern on the grid as would be the case with regular A*, making the trajectory piece-wise straight. The pseudo-code of this function is shown in figure 5.8.

```
def update_node(c, n):
    if line_of_sight(c.parent, n):
        new_gscore = c.parent.gscore + cost(c.parent, n)
        if new_gscore >= n.gscore:
            return None
        n.parent = c.parent
    else:
        new_gscore = c.gscore + cost(c, n)
        if new_gscore >= n.gscore:
            return None
        n.parent = c
    n.gscore = new_gscore
    fscore = new_gscore + heuristic(n)  # final score
    if neighbour in minheap:
        minheap.remove(n)
    minheap.push(n, fscore)
```

Figure 5.8: Update node method pseudocode

Line of sight

The *line_of_sight* method works by constructing a line connecting two points slightly above the two tiles on the heightmap (e.g. 1 meter above the ground). It then checks each heightmap tile between these two nodes for collision, which occurs if the tile is above the line, or if there is an obstacle at the tile (in the occupancy grid). A line slightly above the ground has to be considered, because otherwise there would be an intersection every time. Which tiles to check between the two boundary nodes is selected using the Bresenham line rasterisation algorithm [21]. If there are no collisions found, the line of sight between these nodes exists. This method can also be easily configured to restrict the maximum altitude above the terrain detecting if the line is too high above the heightmap.

Cost and heuristic functions

Both the cost and heuristic methods return the 3D euclidean distance between the two tiles in the heightmap. In the cost function, if the height difference between these two tiles exceeds specified threshold, it is additionally multiplied by some factor before calculating the distance. In this way, the algorithm will avoid large altitude changes between tiles i.e. steep terrain, but if necessary, will not be restricted from taking these paths.

Chapter 6
Experiments

The chosen implementations are first compared in two experiments on virtual datasets obtained through the AirSim simulator, where an accurate ground truth point cloud is available. The first experiment was done without any distortion, considering only a simple pinhole camera model with precisely known parameters and camera positions. This experiment allows us to see how the software performs under ideal conditions. For the second experiment, in order to assess the robustness of selected implementations, simulated real world disturbances are introduced. The best implementation is combined with the path planning algorithm into a complete navigation system.

6.1 Comparison of methods

Hardware

All experiments were conducted on a Laptop PC equipped with an Intel Core i7-8750H (6x 2.20 GHz), 24 GB RAM and an Nvidia GTX 1050 Ti.

Comparison criteria

Since our goal is to navigate through terrain without crashing into it, accuracy is the most important criterion. For accuracy evaluation, the absolute difference between the output heightmaps and the ground truth heightmap is considered. The distribution of this absolute difference is described using its maximum, mean, median and 95th percentile. To take into account the coverage of the heightmaps the missing data percentage shows, how many tiles lack height information due to poor reconstruction.

The number of points of the generated point clouds together with total runtime of the methods are also mentioned. Although these two metrics do not reveal much about the accuracy, the number of points expresses the ability to recover data from the images and runtime has practical significance. Let these two numbers serve as a heuristic measure where the error statistics are not sufficient to decide which method to prefer over the others. Note that the runtime is only meaningful to compare between ODM and OpenMVG

+ OpenMVS (OMVG + OMVS), since these both run exclusively on CPU, whereas Colmap utilizes GPU.

Graphs

The error is visualised using histograms and distributions of the absolute error. These graphs were constructed while dividing the shown 1.5 meter error range into 300 bins. The heightmaps are visualised using colors to give an idea of how the heightmaps look and how they differ between the implementations.

The white color in these heightmaps represents missing data, that the algortihms failed to recover from the photos. The big white square seen on all the heightmaps including the ground truth was purposely omitted, because the area was too hard to reconstruct for all the implementations and needlessly distorted the experiments' results. Finally the error maps visualise how the absolute error is distributed throughout the map.

Parameters

Most parameters of the implementations are left on default values. The only non-default parameters are the settings of the Colmap model aligner method which have to be specified to enable georeferencing and were set as shown in table 6.1.

parameter	value
--robust_alignment	1
--robust_alignment_max_error	0.001

Table 6.1: Model aligner parameters

For both experiments, the photos are acquired at an altitude of 40 meters above the ground level due to constraints dictated by the Unreal Engine platform. Overlap of 80 % and sidelap of 70 % between images (measured at 0 meters altitude) was used as recommended in [22]. The area of interest is 150 meters wide and 350 meters long and is determined by the size of the virtual environment. The sampling resolution for the heightmap generation is set to 0.3 meters, which results in a 500x1166 heightmap. This value of resolution was determined heuristically, because it simply works well for navigation and does not take too long to compute.

6.1.1 Experiment without distortion

The first experiment was done using the AirSimCV class and uses precisely set positions of cameras. No distortion or noise were simulated and only a pinhole camera with exactly known parameters is used.

