System Design and Real-Time Guidance of an Unmanned Aerial Vehicle for Autonomous Exploration of Outdoor Environments

Dissertation

zur Erlangung des akademischen Grades
Dr. rer. nat.
an der Fakultät für Mathematik,
Informatik und Naturwissenschaften
der Universität Hamburg

eingereicht beim Fachbereich Informatik von
Benjamin Adler
aus Hamburg

November 2014

Gutachter:
Prof. Dr. Jianwei Zhang
Prof. Dr.-Ing. H. Siegfried Stiehl

Tag der Disputation: 27. Februar 2015

Acknowledgements

Completing this PhD project and writing this document took almost 4 years. During this time, the character of the project often changed, allowing me to work with different people, read literature, create software and integrate hardware. As such, it was a very interesting and rewarding time, bringing about many failures as well as a few achievements.

First of all, I would like to thank my advisor Prof. Dr. Jianwei Zhang for being a constant source of guidance, support and criticism throughout these years. Also, Prof. Dr.-Ing. H. Siegfried Stiehl kindly accepted to review this thesis, taking a lot of time and giving helpful guidance in the process. Thank you, Prof. Stiehl!

Furthermore, I would like to express gratitude to my colleagues at TAMS, who provided a very productive environment in which ideas can evolve from "ridiculous!" to "works!". When everyone is keen to listen, laugh and help, this makes for a great place to learn.

Houxiang, thank you for giving me a great start!

Special thanks go out to you, Bernd, for your generous support, incredible patience and thoughtful advice. Also, thank you for always having time to hold the fishing rod!

Junhao, I am grateful for both our friendship and collaboration. Because of you, I got to see another culture, learn a tiny bit of Chinese history and eat a lot of great food!

Benjamin Adler

Abstract

The central endeavor of this thesis is the successful development and evaluation of an aerial mobile robotic system that produces precise, georeferenced, three-dimensional maps of outdoor environments by means of autonomous exploration.

The system consists of a ground station and a custom-built UAV with six degrees of freedom, featuring an on-board computer, an inertial navigation system, and two 2D laser range finders. In addition to a description of the hardware architecture and individual components being used, this dissertation presents challenges and problems that arose during the construction of its hard- and software as well as optimizations applied in the course of its development.

A fundamental part of this work is the distributed software architecture for in-flight sensor fusion and data analysis, with a focus on a novel, truly three-dimensional algorithm generating multiple next-best-views (NBVs). Designed for application on airborne platforms in outdoor environments, the approach works directly on raw, unstructured point clouds and can be used either indoors or outdoors with any sensor generating spatial occupancy information.

Based on the generated sensor-poses and the incrementally growing point cloud, trajectories are computed for the UAV to autonomously map its environment. To ensure safe operation, collision avoidance constantly monitors the planned path and updates it whenever obstacles are detected.

In order to satisfy real-time constraints, all algorithms are implemented on a highly parallel SIMD architecture found in modern GPUs, allowing for extremely fast motion planning and responsive visualization. As the underlying hardware imposes limitations with regards to memory access and concurrency, necessary data structures and further performance considerations are explained in detail.

Data has been captured during real, autonomous flights and is used to analyze the performance of all major components (flight controller, next-best-view generation, dynamic path planning and collision avoidance) of the system in realistic outdoor scenarios. The performance of the GPU-based next-best-view algorithm is also compared against a previous, CPU-based proof of concept.

Kurzfassung

Die vorliegende Dissertation berichtet über die Entwicklung eines luftgestützten Robotersystems zur autonomen Erkundung von Außenumgebungen mit dem Ziel der Erstellung präziser, georeferenzierter Karten.

Das vorgestellte System besteht aus einer Bodenstation sowie einer eigens konstruierten Drohne, welche einen Computer, ein Navigationsystem und zwei Laserscanner mitführt. Neben einer Beschreibung der Hardwarearchitektur und ihrer Komponenten werden Herausforderungen und Einschränkungen bei Systemintegration und Softwareentwicklung, sowie Optimierungen im späteren Verlauf der Entwicklung präsentiert.

Ein zentraler Teil der Arbeit ist eine verteilte Softwarearchitektur zur Fusion und Analyse von Sensordaten im Flug, wobei hier der Fokus auf einem neuartigen Algorithmus zur Erstellung von *next-best-views* aus bereits erstelltem Kartenmaterial liegt. Obwohl zur Anwendung auf luftgestützten Plattformen zur Kartographie von Außenumgebungen mittels LIDAR erdacht, ist der Algorithmus direkt auf jegliche räumliche Sensordaten (z.B. von sämtlichen Laserscannern, RGBD- und Time-of-Flight-Kameras) anwendbar. Basierend auf den erzeugten Sensorpositionen und dem aktuellen Stand der Karte erzeugt das System sichere Flugrouten zur weiteren Kartierung des Gebiets und überwacht diese auf Hindernisse.

Um einen Einsatz in Echtzeit zu ermöglichen, sind die Algorithmen auf parallel arbeitenden GPUs implementiert. Dies ermöglicht nicht nur schnelle Flugplanung, sondern auch effiziente und interaktive Visualisierung der aufgenommenen Daten und der Arbeitsweise der eingesetzten Algorithmen. Da die zugrundeliegende Hardware besondere Anforderungen an Speicherzugriffsmuster und Nebenläufigkeit stellt, werden Datenstrukturen und weitere Überlegungen zur Leistungssteigerung im Detail erläutert.

Weiterer Bestandteil der Arbeit ist eine Analyse von Daten aus echten Flügen in Außenumgebungen zur Beurteilung der Anwendbarkeit und Leistungsfähigkeit aller verwendeten Komponenten (Flugsteuerung, Erzeugung von *next-best-views*, Pfadplanung und Kollisionsvermeidung). Ferner wird das Laufzeitverhalten des GPU-basierten *next-best-view* Algorithmus mit dem eines CPU-basierten Prototyps verglichen.

Contents

List of Figures

List of Tables

Glossary

AABB Axis-Aligned Bounding Box

AoS Array of Structs

CRS Coordinate Reference System

DARPA Defense Advanced Research Projects Agency

DEM Digital Elevation Model

DSM Digital Surface Model

ECEF Earth-Centered, Earth-Fixed coordinate system

EKF Extended Kalman Filter

FOG Fiber-Optic Gyro

FoV Field of View

GNSS Global Navigation Satellite Systems

GPS Global Positioning System

GPU Graphics Processing Unit

ICP Iterative Closest Point

IERS International Earth Rotation Service

IMU Inertial Measurement Unit

INS Inertial Navigation System

IRM International Reference Meridian

ISA Instruction Set Architecture

LIDAR Light Detection and Ranging

LLA Latitude, Longitude, Altitude

LRF Laser Range Finder

MEMS Microelectromechanical System

MIMO Multiple-Input, Multiple-Output

MTA Multiple Time Around

NAVSTAR Navigation System using Timing and Ranging

NBV Next Best View

PID Controller proportional-integral-derivative controller

PPS Pulse Per Second

PRNG Pseudo-Random Number Generator

PTU Pan-Tilt Unit

PVT Position, Velocity, Time

RGBD Red, Green, Blue, Depth (the channels of a 3D color camera)

RTK Real-Time Kinematic

SHT Spatial Hash Table

SLAM Simultaneous Localization And Mapping

SoA Struct of Arrays

SRS Spatial Reference System

TEC Total Electron Content

ToF Time of Flight

TSP Traveling Salesman Problem

UAS Unmanned Aircraft System

UAV Unmanned Aerial Vehicle

UTM Universal Transverse Mercator

VBO Vertex Buffer Object

ZUPT Zero-Velocity Update

Chapter 1

Introduction

1.1 Motivation

The central topic of this thesis is the convergence of robotics and geodesy in the context of unmanned aerial vehicles. While the robotics is a rather young discipline, geodesy has been practiced for as long as mankind exists. The following paragraphs shall briefly introduce both disciplines, eventually motivating the work carried out in this thesis.

The field of robotics was named after a term that first appeared in "Rosumovi Univerzální Roboti" (R.U.R.), a science-fiction-play of Czech writer Karel Čapek. Premiered in 1920, R.U.R. tells the story of human-like, but artificial creatures that are made in a factory and sold in order to take over hard, laborious tasks from their human owners. Since *robotā* is the Slavic word for "hard work" and "slavery", Karel's brother Josef christened these creatures *roboti*. The play itself did not focus on technical details, but mainly addressed the ethic issues arising from the existence of a large population of roboti living amidst humans. It quickly became very popular and was translated to more than thirty languages within three years. The English translation named these creatures *robots* - hence, the play was called *Rossum's Universal Robots*.

However, the term robot has only given a name to a fantasy that inspired humans much earlier than that; even ancient history provides some examples of man envisioning and building simple, yet useful automata like music machines. In the Late Middle Ages, Leonardo da Vinci designed a mechanized armor that – based on strings and pulleys – was able to execute basic movements. Of course, as knowledge of the required prerequisites was so limited, none of the more advanced ideas could actually be realized. This vast divide between human imagination and capabilities introduced interesting anecdotes into robotics history: one of the most well-known is the automaton chess player named the *Mechanical Turk*. Built in the 18th century, the Mechanical Turk

was a fake machine that seemed to be playing chess. In fact, it was a mechanical illusion designed to hide a human chess player of short stature, operating the device from the inside. After becoming well-known, the machine was presented in large parts of Europe and America for more than 50 years, often winning chess matches against capable chess players and causing both curiosity and fear in members of the audience (Standage, 2002).