A total of 264 images of the terrain were captured and used in this experiment. The heuristic metrics and the statistics are shown in tables 6.2, resp. 6.3. The figures 6.1 and 6.2 show the absolute error histogram and distribution respectively. The figure 6.3 shows the generated heightmaps and the figure 6.4 visualizes the heightmaps' absolute error.

software	runtime [minutes]	points [millions]
Colmap	95	4.155
ODM	55	18.760
OMVG + OMVS	**36**	**19.287**

Table 6.2: Heuristics without distortion

software	missing	max	mean	median	P_{95}
Colmap	6.78 %	18.29 m	**0.71** m	**0.22** m	**0.87** m
ODM	6.21 %	14.5 m	1.04 m	0.23 m	2.36 m
OMVG + OMVS	**4.79** %	**14.44** m	0.83 m	0.23 m	1.19 m

Table 6.3: Error statistics without distortion

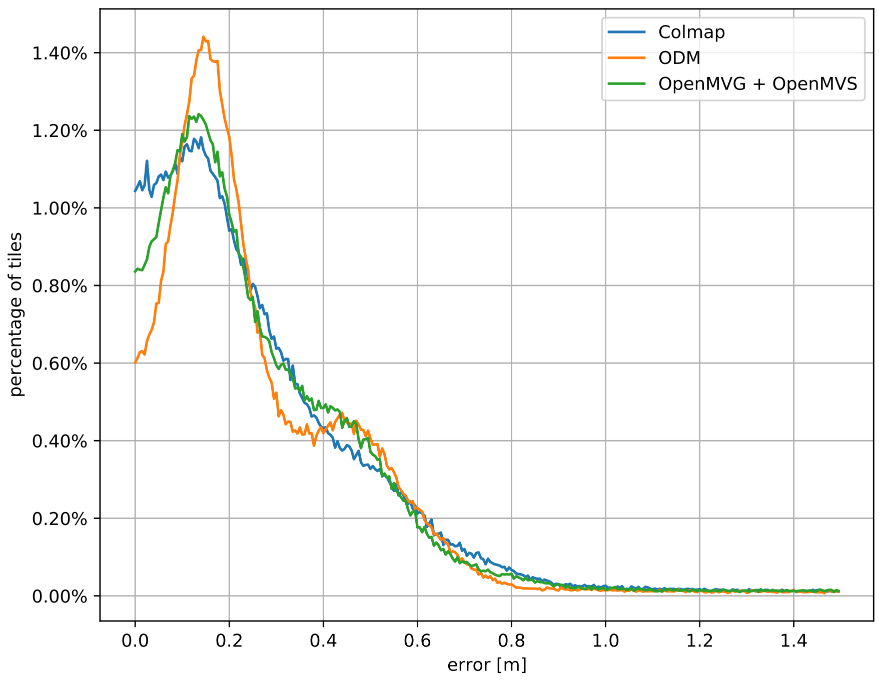


Figure 6.1: Error histogram without distortion

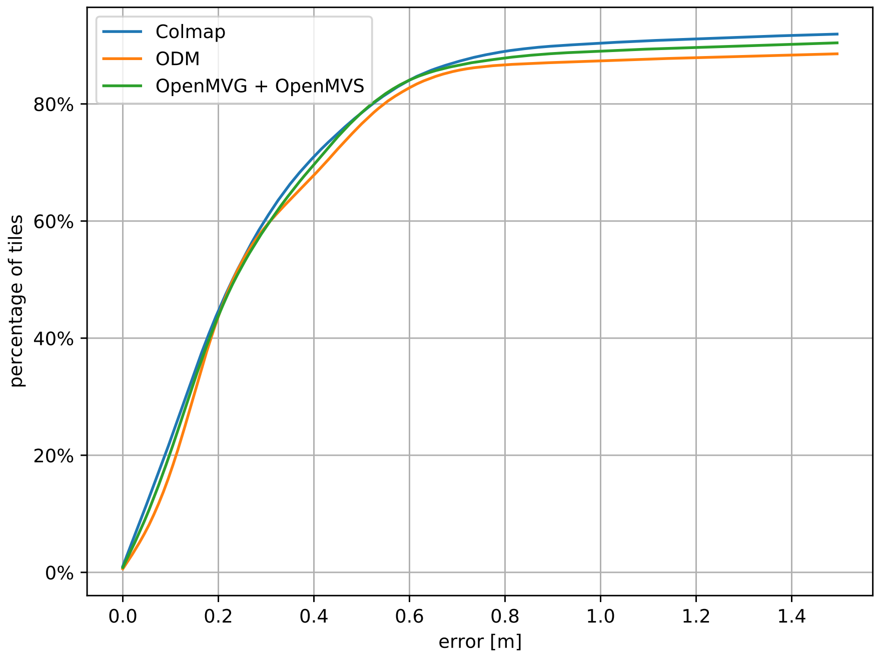


Figure 6.2: Error distribution without distortion

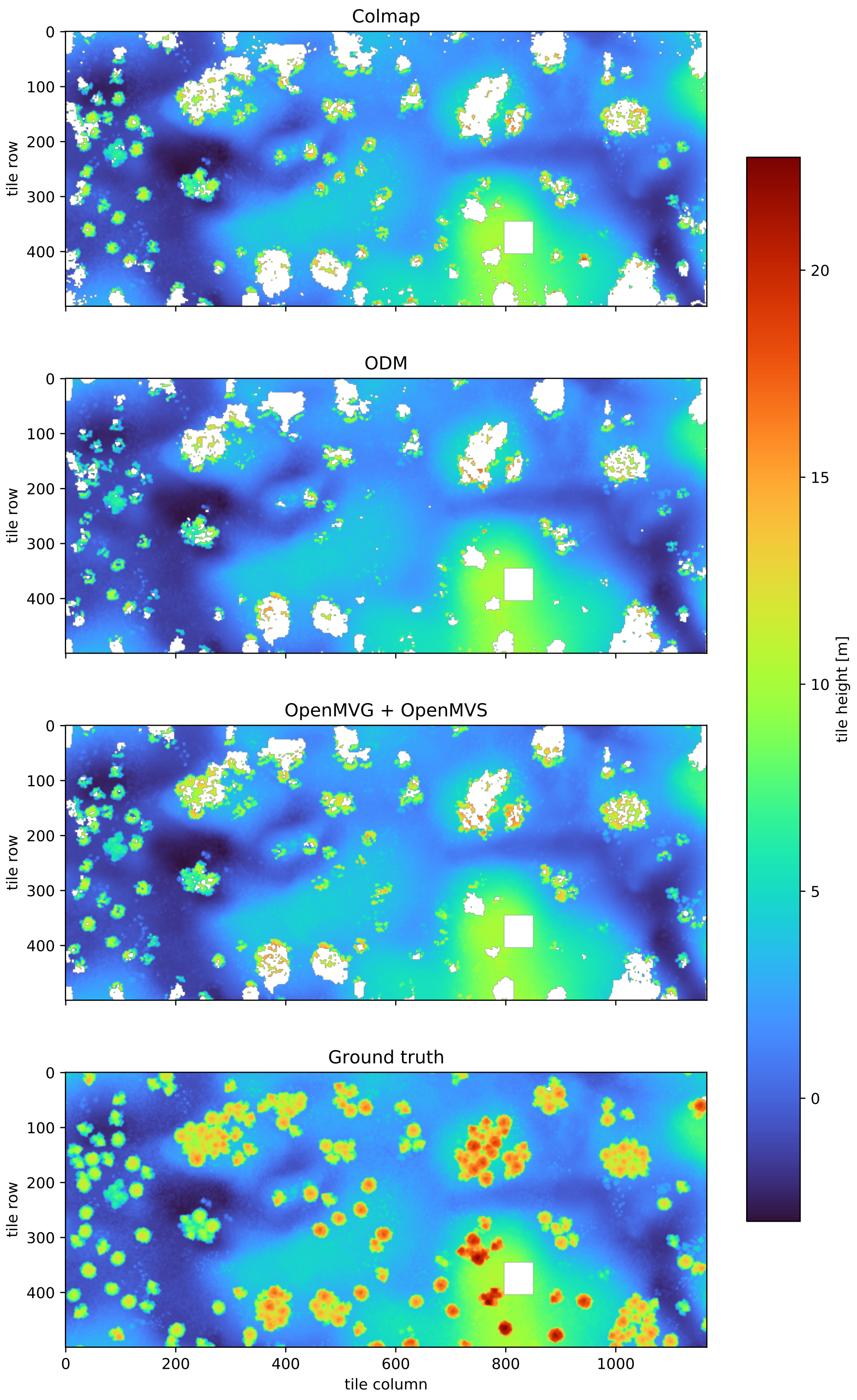


Figure 6.3: Heightmaps without distortion

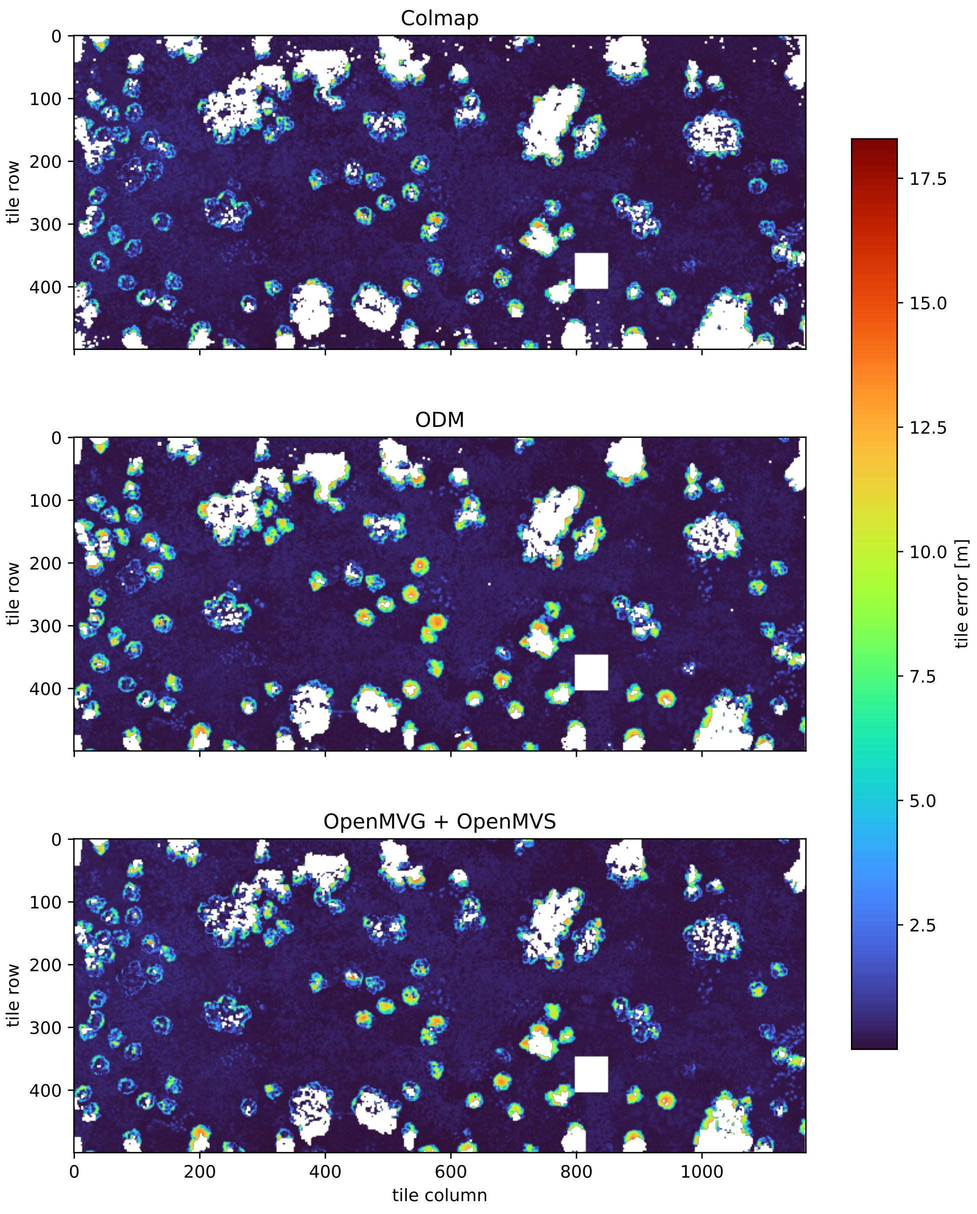


Figure 6.4: Error map without distortion

6.1.2 Experiment with distortion

The first experiment was done using the AirSimUAV class and so the trajectory was flown by the drone inside simulation autonomously with some uncertainty. In this experiment motion blur, clouds, random camera sensor noise and camera lens distortion were enabled.

A total of 259 images of the terrain were captured and used in this experiment. The camera parameters were for realistic results obtained using a simulated calibration with virtual checkerboard pattern. The resultant parameters are then directly fed into the mapping software together with the images and their coordinates. The Colmap implementation is missing from this part of the experiment, since it failed to produce any result. The heuristic metrics and the statistics from this experiment are shown in tables 6.4, resp. 6.5. The figures 6.5 and 6.6 show the absolute error histogram and distribution respectively. The figure 6.7 shows the generated heightmaps and the figure 6.8 visualizes the generated heightmaps' absolute error.

software	runtime [minutes]	points [millions]
Colmap	-	-
ODM	42	**16.996**
OMVG + OMVS	**29**	15.658

Table 6.4: Heuristics with distortion

software	missing	max	mean	median	P_{95}
Colmap	-	-	-	-	-
ODM	**12.74** %	**14.49** m	**0.76** m	**0.13** m	0.86 m
OMVG + OMVS	17.57 %	14.68 m	0.78 m	0.18 m	**0.81** m