After the second world war, the transistor was invented, heralding the age of miniaturization. Electronics, mechanics, engineering, and computer science advanced at unprecedented speed, spawning electronic computers and fusing into what now is often referred to as mechatronics. Within the next ten years, robots started gaining a foothold in production processes, founding the industrial robotics sector. In the following thirty years, these machines have developed superior performance in comparison to human labor in many regards: often, they are either faster, more precise, more durable or even cheaper in the long term. Today, production robots often outdo humans in all of these at the same time.

On the other hand, while robots have become very common in highly structured, industrial environments, their adoption in other fields of human labor has been excruciatingly slow. In unstructured environments, today's robots still are incapable of performing even the most simple tasks in acceptable time – how else can it be explained that as yet, there are no robots helping us in the supermarket or at the gas station? As Danish author Tor Nørretranders puts it,

> "It is not that difficult to build computers capable of playing chess or doing sums. Computers find it easy to do what we learned at school. But computers have a very hard time learning what children learn *before* they start school: to recognize a cup that is upside down, for example; navigating a backyard, recognizing a face; *seeing*." (Nørretranders, 1999, p. 179)

For the age of robots to truly begin, humans will have to find a way to teach them what children learn before they start school: be mobile, sense, feel, remember and recognize their environment, navigate, manipulate objects, and act in social contexts. The farther away from industrial manufacturing and other similar environments robots are to be of help, the more important these skills become.

The history of geodesy is a very different one. The word "geodesy", in its original Greek form, is a concatenation of the terms for "earth" and "division" and is defined as the science of measuring and representing earth. Geodesy was motivated by the need to partition land and define its bounds, creating borders between real estate and states. Doing so, geodesy has also affected fields as fundamental as time-keeping and as important as politics and economics. Since these impacts of geodesy fall into the domain of history, they are deemed out of this section's scope. More interesting to us, however, is how geodetics has been subject to influences from other domains: the following paragraphs shall focus on how science and technology have changed the

capabilities of measuring earth.

About 500 years B.C., Pythagoras postulated that earth was not flat, but had in fact a sphere-like shape. It took about 150 years until Aristoteles proved this assumption based on observations with the naked eye. Ever since then, geodesy was limited by the precision of technical instruments available at the given time. Nevertheless, measurements have been conducted with astonishing precision: in the year 1023, an Islamic scholar named Abu Reyhan Biruni first measured the distance between a known point and a mountain, then, at the same known point, the angle between the astronomical horizon and the mountain's top. He then used trigonometry to calculate the mountain's height, climbed it and measured the angle between true and astronomical horizon. Doing so, he determined earth's radius to be 6339.6 km (Lumpkin, 1997), which is 99.4% of earth's true radius at the equator and 99.7% of its radius at the poles[1].

Some of the most important advances in geodesy were introduced at the beginning of the 19th century by Carl Friedrich Gauß. His exceptional grasp of mathematics and physics allowed him to author significant contributions in numerous fields, but his invention of the least-squares method and achievements in non-euclidean geometry had an enormous impact on geodesy. As a man of practical experience, he was ordered to survey the Kingdom of Hannover in 1818 and applied the least-squares method to create maps of revolutionary accuracy using triangulation. Also interested in the instruments required for surveying, he invented the heliotrope to mirror sunlight from surveying points towards a theodolite many kilometers away. This worked so well that the earth's curvature remained the only constraint to baseline length. Furthermore, Gauß contributed works relating to projection, an important requirement for representing earth's non-flat surface on flat maps.

For the next 150 years, geodesy continued relying largely on triangulation. The required data (azimuth-only at first, later also elevation) was gathered using theodolites and allowed for precision in the sub-centimeter range over short baselines. Unfortunately, these procedures necessitated free lines-of-sight between points, meaning that oftentimes, a mesh of triangles had to be created in order to support the desired final measurement. Creating larger maps was possible, but hindered by the fact that increasing imprecision in the mesh due to accumulating errors in local measurements was hard to rectify.

Until the end of the last century, robotics and geodesy had rather few things in common.

In the 1970s, the cold war motivated the United States government to develop a more precise, global successor to previous navigation systems. What became known as NAVSTAR-GPS[2] is, in principle, nothing more than a network of moving reference points. Instead of triangulation, trilateration is used to determine positions, and even

[1] compared to the WGS84 ellipsoid (explained in chapter 4)
[2] NAVigation System using Timing And Ranging, Global Positioning System

though direct line-of-sight between the point to be localized and four reference points (satellites) is still required, this constraint is rarely an issue in outdoor environments, given that the number of satellites and choice of their orbits was also optimized for maximum visibility in the latitudes densely populated by humans (Eissfeller et al., 2007).

After becoming operational in the beginning of the 1980s, GPS remained a closed, military technology for many years. Things changed with the end of the cold war: the United States grew less concerned about sharing NAVSTAR's benefits with other nations – aside from a few constraints (Adrados et al., 2002), the first GNSS[1] could now be used in the civilian sector. Ever since, satellite technology generally and GNSS particularly have helped advance the state of the art in geodesy and mobile robotics tremendously (Kumar and Moore, 2002).

In the early 1990s, carrier-phase-GNSS enabled determining positions around earth with centimeter-accuracy. The capability to provide this precision over kilometer-scale, non-line-of-sight baselines was very attractive to the surveying community, which subsequently became the largest market for this technology. Figure 1.1(b) depicts one of the first Real-Time-Kinematic (Carrier-phase and RTK-GNSS are explained in section 3.1.3 A) GNSS antenna and receiver carried on a yak on an expedition to Mount Everest in one of the first documented applications of RTK-GNSS in 1992. Recognizing the value of ubiquitous positioning services, governments around the world are now implementing their own GNSSs: after GPS and GLONASS constellations have provided worldwide public service for many years, the Chinese BeiDou constellation currently offers regional service with medium accuracy and the European Galileo satellite system is currently performing its in-orbit validation. A more detailed look at each constellation's status shows that there are currently 32 GPS, 24 GLONASS, 15 BeiDou and 4 Galileo satellites servicing earth, adding up to a total of 75.

Figure 1.1(a) shows recent testing of a Galileo satellite, which is part of the European Union's efforts to establish an independent GNSS. Although these developments are in large parts motivated politically, they do help improve reliability and precision of global satellite-based navigation. At about $10k per receiver, RTK-enabled GNSS equipment is still expensive, severely hindering its use beyond professional applications. As presented in figures 1.1(b) to 1.1(d), it is currently seeing widespread use in surveying, construction and agriculture. While GPS was first developed for precise guidance of intercontinental ballistic missiles, it is now integrated in most newly developed military platforms: three examples are given in figures 1.1(e) to 1.1(g), showing vehicles in air, water, and on land.

Robotics research was initially constrained to structured indoor environments (where GNSS signal reception is weak at best) and quickly revealed that autonomous environment sensing and learning of a world model are most fundamental challenges.

[1] Global Navigation Satellite System

Especially in service- and field-robotics, the availability of an environmental map is essential for many tasks such as localization, path planning, navigation, and manipulation. Consequently, vast amounts of research have been directed at this field. A very good example of progress is the ability of simultaneous localization and mapping (SLAM): little over a decade ago, first-generation SLAM approaches (e.g. as presented in Thrun et al. (1998)) have been limited to robots moving on planar ground in highly structured indoor environments. During the last fifteen years, the algorithms advanced as researchers' understanding of the underlying scientific problems made substantial progress. At the same time, the state of the art in mechanics, chemistry and electronics has also advanced: better locomotion has allowed robots to become far more maneuverable and operate in outdoor environments, better energy sources have allowed for longer operation and better computers allowed for faster speed.

When mobile research robots started conquering the outdoors, GNSS became a natural choice for localization: with limited demands for accuracy, a GNSS receiver for less than $50 can enable a robot to solve its position at any time of day under all weather conditions. In 2005, DARPA hosted the Grand Challenge, promising a reward of $2M for the team whose vehicle would first negotiate a 212km long off-road track. The winning vehicle, shown in figure 1.1(h), finished the course after about 7 hours (Thrun et al., 2006). Like the platform to be presented in the following chapters, it used LIDAR[1] sensors and a global positioning/inertial navigation system to build a 3D map of the environment and make decisions on how to move in order to best reach the given goal. Just three years ago, algorithms were presented capable of localizing robots with six degrees of freedom, while mapping three-dimensional space in real-time (Newcombe et al., 2011; Izadi et al., 2011).

Unmanned Aerial Vehicles (UAVs) have existed since the mid 1980s, when DARPA financed their development for military applications. With flight-times of up to 48 hours, these aircraft could perform reconnaissance missions significantly longer than their piloted counterparts. About 20 years later, the United States military started equipping some UAVs with air-to-ground weapons, which, combined with aforementioned flight-times, established a state of constant surveillance over large areas, with the capability to conduct strikes against perceived enemies at any time. The inevitable killing of innocent bystanders, the killing of civilians due to incorrect intelligence information and the constant threat of unforseeable attacks has induced stress and fear in the population living in these areas. Approximately one decade of this practice has given the terms UAV and drone a negative connotation that fosters reservations towards civil UAV applications to this day.

Starting in about 2010, UAVs in sub-meter scale were developed by hobbyists and small companies for recreational use. Growing more powerful, dependable and autonomous over the last years, these vehicles are starting to see more commercial applications: visual inspection of high-rise architecture, photogrammetry, precision agriculture and

[1] Light detection and ranging

filmmaking are just a few of them.

Although the variety of applications for civil UAVs has grown strongly in the last few years, mapping an unstructured outdoor environment by simply defining a region of interest in 3D space would constitute a considerable improvement over current UAVs capabilities. The aim of this thesis is to create such an experimental platform. To do this, the system must bring together latest developments in geodesy, robotics, computer science and other fields, to

1. safely carry the required sensors in the presence of wind and obstacles.

2. perform self-localization for all six degrees of freedom.

3. fuse information about its own position and orientation with laser ranging data acquired during flight, in order to create a georeferenced point cloud of its environment.

4. autonomously generate waypoints from the incrementally growing point cloud, optimized to efficiently advance the mapping process and to create a watertight (i.e., gap-free) point cloud of the environment,

5. approach these waypoints while constantly performing collision avoidance.

Being an *experimental* platform, the system is *not* expected to operate in all weather conditions, produce measurements of survey-grade accuracy or support non-essential features such as automatic take-off and landing. Furthermore, mapping GNSS-denied areas like indoor environments or outdoor surroundings with highly obstructed skies will not be required either.

Research and – if successful – further development of such a system puts new capabilities at the operator's disposal, which always bears a social impact. These capabilities can be used for civil applications, where they might replace some parts of human labor, e.g. in surveying or 3D modeling. The motivation to automate human labor has always been a driving force behind robotics research. Yet, when applied in practice, jobs are oftentimes replaced. Similar systems with longer flight-times as well as higher scanning-range, speed and accuracy will undoubtedly provide a feature-set that is attractive to the military, as it can be operated day or night, extending airborne reconnaissance data into the third dimension. The public needs to be made aware of these developments, so that they can be accompanied by a public debate, allowing both concerns and opportunities to be openly discussed.

Videos showing the successful application of the presented approach for exploration in real-time are available via `http://tams.informatik.uni-hamburg.de/videos/`.

(a)

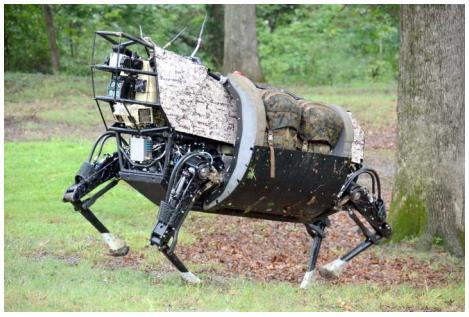
(b)

(c)

(d)

(e)

(f)



(g)

(h)

Figure 1.1: (a): A Galileo space vehicle in testing, showing its L-Band antenna array. (b): An RTK-GNSS receiver on a yak during a 1992 expedition on Mount Everest. (c),(d): GNSS-based machine control in different industries. (e),(f),(g): military applications for GNSS. (h): Stanley, with GNSS pinwheel antennas visible in the back.
Images ©: (a) 2013 ESA Anneke Le Floc'h, (b) unknown, (c) 2009 Trimble Ltd., (d) 2014 Marius Frei, Gut Lunzberg, (e) (g) (h) public domain, (f) 2014 ASV Ltd.

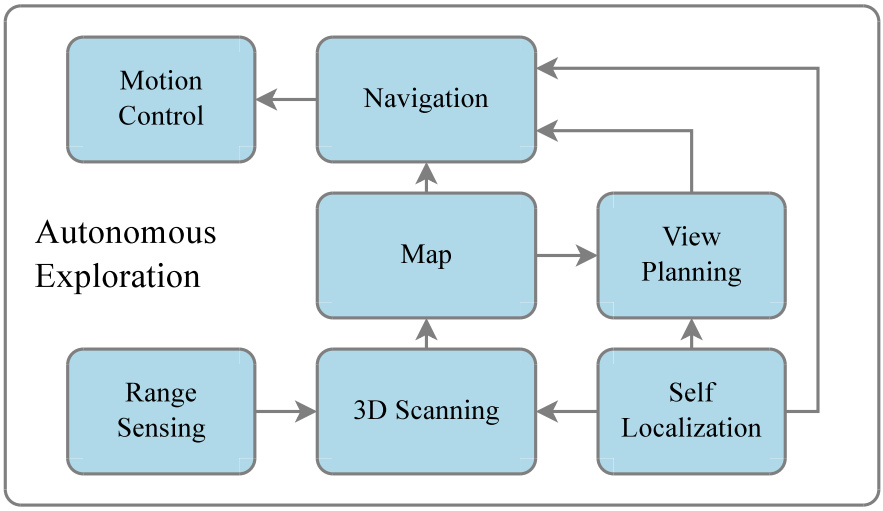


Figure 1.2: A diagram showing the architecture of and interaction between key components that are necessary for autonomous robotic exploration.

1.2 Research questions and contributions

After establishing this thesis' goals, its research questions span multiple topics, and can be formulated as follows:

- Engineering & Mechatronics: How to design a lightweight and maneuverable unmanned aerial vehicle, acting as a sensor platform that is capable of streaming a point cloud of the scanned environment to a base station in real-time?
- Algorithms & Datastructures: How to design and implement algorithms that navigate this vehicle autonomously through unstructured outdoor environments for efficient mapping?
- Real-Time Computing: How to detect and react to potential collisions that appear during mapping or originate in dynamic obstacles?
- Control Theory and Practice: How to design and implement approaches to motion control, how to test them without failures incurring expenses and downtime?

The following contributions are presented in the following chapters:

- A custom-built UAV that integrates four main sensors on a fault-tolerant platform (equipped with redundant drives), operating in a six-dimensional configuration space. This system serves as the experimental platform and is discussed in section 3.1.
- A software architecture – described in section 3.2 – that allows flight testing in simulation, real-time shader-based visualization and replay of real and simulated flights as well as GNSS mission planning.

- Real-time capable algorithms to find configurations in that space, maximizing information gain in the process. Chapter 4.2 introduces an initial proof-of-concept implementation on the CPU, then describes their implementation on a graphics card using NVIDIA's Compute Unified Device Architecture (CUDA).
- An approach that allows for safe airborne navigation in dynamic surroundings. This is discussed in section 4.3.
- A high-level flight controller that steers the UAV towards a designed waypoint in the presence of wind, and is also optimized for information gain. This is documented in chapter 4.4.
- A critical analysis of the system's performance in real-world conditions, given in chapter 5.
- A sketch of future research in chapter 6.

1.3 Structure of the Thesis

The remaining parts of this thesis are organized as follows:

- Chapter 2 presents a detailed overview of the foundations, problems, and work in this and related fields. Since this thesis discusses many subfields of robotics, the chapter is separated into corresponding sections.
- Chapter 3 revolves around the practical aspects of the experimental platform. It introduces the general architecture of the hardware and software setup. Explanations of each part's basic operational principles go along with its specifications. Experiences that were gathered with these components are also noted, as they often called for redesign of the platform. The second part elucidates on the software architecture, first analyzing the arguments for and against a custom design, then explaining the separation into modules and their purposes.
- Chapter 4 focuses on the theory and algorithms used. It starts by presenting the foundations required to generate georeferenced maps, then continues to describe the idea behind the approach for generating waypoints providing high information gain. The required computational hardware is described first, followed by the data structures and algorithms that were used in order to reach real-time performance. The next section describes how these waypoints are sequenced into collision-free paths through the environment, and the chapter concludes by demonstrating the motion control systems able to guide the UAV.
- Chapter 5 presents the results achieved in the previous chapters. Basic motion control results are shown and related to environmental envelopes. Waypoints and paths generated for point clouds captured during real flights are presented and the runtime behavior of the supporting algorithms is analyzed.

- Chapter 6 concludes the thesis by summarizing the results and discussing future work.

1.4 Prior Publications and Research Collaboration

During the last four years, the work done for this thesis has led to several publications, which were accepted at conferences and journals. The following list includes only those publications directly related to this thesis, which form the basis of the respective sections of chapters 2, 3, 4, and 5.

- Benjamin Adler, Junhao Xiao, *Towards Autonomous Airborne Mapping of Urban Environments*, 2012 IEEE International Conference on Multisensor Fusion and Information Integration (MFI 2012), Hamburg, Germany, September, 2012, pp. 77–82.

  This first paper introduces the first configuration of the UAV and presents the idea of using particle simulation to test a point cloud for watertightness. The CPU-based proof-of-concept is described and its effectiveness is validated using simulation.

- Benjamin Adler, Junhao Xiao, Jianwei Zhang, *Finding Next Best Views for Autonomous UAV Mapping through GPU-Accelerated Particle Simulation*, 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 2013, pp. 1056–1061.

  This publication extends the previous paper by presenting an implementation of the next-best-view algorithm on a GPU. Execution on this hardware architecture now allowed for real-world, real-time experiments, which are also presented.

- Benjamin Adler, Junhao Xiao, Jianwei Zhang, *Autonomous Exploration of Urban Environments using Unmanned Aerial Vehicles*, Journal of Field Robotics, Volume 31, Issue 6, November/December 2014, pp. 912–939.

  This paper discusses the integration of localization, mapping, motion-planning and -control, and shows the effectiveness of the resulting platform.

In these publications, the first author was responsible for problem definition, solution finding, programming and visualization. He took part in and steered discussion and experiments.

This thesis itself represents a contribution by documenting the complete and refined concept as well as the development of the experimental platform in a clear and concise way.

Furthermore, the simulator uses 3D models and textures published by the Ogre project and other authors under open-source or creative commons licenses.