Table 6.5: Error statistics with distortion

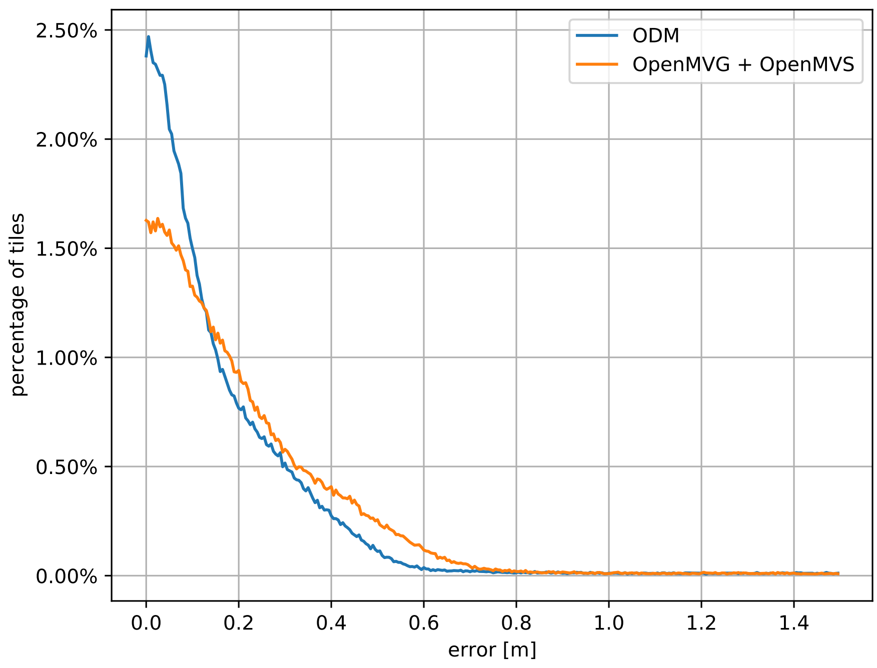


Figure 6.5: Error histogram with distortion

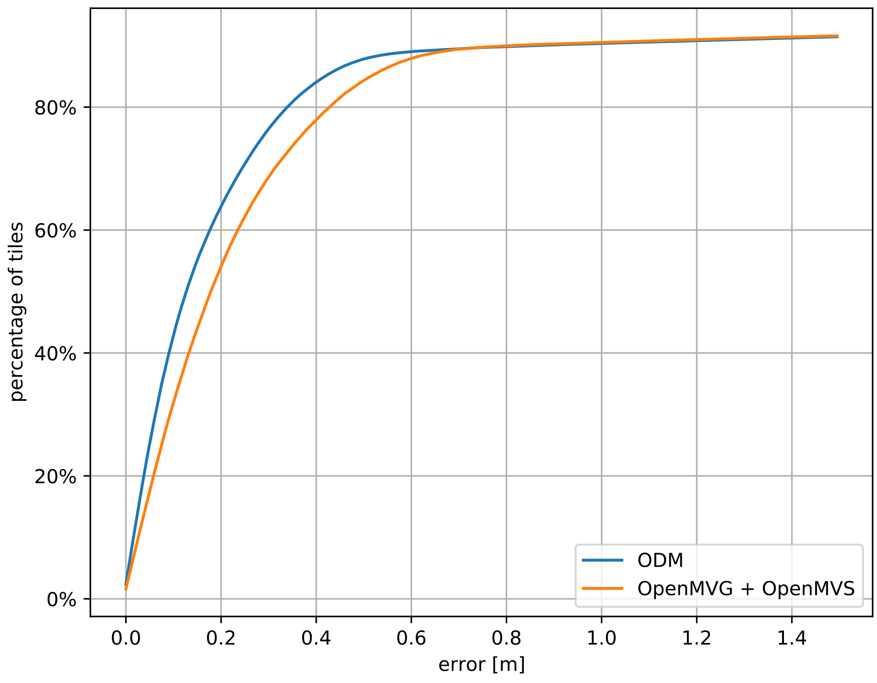


Figure 6.6: Error distribution with distortion

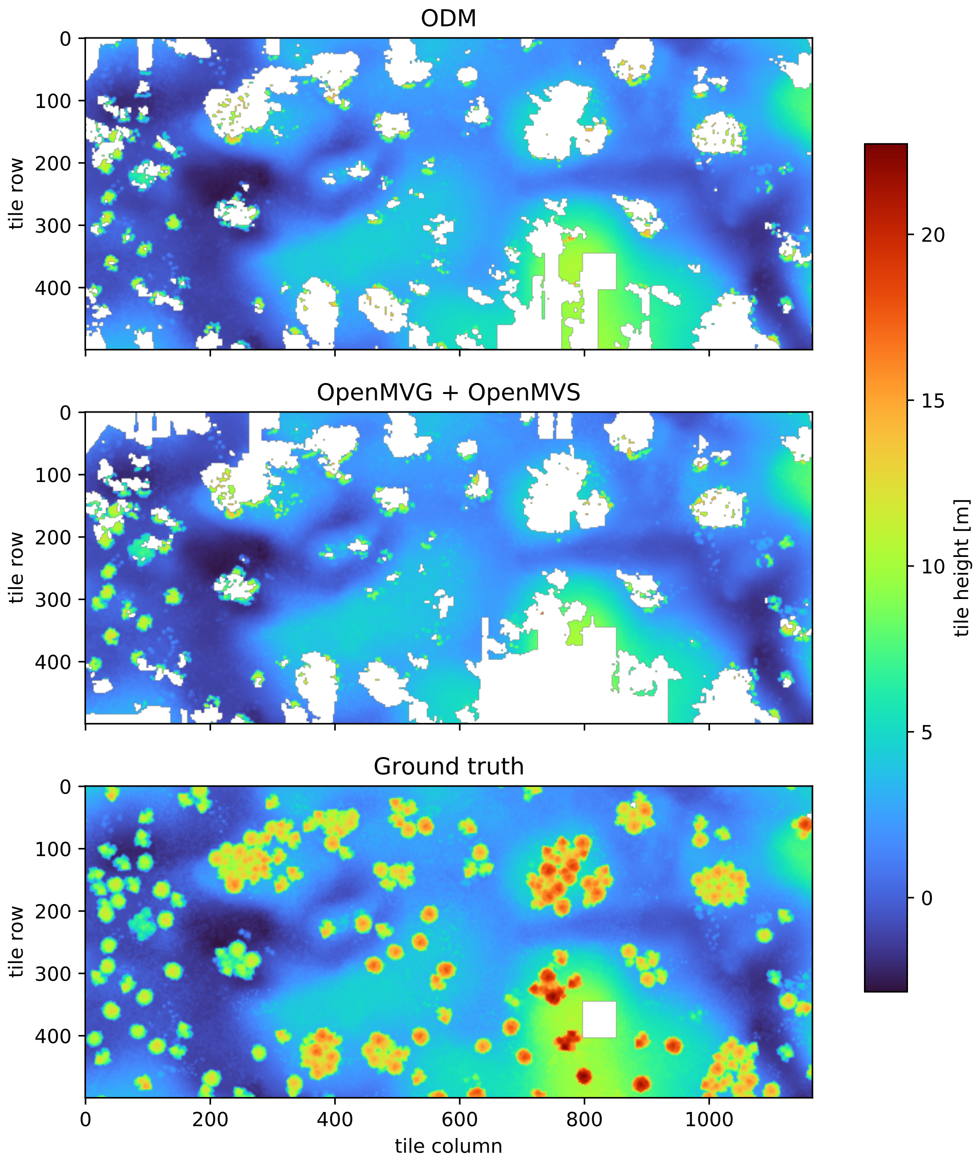


Figure 6.7: Heightmaps with distortion

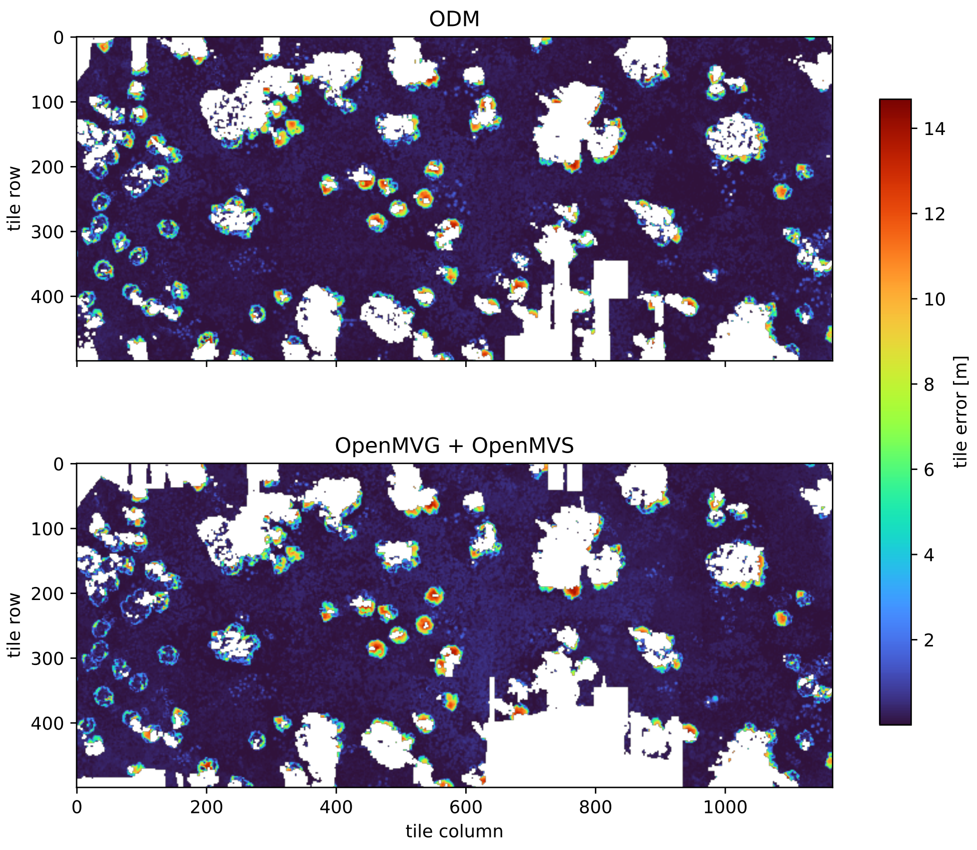


Figure 6.8: Error map with distortion

6.1.3 Comparison discussion

In the ideal conditions Colmap performed the best in terms of raw accuracy of the data points it provided. However, it produced far less data points than the other methods, which could be unfavourable in certain applications, where dense covering is beneficial. For our navigation purposes, the amount of data is sufficient and accuracy is way more important and thus Colmap would be the software of choice if we operated in ideal conditions. Unfortunately, in simulated real life conditions, Colmap failed to produce any result. ODM and OMVG + OMVS produced similar amounts of data in both experiments, though ODM was about 50% slower. In the ideal conditions, ODM had less coverage and slightly worse 95th percentile of error. Under simulated distortion, ODM had far better coverage and negligibly better accuracy than OMVG + OMVS. Although ODM generated more data, OMVG + OMVS seems to be the best choice for its high accuracy in both experiments and also faster runtime compared to ODM.