Chapter 2

State of the Art in Autonomous Exploration

Even though the term *autonomous exploration* is quite descriptive, it is not well-defined. Progress in the last decades has shown that its definition seems to grow with modern robots' capabilities: depending on interpretation, autonomous exploration was implemented a long time ago, when very basic robots were built in educational contexts to locate and approach light sources. Today, robots must exhibit far more complex behavior in order for humans to attest them this skill.

Abstracting from the ever-changing state of the art, exploration extends the activity of measuring the world in passive manner: data produced by measurements is generally incomplete and contains uncertainty, so for the autonomous explorer to reduce this uncertainty, it is required to "seek out those parts of the world which maximize the fidelity of its internal representations, and keep searching until those representations are acceptable" (Whaite and Ferrie, 1997).

There are only very few robots that really explore arbitrary terrain, and the primary reason for this is rooted in the fact that autonomous exploration is a very complex skill to master. Indeed, it is a combination of a surprising number of other capabilities that robots have only achieved in recent years. Most of these capabilities emerged from advances in related fields such as algorithms and software, some from progress in mechanics and miniaturization or – less obviously – in chemistry, which brought about batteries that enable reasonable endurance of e.g. multirotor flying robots.

The remaining part of this chapter presents an overview of the most interesting problems and technologies involved in autonomous airborne exploration.

2.1 Unmanned Aerial Vehicles

As hinted in the previous chapter, the term UAV applies to a wide range of aircraft, which have followed the typical trend of miniaturization in the last decades. For UAVs, an obvious dichotomy is the classification into fixed-wing and helicopter drones. Vehicles in the former class traditionally allow for larger payloads and flight-times, but also put higher demands on the infrastructure (e.g. runways, long-range communications etc.). Being far less maneuverable, fixed-wing aircraft also require more complicated path-planning. One of the consequences is that restricting flight-paths to remain within line-of-sight cannot be guaranteed, especially so in urban environments. Lastly, fixed-wing aircraft cannot hover – indeed, the minimum velocity required to avoid a stall can become problematic, as it imposes an upper bound to point cloud density when scanning with LIDAR sensors of limited sample rate.

For this reason, multirotor UAVs were chosen as the integral part of this thesis' experimental platform. Although their comparably limited payload and weight are major limitations, these can be accounted for by selecting lightweight sensing hardware. Furthermore, flying UAVs with a total take-off weight above 5kg requires tedious and expensive certification procedures for both the vehicle and the operator. On the upside, multirotor drones are extremely maneuverable, can fly at very low speeds and even hover, which allows path planning and motion control to accomodate special requirements of sensors carried on-board the platform. When equipped with more than four rotors, the resulting redundancy is another advantage. When using an octocopter (with 8 drives), an in-flight failure of a single motor or propeller can be tolerated. Most importantly, a multirotor UAV can be flown in very constrained environments: when implementing and testing a flight controller, the possibility to fly the aircraft at very low speeds and altitude (or even attached to a long fishing rod) is extremely helpful. This is expecially true when the budget does not allow for a crash.

A survey of the market in the end of 2010 showed that when allowing maximum flight-times to decrease down to the ten-minute range, multirotor UAVs with a payload capacity of 2kg do exist. Some of them even allow for input from an external high-level flight-controller, making them a perfect fit for this thesis.

2.2 Range Sensing

Range sensing is performed using devices that measure distance between themselves and an object's surface. A wide range of sensors are available for range sensing today, and – depending on their targeted application – differ greatly in how they operate. Range sensors represent one of the most fundamental components of a mapping robot and can be classified along numerous criteria:

- Physical principle of operation. This factor bears the largest influence on the

nature of the sensor, and all following criteria are most strongly influenced by this initial design choice.

- Range. Sensors usually impose restrictions on both minimum and maximum distances they are able to sense, also called the near and far planes. These ranges depend largely on the principle of operation, sensor-size, and power consumption. Typical ranges extend from few centimeters to the small kilometer-range.

- Field of View (FoV). A sensor's field of view is one of its most important design aspects and often the primary criterion for applicability to a given task. FoVs of range sensors vary from infinitely small (a single ray) almost to the full sphere. Oftentimes the sensors are conceptually one-dimensional, but are extended to more dimensions using mechanical setups like mirrors and/or angular actuators like pan/tilt units. These sensors are also known as *actuated* (laser) range finders ("aLRF").

- Angular resolution and accuracy. For sensors that sample multiple ranges, resolution is the angle between neighboring rays. Angular resolution can be different along azimuth and elevation. In unison with the FoV, this property also describes the amount of samples a sensor produces in one scan. The latter property describes the maximum angular error, which translates to a positional error scaling linearly with the recorded distance of the reading. Angular errors can originate in mechanical misalignment as well as the width of the beam used to sample the surface, so usually both are sought to be minimized.

- Range resolution and accuracy. Whereas high accuracy is required for surveying and related applications, low accuracy can be perfectly sufficient for other tasks like collision avoidance.

- Sample rate. This property describes the number of ranges sampled by the sensor during a given time span.

- Waveform digitization. When targets reflect the sensing beam back to the scanner, the resulting energy on the detector can be represented as an energy-over-time plot (the *waveform*). Simple signal processing in lower grade scanners registers a return as soon as this energy exceeds a threshold. When the full waveform is saved, it can be analyzed to retrieve additional information, such as amplitude and pulse width.

- Multiple-target capability. A higher beam divergence causes the beam's footprint to increase faster with distance. When the same beam hits multiple targets at different distances, some scanners are capable of detecting this condition, and correctly return multiple distances. This requires full-waveform-analysis and a time-of-flight based scanner (since phase-based scanning does not allow for multi-target detection).

- Multiple-Time-Around (MTA). For LIDAR scanners, having to wait for the laser

pulse to return can limit the pulse repetition rate. Some scanners are capable of firing the second laser pulse before the reflection of the first pulse has arrived. This introduces an ambiguity when trying to assign incoming reflections back to outgoing laser pulses. To resolve this ambiguity, further information is required.

- Size, weight and cost, which usually correlate with each other. Observation of the market reveals that all of them decrease over time.

Up until a decade ago, the market for range sensors has been very sparse. On the low-end, infrared and ultrasonic distance sensors were available, but delivered one-dimensional results with inadequate precision: the former emit a single beam of infrared light and measure the angle of the incoming reflections using a slightly offset infrared-sensor-array behind a lens (emitter and sensor shown in figure 2.1(a)). Using this angle and the distance between IR emitter and lens, the range can be determined through triangulation. Ultrasonic sensors (see figure 2.1(b)) measure distance by observing the time-of-flight of sound waves, but suffer from even wider beam widths and other problems such as crosstalk, necessitating special techniques such as temporal synchronization between sensors or software multipath mitigation (Borenstein and Koren, 1995).

Infrared, ultrasonic and other principles that are clearly unsuitable for outdoor sensing will not be discussed any further. Instead, the remaining parts of this chapter will focus on time-of-flight (ToF) and structured-light-based scanners.

In 1999, SICK AG introduced the LMS 200 (see figure 2.1(e)), a laser range finder that improved upon aforementioned sensors in terms of maximum sensing range and sampling rate by more than an order of magnitude (also see figure 2.2 for a comparison of range sensors). Using time-of-flight measurements, it scanned a 180° FoV at a rate of up to 75Hz with 0.5° angular resolution and registered reflections from objects up to 80m away. Conceptually, the LMS 200 is a one-dimensional range measurement device. By adding a mirror angled by 45° relative to the beam, it is deflected by 90°. Because this mirror rotates on the axis of the original beam, the ray effectively samples a plane, producing two-dimensional range data. SICK originally designed the range sensor for use in industry and safety (where its power consumption of 30W and weight of 4.5kg caused no concern), but it quickly became the most widely-used range scanner in robotics, fostering the development of countless localization and mapping approaches on ground-moving robots (see e.g. Pfister et al. (2003)).

The Hokuyo URG 04lx was introduced to market in 2005. It measured range by determining the phase difference between the emitted and reflected light. While the scanner was very small, lightweight and could be powered through USB, the range-readings were rather inaccurate and limited to less than 4m. Its successor, the UTM-30lx pictured in figure 3.10, was released in summer of 2009 and can be described as a miniature version of the LMS 200, also employing ToF range-measurement techniques. Offering similar precision and comparable range for targets of medium reflectance, the

(a) (b)

(c) (d)

(e) (f)

Figure 2.1: (a) and (b): Infrared and ultrasonic one-dimensional range sensors with ranges of less than 2m. (c): The PMDVision CamCube Time of Flight camera. (d): The Microsoft Kinect 1, sensing based on structured light. (e): The SICK LMS 200, a 2D laser range finder with a 180° FoV. (f): Velodyne HDL 32E 3D laser scanner with 360° horizontal and 40° vertical FoV. Images ©: (a) SHARP Corporation, (b) Parallax, Inc., (c) PMDTechnologies GmbH, (d) public domain, (e) SICK AG, (f) Velodyne Lidar

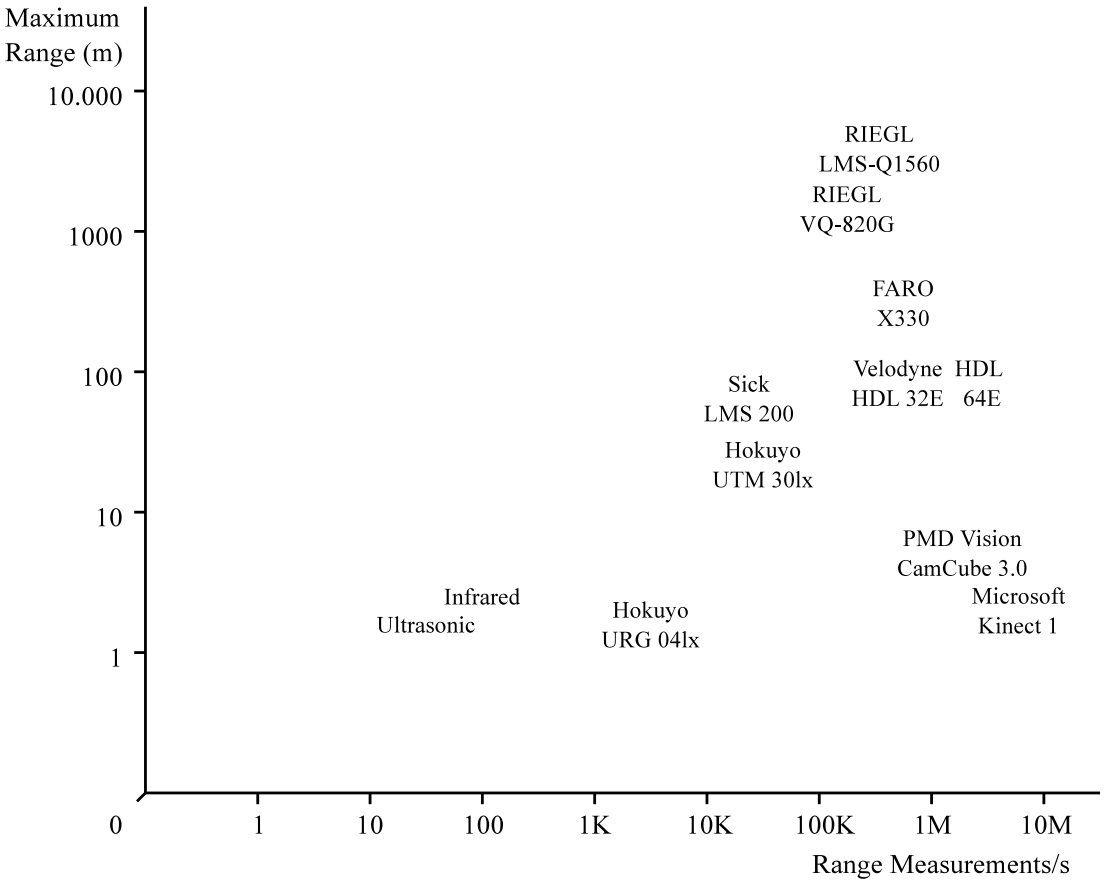


Figure 2.2: The range sensor market situation as of 2014, relating maximum range and sampling speed.

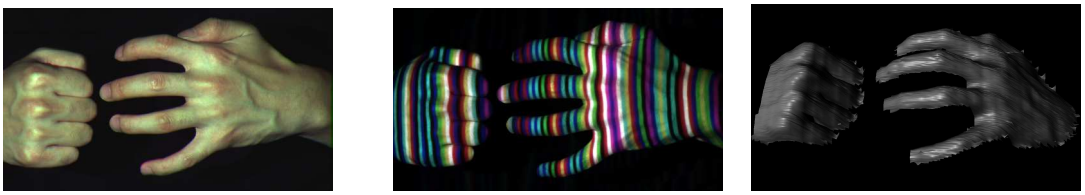
UTM-30lx slowly replaced the SICK and allowed smaller, more mobile robots to map environments. Weighing about 5% of the LMS 200, the UTM-30lx also was the first LIDAR payload suitable for outdoor mapping on small UAVs.

In the past, when weight was of no concern, many research-projects in robotics employed multiple 2D scanners in different (often perpendicular) orientations to gather 3D information (Frueh and Zakhor, 2001; Thrun et al., 2000; Zhao and Shibasaki, 2003). More recently, research groups have used actuated laser range finders to extend the FoV of these 2D scanners into the third dimension. To do so, 2D LRFs are mounted onto a fixture that can be panned or tilted, in effect sweeping the scanplane. Some publications are based on custom-made actuators (Nüchter et al., 2006; Bosse and Zlot, 2009), others use much more expensive, commercially available scanners that are oriented towards surveying and thus, provide much higher accuracy at a much higher price (Georgiev and Allen, 2004; Elseberg et al., 2012).

As much as this setup extends the FoV, it also extends the time necessary to scan it, often into the range of minutes. Furthermore, precise timing and synchronization of scanning and tilting motions are required in order to produce correct data.

Research conducted in more recent years has advanced miniaturization up to the point where integrating multiple laser emitter/detector pairs into compact enclosures has become feasible. Instead of simply increasing angular velocity or scan-rate, this achievement was utilized to create truly three-dimensional LIDAR sensors. The possibly most well-known instance of this class is the Velodyne HDL 64E, first seen at the 2004 DARPA grand challenge, introduced to market in 2007 and prominently featured on Google's driverless cars. In contrast to the aforementioned 2D scanners, this scanner has a horizontal FoV of 360° and hosts 64 emitter/detector pairs that are oriented in the same azimuth, but have a vertical angular spacing of about 0.4° for a 27.8° vertical field of view. In 2010, Velodyne released the HDL 64E's successor, the HDL 32E (pictured in figure 2.1(f)): offering 32 emitter/detector pairs at a cost of about USD 30k, the scanner cuts both the number of lasers and the price of its predecessor in half. Spread over a 40° vertical field of view, the angular stride is increased to about 1.3°. Although the HDL 32E weighs approximately 1.2kg, it has seen its first flight on an electrically powered multirotor UAV in 2013 (Velodyne Lidar, 2013).

Another technique for range sensing is the so-called *structured-light*-approach. Sensors based on this concept consist of calibrated projector/camera combinations that project a predefined light pattern into an environment. An example of the setup is presented in figure 2.3: using the camera, the projection of this pattern on the environment's geometry can be recorded. The scale of the scene can be reconstructed quickly, since the projection parameters of both light source and camera are known. When it is projected on a flat surface with its normal parallel to the projection axis, the pattern will appear similar to the original pattern. On non-flat or non-parallel surfaces, the pattern's projection will become distorted, allowing image-processing algorithms to reconstruct the surface's shape (Zhang et al., 2002). This way, 3D range sensors can be built by



(a) (b) (c)

Figure 2.3: 3D reconstruction using structured light: (a): photo of a simple scene to be reconstructed (b): pattern emitted into the scene is distorted when projected onto a non-flat shape. (c) shows the reconstructed shape of the object. Reprinted from Zhang et al. (2002). Images © 2002 IEEE.

employing comparably cheap and commonly available components. A good overview of different structured-light techniques that includes results of some implementations is given in Salvi et al. (2010).

At the end of 2010, Microsoft introduced the Kinect sensor (see figure 2.1(d)) as an accessory for its XBox game console. Within this device, an infrared laser projects a dot-pattern into space. The distance to points in the scene can be determined when an image of them is recorded by an infrared-sensitive camera. Because laser's light lies outside of the spectrum visible to humans, an additional RGB-camera is able to produce a video stream of the scene. Combining these sensors data results in a colored 3D point cloud, explaining why the Kinect is called an RGB-Depth (or RGBD) camera. Neither Microsoft nor PrimeSense (the manufacturer of the Kinect's signal processing components) have disclosed details of the sensor's algorithms, and the exact functional principle has withstood many attempts of reverse-engineering to this date. However, seeing the impressively precise depth data (VGA resolution produced at a rate of 30Hz, near plane at 4m, far plane at 7m) and a price point at a fraction of other sensors' cost, the robotics community quickly reverse-engineered the protocol used to stream data to the host-computer, and adopted the Kinect for indoor range sensing. Since its release, numerous scientific papers focusing on Kinect's abilities and possible applications have been published – today, sensors similar to the Kinect weigh only slightly more than 200g. At first sight, these characteristics seem to be a great match for airborne surface reconstruction.

When structured-light sensors project a pattern to a surface (creating the *signal*), the light produced has to be distributed over the surface's area. The irradiance of light on a surface is measured in watts per square meter, and it follows that for a projector with constant power, this value is inversely proportional to the area covered by the pattern. Depending on the algorithms used, the image produced by the camera will have to represent the pattern with a sufficient amount of contrast. That is, the signal to noise ratio in the image must not fall below levels that are acceptable to the image processing algorithms employed in later stages of the pipeline. For a structured-light camera, solar irradiation is the primary source of noise in the outdoors, quickly diminishing the signal-

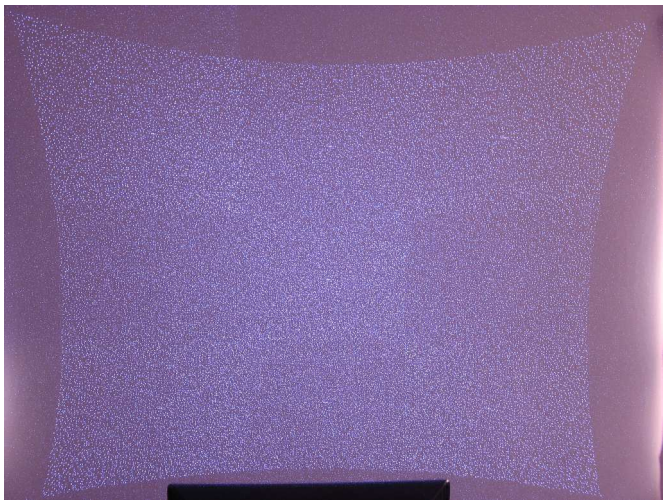


Figure 2.4: The Kinect's infrared light speckle pattern allows three-dimensional reconstruction (image in public domain, reprinted from Anonymous (2010))

to-noise ratio and causing the approach to fail in the open. There are some measures to counter the detrimental effects of the sun on structured light sensors: the Kinect features an 830nm bandpass filter on the IR camera, blocking light from other devices such as remote controls as well as large parts of sunlight's IR energy. However, the sensor still tends to fail in places lit by sunlight, attesting to the power of sunlight in this region of the spectrum. Improving the signal-to-noise ratio by increasing the projector's power is an obvious strategy, but constrained by concerns regarding eye-safety. As witnessed by PrimeSense's patenting and patent licensing activity (and described in detail by Kramer et al. (2012)), Kinect's designers invested much effort to create a strong, yet safe IR pattern using a laser diode: most optical diffusors used to create speckle patterns would cause a bright dot to remain in the pattern's center. In order to keep light emissions from the 70 mW laser diode eye-safe, this imperfection was rectified by distributing this energy over 9 dots of lower brightness (see figure 2.4). In conclusion, designers of structured light sensors are forced to engage in a compromise between range, power consumption, eye safety, and accuracy.