What is worth mentioning are the large missing areas on all of the heightmaps generated with photogrammetry, especially in the second experiment. Photogrammetry is notorious for having problems with shadows, movement and thin, small objects. Majority of the missing values in the first experiment are due to the trees' thin branches, which are hard to reconstruct due to all three of these effects. These tree branches are also higher up and thus were captured in fewer images - again complicating the reconstruction. In the second experiment, the clouds' shadows confused the photogrammetry even more resulting in larger spots of missing data. The error maps in both experiments show that the areas with the highest error are also concentrated around the trees. This again has to do with the fine details of the branches that are hard to recover from the photos. This is why it is recommended to capture the images under uniform lighting, without wind and large enough overlap between the images for best results in practice. Also two passes of the drone above the terrain are preferably done at different altitudes with a slightly tilted camera in the second pass [22] for more accurate results. However this was not possible, due to limited computational power used.

6.2 Final navigation system

The OpenMVG + OpenMVS pipeline together with the appropriate preprocessing for planning as discussed in 5.3 can be accessed through the *preprocess.py* script. It takes in a folder with georeferenced images and outputs the grown heightmap together with the corresponding occupancy grid needed for navigation. The navigation stage as discussed in 5.4 can be accessed using the *navigate.py* script, which takes start and goal world x, y coordinates as arguments and outputs shortest path found. The whole navigation system diagram is illustrated in figure 6.9.

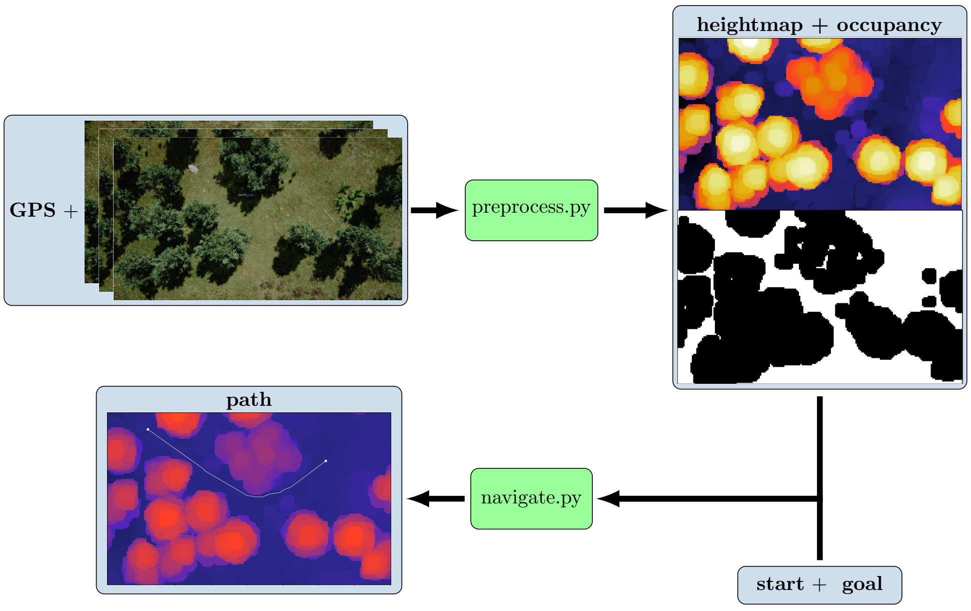


Figure 6.9: Navigation system scheme

6.2.1 Performance evaluation

Images from the distortion experiment were used for the following experiment together with the same parameters. The resulting heightmap from preprocessing stage can be seen in figure 6.10 and the generated grown heightmap and occupancy grid together with and example of a path are shown in figure 6.11. In order to evaluate the system's performance in terms of navigation, a pair of points (start and goal) on the map was generated randomly and the system was used to generate a path between them. This path was then used to navigate the drone inside AirSim simulator between these points to check for collisions with the terrain. To fly on the path the method *move_on_path* from the *AirSimUAV* class was used. This procedure was repeated 500 times out of which only **20** flights (**4** %) ended in a collision. After investigation, most of the collisions were caused by running into small obstacles like tree branches, which are notoriously hard for photogrammetry to reconstruct.