So far, while some structured-light sensors known to the author are capable of mapping in open-sky and daylight conditions, none provide a reasonable range (see Mertz et al. (2012), figure 2.5). ToF-based LIDAR sensors currently offer properties more suitable to airborne outdoor mapping. For this reason, two lightweight Hokuyo UTM-30lx range finders are used on the experimental platform.

2.3 3D Scanning, Registration and Self-Localization

A range sensor's range and field of view limit the surface of a scene that can be scanned using a single pose. In both indoor and outdoor environments, even small scenes often

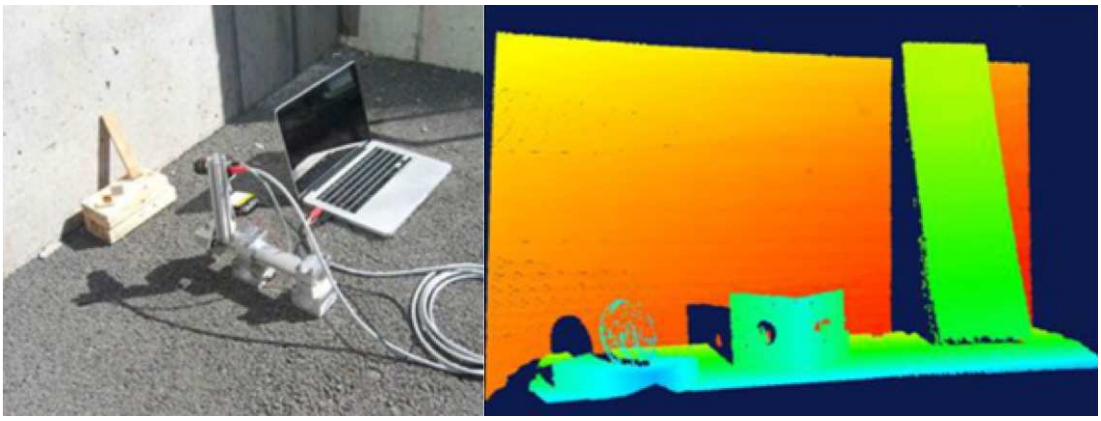


Figure 2.5: Insufficient range of structured light sensors in the outdoors, from Mertz et al. (2012). Images © 2012 IEEE.

cannot be reconstructed from a single scan, because there is no viewpoint from which all surfaces can be seen. Consequently, a scanner must be placed in different view-configurations in order to completely map a given geometry, which will result in multiple scans. Oftentimes, the first scan defines the origin of the global coordinate system being used. As soon as the next scan (known as *source scan*, *current scan* or *data set*) has been acquired, it can be *registered*, meaning that a transformation (translation and rotation) from its local coordinate system into the global coordinate system must be found, such that it is correctly aligned with the previous scan(s) (known as *target scan*, *reference scan* or *map*). It is important to note that with the range scanner rigidly mounted to a robot, the problem of scan registration becomes equivalent to the classic robotics problem of self-localization (Olson, 2000). In other words: if two consecutive scans can be registered, the transformation of the robot between these two scans is also solved. If, on the other hand, the robot's transformation between two scans can be determined by other means (e.g. a navigation system), scan registration is reduced to applying this transformation to one of the scans. For this reason, 3D scanning, scan registration and self-localization are closely interdependent, and their state of the art is introduced in the same chapter.

2.3.1 Static scanning

The idea of static scanning is to fix the ranging device in a stable position while it scans its complete FoV (see figure 2.6 for an example application). Depending on the actual device being used, this process might take between anywhere from fractions of a second up to a few minutes. After completion, the scanner is moved and the procedure repeated until all surfaces of interest have been scanned.

When registration is computed by trying to align the new scan to any other in the pool of correctly registered scans, the search space increases as scanning progresses. Registration search can be constrained by reducing the pool of target scans to only the

(a) (b)

Figure 2.6: (a): Static scanning using a tripod-mounted aLRF and reference spheres for scan alignment in post-processing. (b): A visualized scan with visible reference spheres remaining. Images ©: (a) 2013 International Partner Büro S.R.L., (b) 2014 DeWalt Corp.

previous scan, resulting in pairwise registration.

Every registration includes small errors in alignment, so a growing chain of registrations will accumulate a growing error. Whenever a scan overlaps not only with its direct, but also with a non-direct predecessor, this presents a chance to determine and then correct registration errors that have built up during the course of mapping. This approach is called loop-closure and is very efficient in improving the global consistency of produced maps. Because of this, a compromise must be made between pairwise-only registration (requiring lower computational complexity due to a reduced search space) and unrestricted registration, which allows for loop closure.

For algorithms to correctly register two consecutive scans, a sufficient amount of overlapping surfaces must be included in both. Algorithms can be classified as either local or global, depending on the search method that backs the registration.

Local algorithms require not only the scans themselves, but also an initial estimate of the transformation between them. In robotics, this information is often derived from odometry sensors such as wheel encoders or double-integrated readings from an inertial measurement unit (IMU). Probably the most well-known local registration algorithm is ICP (Iterative Closest Point, Chen and Medioni (1991); Besl and McKay (1992)), which works as follows: first, for evey point in one scan, the closest point in the other scan is selected as the corresponding point. ICP then strives to compute the optimum alignment between the two scans by transforming the source scan so that the sum of the square of all distances between corresponding points is minimized. This process is repeated until the sum of squared distances converges, often leading to correctly aligned scans. The algorithm is conceptually simple, but as described, requires that for every point in the source scan, the nearest neighbor in the target scan is found – a process that is time consuming and must be repeated in every iteration. In order to alleviate this shortcoming, common approaches to neighbor search acceleration like kd-

trees have been implemented (Nüchter et al., 2007a). Other publications have further improved other characteristics of the algorithm; for example, ICP has traditionally been sensitive to outliers, as they introduce large offsets into the sum of squares, increasing overall misalignment when ICP tries to accommodate them. Segal et al. (2009) present an attempt in order to decrease ICPs susceptibility to outliers specifically as well as measurement noise in general.

However, even after a decade of improvements, ICP's real-world application is limited to precise, locally consistent scans with good initial alignment transformations. This is because the point correspondence criterion (Euclidean distance), while being simple to implement, does not guarantee correctness. Therefore, especially scans of complex geometry or insufficient pre-alignment are prone to converge to local minima.

Bosse et al. (2012) improve on this shortcoming by using surface elements (named *surfels*) for local registration. In the mobile mapping approach presented, surfels match when they are spatial and temporal neighbors and are found to have similar surface normals. While this correspondence search requires more preprocessing, less false positives slip into the computed correspondences, thus enabling more reliable registration.

On the other hand, global registration methods do not require any estimates of relative poses. Instead, they independently search both scans for features, then try to find correspondences between these. When a sufficient number of corresponding features has been found, a transformation is constructed to spatially align them. As a last step, a local registration algorithm like ICP can be used to refine the registration.

The most challenging part of global registration algorithms remains to be the reliable extraction of meaningful features. Numerous different approaches to feature-detection in point clouds have been suggested in the literature. The most common use spin images (Johnson, 1997), Extended Gaussian Images (Horn, 1984), Fast Point Feature Histograms (FPFH, Rusu et al. (2009b)), Depth-Interpolated Local Image Features (DIFT, Andreasson and Lilienthal (2010a)), as well as planar patches (Pathak et al., 2010). For a better introduction to the problem of shape correspondence and registration algorithms, the reader is directed to Van Kaick et al. (2011). In many commercial applications, global registration is achieved by solving a coarse GNSS-derived position during each scan and using dedicated control points in the form of white spheres (shown in figure 2.6(a)). These are placed in the environment before scanning commences and, due to their special shape and reflectivity, can be automatically and reliably detected in the resulting data. After finishing acquisition, software globally positions the point clouds using the captured GNSS position, then locally aligns the scans by aligning the extracted control points. As presented in figure 2.6(b), the control points remain part of the merged map and potentially obstruct interesting geometry, motivating research for automatic and marker-less alignment of scans.