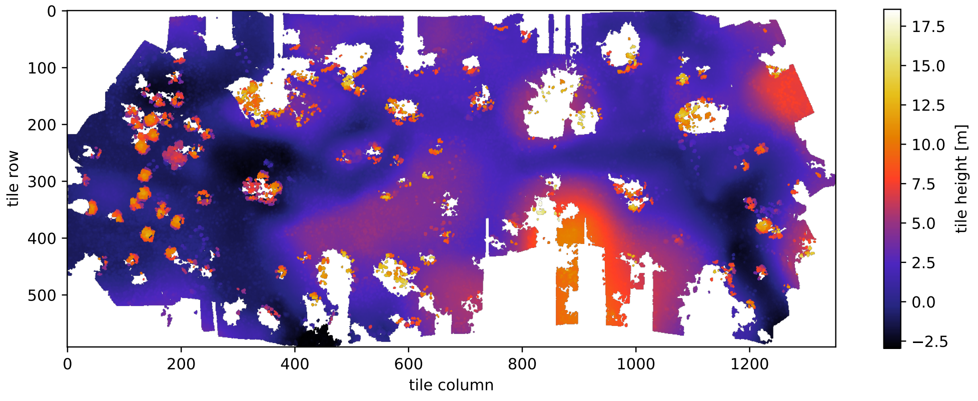


Figure 6.10: Generated heightmap

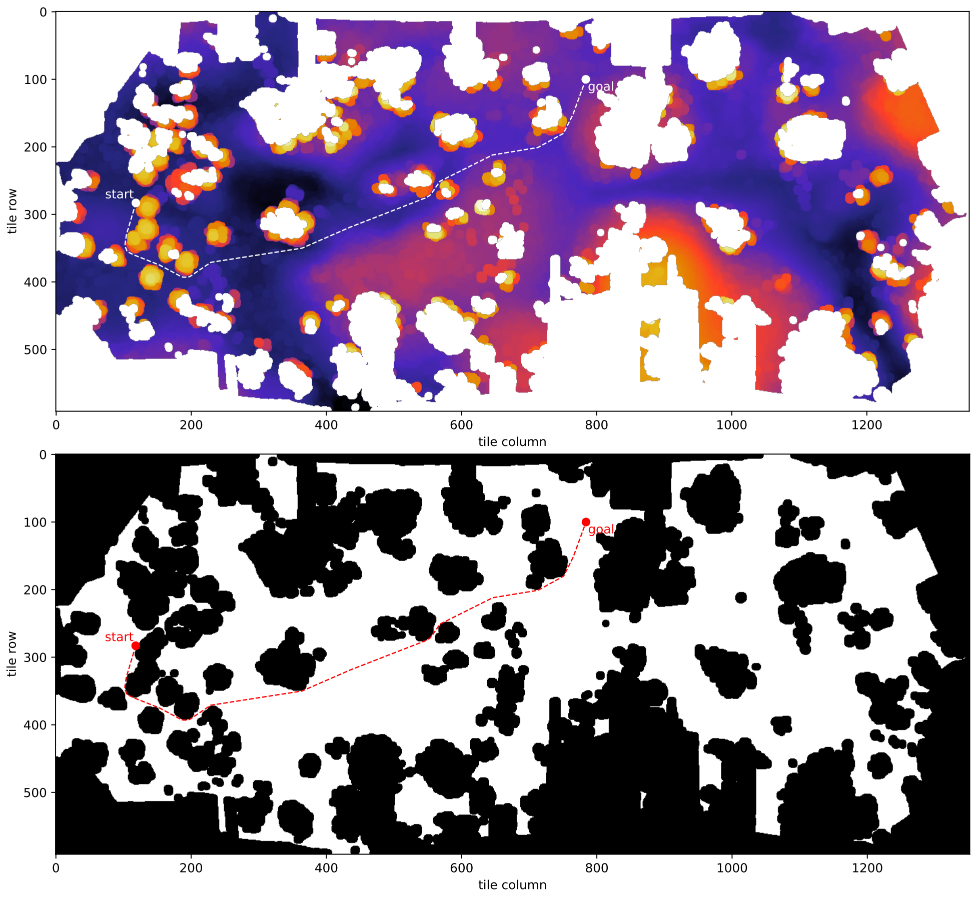


Figure 6.11: Generated path between two points

Chapter 7
Conclusions

7.1 Summary

In this thesis the use of photogrammetry methods for terrain mapping and low-altitude flight navigation was explored. First, the basic mathematical principles behind the camera model and 3D reconstruction were introduced together with a basic outline of how a common photogrammetry pipeline works. Then three suitable photogrammetry pipelines Colmap, Open Drone Map and OpenMVG + OpenMVS were selected for further comparison. As the development relied on a simulator AirSim, its capabilities, limitations and details about working with the underlying Unreal Engine were discussed.

In the practical part, the AirSim simulator interface was implemented, which enabled easier employment of the simulator as well as precise ground truth point cloud and distorted image acquisition. The steps that had to be taken to properly handle the data both for and from the photogrammetry software were mentioned. As the point clouds generated by the used photogrammetry implementations are not suitable for path planning purposes, a post-processing method was devised to transform them into an appropriate form. Since the terrain is mostly flat and only its surface is relevant, 2D heightmaps were chosen as the best solution to this problem. To enable the use for path planning algorithms, further processing steps were introduced. Two methods to define obstacles were implemented, one using the absolute height and the second using the height gradient of the terrain. Finally a path planning algorithm with desirable properties for drone navigation Theta* was implemented.

In the experiments, the photogrammetry implementations were compared using the simulated data. First experiment showed how the programs performed under ideal conditions. The second experiment was aimed at testing the robustness of these programs under simulated distortion such as moving cloud shadows and camera noise. Open Drone Map and OpenMVG + OpenMVS performed very similarly during the experiments, still the OpenMVG + OpenMVS was selected due to its faster runtime. This implementation was combined with the processing steps and path planning algorithm to form a complete navigation system, whose functionality has been tested using randomly generated paths through the environment. The system was found to be quite effective, with only a minority of generated trajectories ending in a collision with the terrain.

7.2 Future work

AirSim proved to be a versatile tool for visually oriented robotics development. However, the Unreal Engine is capable of much more and the fidelity of the simulation could be perfected given more time. Better quality assets, fine tuned lighting and more detailed environment could make it a more faithful representation of the real world and thus increase the value of the experiments.

Appendix A

Bibliography

[1] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2 edition, 2003.

[2] Duane C. Brown. Close-range camera calibration. *PHOTOGRAMMETRIC ENGINEERING*, 37(8):855–866, 1971.

[3] Richard Szeliski. *Computer vision algorithms and applications*. Springer, London; New York, 2011.

[4] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110.

[5] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395.

[6] OpenMVG contributors. Awesome 3D reconstruction list. `https://github.com/openMVG/awesome_3DReconstruction_list`, 2020.

[7] Peter Falkingham. free and commercial photogrammetry software review 2020. `https://peterfalkingham.com/2020/07/10/free-and-commercial-photogrammetry-software-review-2020/`, 2020.

[8] ODM Authors. OpenDroneMap [Computer software]. `https://github.com/OpenDroneMap/ODM`, 2017.

[9] Dan Cernea. OpenMVS: Multi-view stereo reconstruction library. 2020.

[10] Pierre Moulon, Pascal Monasse, Romuald Perrot, and Renaud Marlet. Openmvg: Open multiple view geometry. In *International Workshop on Reproducible Research in Pattern Recognition*, pages 60–74. Springer, 2016.

[11] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[12] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. Pixelwise view selection for unstructured multi-view stereo. In *European Conference on Computer Vision (ECCV)*, 2016.

[13] AliceVision. Meshroom [Computer software]. `https://github.com/alicevision/meshroom`, 2018.

[14] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[15] Epic Games [Computer software]. Unreal engine. `https://www.unrealengine.com`, 2020.

[16] OpenCV. Open source computer vision library, 2015.

[17] mapillary. OpenSfM [Computer software]. `https://github.com/mapillary/OpenSfM`, 2020.

[18] PDAL Contributors. Pdal point data abstraction library. `https://doi.org/10.5281/zenodo.2556738`, November 2018.

[19] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[20] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *J. Artif. Intell. Res. (JAIR)*, 39, 01 2014.

[21] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[22] ODM Authors. OpenDroneMap tutorial. `https://docs.opendronemap.org/tutorials.html#flight-pattern`, 2020.

Appendix B

DVD content

The contents of supplied DVD are shown in figure B.1.

```
DVD
├── README.md
└── navigation_system
    ├── calibrate.py
    ├── preprocess.py
    ├── navigate.py
    ├── sensor_width_camera_database.txt
    ├── RunDocker.cmd
    ├── Dockerfile
    └── tools
        ├── __init__.py
        ├── airsim_env.py
        ├── data_extraction.py
        ├── dem_handling.py
        ├── distortion.py
        ├── geo.py
        └── cython_files
            ├── setup.py
            ├── heap.pyx
            ├── heap.pxd
            └── thetastar.pyx
```

Figure B.1: DVD content