(a) (b)

Figure 2.7: (a): A Schiebel S-100 Camcopter UAS with an integrated Riegl VQ-820 LIDAR sensor and Trimble/Applanix AP50 GNSS/Inertial navigation system. (b) Ground-based mobile indoor mapping. Images ©: (a) 2012 Schiebel Corporation, (b) 2010 Darmstadt Robot Rescue Team

2.3.2 Mobile scanning

Maps originating from static scans have traditionally exhibited superior accuracy when compared to those produced by mobile mapping. However, mobile robotics does not accommodate static scanning: the stop-scan-move cycles forced on the intrinsically mobile mapping robot impose many disadvantages: most importantly, they considerably increase the time required for mapping and decrease energy efficiency. Some robotic platforms, like the one presented in this thesis, simply cannot support a static scanning pose, so that mobile scanning remains the only option.

Many challenges surface once the assumption of a static scanning pose has been abandoned: primarily, it is no longer possible to create a self-consistent point cloud with any scanner hindered by a non-negligible integration time. Strictly speaking, this holds true for any scanner, but as long as the time required for a single scan is dominated by light's ToF (as in e.g. structured-light or Flash-LIDAR sensors), the error resulting from concurrent motion often remains imperceptible. However, in the case of actuated laser scanners where a complete scan can take up to several minutes, motion must be accounted for.

Recovering the trajectory[1] of robot and sensor is a very difficult challenge; any imprecisions manifesting in the scanner's path directly translate to inconsistencies in the resulting point cloud. Conceptually, there are three main approaches to solving a moving robot's position and orientation over time in the outdoors:

The first approach is to mount a GNSS antenna/receiver setup to the robot, so that both position and time can be determined with high precision and frequency. Using RTK-GNSS and a reference station, positions can be solved with centimeter-accuracy. In the case that correction network infrastructure is available within baseline length

[1] defined as the path that a moving object travels through space, as a function of time.

limitations, dedicated GNSS reference stations can be substituted (Rizos and Han, 2003). But there are inherent disadvantages, namely that rotation angles must be constrained in order to keep the receiving antenna pointing upwards, and that GNSS alone can only compute position, but not orientation. One way to solve this problem is known as *GNSS-attitude* and simply employs three antennas: once the baselines between all antennas are known, the orientation of the object can be inferred from the locations solved for each (Cohen, 1996; Li and Murata, 2002). The accuracy of the orientation sensed in this setup increases with the length of the baselines, and the ensuing compromise between compactness and accuracy is often troublesome for mobile robots. Furthermore, this setup is traditionally expensive and fails immediately when one receiver suffers from insufficient reception of satellite signals.

The second approach is the combination of a precision GNSS antenna/receiver and an inertial measurement unit (IMU), as shown in 2.7(a). The system measures time and position using the GNSS module and, using a filter, fuses the results with accelerations and rotations sensed by the IMU. Inertial and GNSS systems have shown to perform very favorably in combination, because they show very complementary error characteristics: GNSS measurements deliver absolute and drift-free positions, but only support sampling rates of about 25Hz and do not contain information about orientation. Inertial measurement units on the other hand are suitable for sampling motion with high datarates and astonishing short-term accuracy. When combined through a software filter, the resulting measurements inherit each system's advantages, yielding precise and drift-free position and attitude data at high frequencies (Li et al., 2012). The quality of the resulting data is traditionally dominated by the quality of the IMU, which often is the most expensive part of the navigation system. When using tactical-grade IMUs, inertial data can also be used to speed up the GNSS carrier-phase ambiguity search (Yang and Farrell, 2001). Further advantages are the relative compactness compared to the first setup as well as a tolerance against short GNSS outages. For enhanced precision and reliability in GNSS challenged environments, combinations of GNSS-attitude and inertial navigation systems (INS) are also being researched and applied (Hwang et al., 2005; Eling et al., 2013).

Lastly, there is self-localization and mapping (SLAM), which requires no GNSS/INS at all. Instead, data from odometry sensors, cameras or IMUs is used to integrate the robot's motion over a short amount of time. Whenever the range sensors deliver new data, the estimated pose is used as an initial seed for point cloud alignment. The data is then associated with and compared to previously recorded scans, allowing the estimate to be refined. This approach is actually quite similar to those described in section 2.3.1, with the important addition of real-time execution. A good example is Elseberg et al. (2012), who use a Riegl VZ 400 survey-grade 3D laser scanner and an XSens MTi IMU to recover the sensor's trajectories with encouraging results. Taking the concept of *mobile* scanning even further, Bosse et al. (2012) have introduced a handheld scanning platform named *Zebedee*. After integrating IMU and two-dimensional LRF data, the system employs extensive post-processing and loop-closure to produce highly precise

point clouds. Although the most popular, application of SLAM is not limited to range sensors; other combinations like inertial sensors and visual odometry have also been suggested (Roberts et al., 2002; Angelino et al., 2012).

SLAM has proven very valuable in structured indoor environments and successful probabilistic approaches are well-documented in e.g. Dissanayake et al. (2001) and Montemerlo et al. (2002). Figure 2.7(b) shows an application in a robotic rescue context. On the downside, SLAM also suffers from numerous shortcomings: when localization relies on constant and valid updates from the range sensor, aerial vehicles must accept the sensor's range as their service ceiling. With SLAM, the position is prone to drift over time because of subtle errors in data association, which can be partly corrected using loop closure detection. Some environments do cause the robot to revisit previously seen places during exploration, giving sufficient opportunity for loop closure and a precise map (Thrun and Montemerlo, 2006; Bosse and Zlot, 2008). Other tasks (e.g. mapping of powerlines or pipelines) do not. Even when SLAM is capable of creating consistent outdoor maps, they still lack an important property when compared to those created using GNSS-based sensors: they are not georeferenced, so spatial association with maps created at other times or by other robots remains difficult.

In this thesis, a GNSS/Inertial navigation system is used to reconstruct the robot's path. This way, the cost and complexity of GNSS-Atittude setups is avoided, and localization does not depend on SLAM, which is comparably fragile in the outdoors and would introduce further constraints on the robots motion. With the range sensor firmly attached to the robot, it's trajectory can be computed by transforming the INS's trajectory using a simple homogeneous transformation matrix. Once the poses of a scanning sensor's trajectory are known and temporally referenced to its range measurements, the sampled ranges can be transformed into a global coordinate system, once again producing a consistent map.

However, only a small subset of scanners support synchronization with other measurement devices in order to obtain precise timing of individual range readings. The Hokuyo UTM-30lx scanner provides a coarse timing synchronization by means of a signal that indicates the laser's angular position once per revolution, whereas e.g. the Velodyne HDL32E and most survey-grade scanners provide timing information for each individual range with precision in the nanosecond range by synchronization through a dedicated pulse-per-second (PPS) signal. For further details, please see section 3.1.4.

2.4 Map representation

In the context of mobile robotics, a map of the surroundings has traditionally been the most important part of the world model. Depending on the application, the world is often represented in *metric*, *topological* or *semantic* maps.

Metric/Geometric maps are data structures that contain direct information about the

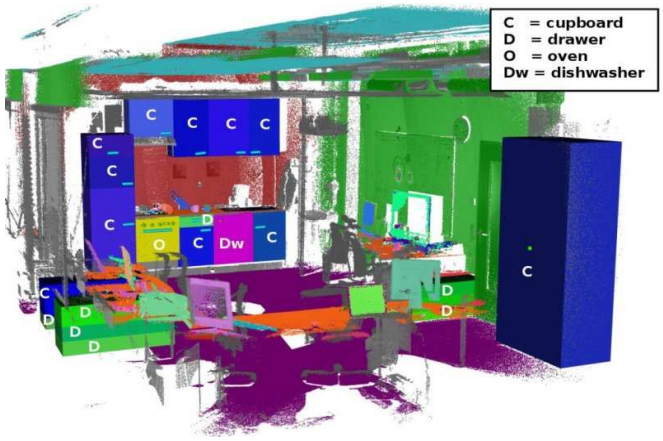


Figure 2.8: A semantic 3D object map, consisting of a metric map that was later annotated with object information. This information requires knowledge of the world and is highly domain specific. Reprinted from Rusu et al. (2009a). Image © 2009 IEEE.

world's geometric shape. Objects are referenced using coordinates relative to a common frame. This type is usually preferred for applications that require high accuracy, e.g. when the map is the end product or used for self-localization or object manipulation.

Topological maps represent the environment as a graph: while vertices correspond to interesting places that the robot needs to be aware of, edges stand for a relationship between them (Remolina and Kuipers, 2004). Although edges can encode any relationship in theory, the relationship is most often spatial in practice and – with some constraints – can also be used for localization (Choset and Nagatani, 2001).

Semantic maps often build on geometric maps, enhancing them by adding information about objects contained. Once this information is available, it can be used to infer further information in reasoning and planning systems (e.g. "This room contains a stove, so it is not a bathroom.", Galindo et al. (2005)). Producing such maps is a very complex endeavor, as it requires that objects of interest be detected in and segmented from the "background" of the map (Nüchter and Hertzberg, 2008). This procedure requires knowledge of the world, which usually constrains solutions to small domains: point clouds annotated with household objects are described in Rusu et al. (2009a); Blodow et al. (2011); Pangercic et al. (2012) (an example is shown in figure 2.8), while Sengupta et al. (2012) create maps of street inventory using classic stereo vision.

This thesis focuses on metric maps, as they are most suitable to create precise representations of physical shapes in outdoor environments. Representing physical shapes in computer memory is a complex problem, and a wide variety of encoding strategies exist to solve it. In general, encodings should exhibit as many of the following properties as possible:

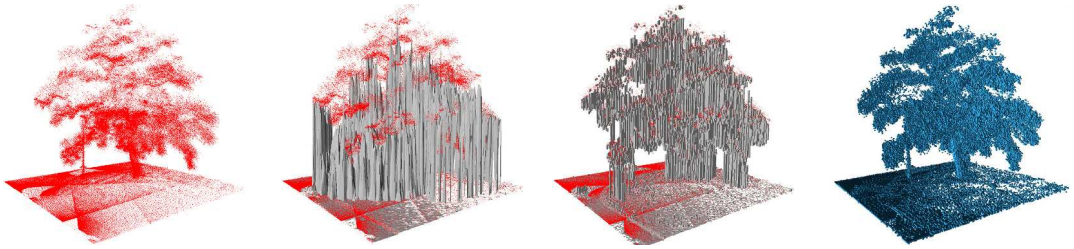
- Compact. The map should store geometric information in as little memory as

possible. This is obviously desirable for storage, but also for processing of the map, as less data commonly translates to smaller memory bandwidth requirements.

- Dynamic. Integrating new information into any region of the map should be efficient at any time, and not require large amounts of data to be reorganized.
- Unbounded. In autonomous exploration, the extents of the final map are unknown when it is created. Having to predefine a map's extent severely limits it's applicability to the task and would require several workarounds.
- Expressive. The map should be able to represent any shape that exists in the real world. Even complex geometry like trees or overhangs should be encodeable.
- Inherent processing. Some map structures require processing steps during integration of sensor data, which automatically improve the quality of the data. For example, some encodings effectively remove noise from sensor readings, while compressing the data at the same time.
- Reusable. When planned carefully, some map data structures can be reused for other problems, such as collision avoidance or path planning.
- Searchable. A map is often not the end product, but an intermediate step in a longer processing pipeline. Searching a geometric primitive's spatial neighbor is one of the the most fundamental operations in metric maps and thus, should not be computationally expensive.

A common map data structure are point clouds (depicted in figure 2.9(a)), which have been used successfully by many robotic mapping systems (Nüchter et al., 2007b; Borrmann et al., 2008; Andreasson and Lilienthal, 2010b). Point clouds simply encode the global positions of the points sensed by the scanner. Usually, a point is represented by its x,y and z coordinates, which are saved with single float precision. In terms of memory management, point clouds are either implemented as a struct of arrays (SoA) or an array of structs (AoS), depending on the underlying hardware and expected access patterns. Either way, this representation allows for straightforward, yet extremely fast visualization when executed on graphics processing units (GPUs). Point clouds are unbounded, allow for arbitrary and variable resolution, and make insertion of new data trivial. However, they do not encode information about surface normals, do not filter sensor noise and grow linearly with the number of recorded points. Processing point clouds to ensure even spatial sampling densities is often necessary, but requires many searches for neighboring points, which is either complex or time consuming.

Grid maps, or evidence grids, have long been a popular data structure for geometric maps (Elfes, 1989). They span a bounding box in space and evenly subdivide it on all axes that are to be mapped, in effect creating grid cells (also known as *voxels*). Each cell then corresponds to a memory location that contains data about the space it encloses: the most common specialization is the occupancy grid map, in which every cell records whether it is occupied, empty, or still unexplored. Spatial indexing is



(a) (b) (c) (d)

Figure 2.9: A 3D representation of a tree, visualized after saving in different data structures: (a): A simple set of all captured points sensed by the scanner, named a *point cloud*. (b): A heightmap, which stores height-information for every cell in a two-dimensional grid. (c) Multi-level surface maps extend heightmaps by allowing multiple heights to be stored for every cell. (d): Voxel-based maps subdivide 3D-space into cubes. More advanced implementations can decrease the cube-size (increase resolution) to represent more complex geometry. Reprinted from Wurm et al. (2010). Images © 2010 IEEE.

comparably straightforward, so that any location's cell can be computed quickly from few parameters, and determining any cell's memory address is just as fast. One of the most important advantages is that its memory footprint remains constant during integration of sensor information (described in detail in Moravec and Elfes (1985)), which heralds huge benefits for data-parallel implementations. On the downside, the extent of a grid map must be fixed at creation time, and the memory requirement grows polynomially when increasing resolution. While uniform cell sizes simplify data access enormously, they often make large scale grid maps inefficient: large regions that are completely free or occupied are sampled and stored with the same resolution as regions with high geometric complexity.

Another map variant are tree-based data structures, such as octrees or kd-trees. These allow far more efficient data storage by providing adaptive resolution and conceptually allow extents to be expanded after initialization. However, these advantages come at the cost of more complex memory layout and thus, more complex management-algorithms and uncoalesced access patterns during cell-lookup. For this and other reasons, implementing dynamic, tree-based maps on data-parallel hardware remained extremely difficult until very recently, when dynamic parallelism was introduced in the ISA of NVIDIA's newest-generation Kepler GPUs.

Abstracting further, grid maps can be reduced to two dimensions, with each cell encoding the height of the contained geometry. This is called a height map and inherits most of the grid map's properties. Such a map is visualized in figure 2.9(b), revealing this encoding's most important shortcoming: information about overhangs cannot be encoded, effectively distorting many common shapes like trees, tables and bridges. In practice, this often constrains applicability to large-scale, low-resolution maps, such as digital elevation models (DEMs).

Triebel et al. (2006) introduce *multi-level surface maps*, which extend two-dimensional

height maps by allowing multiple heights to be stored in each cell. This is commonly named a 2.5D map, as it is not quite as expressive as a real 3D representation. Figure 2.9(c) shows an example of such an MLS map, and the inventors subsequently implemented self-localization employing this data structure (Kümmerle et al., 2008). Still, Wurm et al. (2010) point out that 2.5D maps are unable to store free or unknown areas in a volumetric way, limiting their use for localization and exploration.

During software development for this thesis, octrees were first used for map representation. During porting to a GPU-based implementation, octrees were replaced by other established data structures, namely point clouds and 3D grid maps. These structures are far less complex to manage, which translates to much faster execution on the target platform.

2.5 Next Best View Planning

While self-localization and mapping are essential capabilities for mobile robots, further skills are required to perform autonomous exploration: one of them is planning motion. With the aim of exploration, motion must be optimized towards maximization of the system's information gain.

At its core, the problem is one of visibility, which is well-researched and documented in the field of computational geometry (Ghosh, 2007). Building on the concept of visibility, the aim is to find sensor poses (and, in temporal extension, -trajectories) that cover as much of a scene as possible. For a good introduction to view planning problems and algorithms, the reader is directed to Chen et al. (2008).

In 1987 O'Rourke published "Art Gallery Theorems and Algorithms". The *Art Gallery Problem* describes the real-world problem of guarding an art gallery with the minimum number of guards that, together, can see all of the gallery. Although the book still serves as a good reference and base for many related approaches, the algorithms presented only address rather small parts of the overall problem, as they mostly apply to polygons in two-dimensional space. Even though the definition presumes the map to be known a-priori, the problem is NP-hard (Khopkar, 1997). Thus, this assumption means that the presented solution can not be applied to NBV planning in unknown environments.

Computing the optimum sensor placement in order to maximize visibility is a generalization of the Art Gallery Problem, and was named the *next-best-view* (NBV) problem by Connolly (1985). While SLAM has been researched intensively in the past decades, next-best-view planning has not received nearly as much attention.

Depending on the task at hand, *Best* can have many different meanings, so NBVs might be computed with the aim of optimizing any combination of properties (e.g. mapping speed, map completeness, low energy consumption in the process etc.). Although called the next-best-*view* problem, its solutions often lend themselves well to other scenarios:

for example, robotic vacuum cleaners also need to cover an area as quickly and efficiently as possible – except that instead of a sensor, an actor's trajectory must be planned. In this context, the problem is known as optimizing coverage; Schwager et al. (2009) clearly demonstrate how closely these concepts are related.

Null and Sinzinger (2006) introduce a dichotomy: the interior NBV problem places the sensor within the geometry that is to be mapped - indoor-scenarios are an obvious instance of this class. The exterior NBV problem is defined by the sensor being placed outside of the object to be reconstructed, like a statue or a typical household object. Here, problems arise mainly from self-occlusions of non-convex objects.

Similar to how humans approach this challenge, a frontier-based technique is commonly used in many NBV problems (Yamauchi, 1998; Joho et al., 2007; Shade and Newman, 2011), because it yields sensor-poses located between known and unknown regions. On the one hand these poses offer safe reachability, because the path planner can compute a trajectory through known parts of the environment. On the other hand - given the pose is oriented towards the unmapped environment - it will allow the sensor to deliver valuable information, advancing the mapping process. In order to compute such a pose, knowledge about frontiers has to be derived from the underlying data structure. When using two-dimensional grid maps, "frontier cells are defined as unknown cells adjacent to free cells and this way a global frontier map can be produced" (Mobarhani et al., 2011).

Null and Sinzinger (2006) compute surface normals either from voxel structures or reconstructed mesh surfaces (an expensive operation in itself), letting the sensor align itself with normals at the border between scanned and unscanned regions. For both problems, the paper presents algorithms based on voxel data structures and ray casting that limit the range sensor to a constant height. Because their solution to the exterior NBV problem limits the possible angles of the sensor-pose to point towards the object, it is orders of magnitude faster than the algorithm for interior NBV planning, which cannot limit the search space in such comfortable ways. Still, even the faster algorithm requires almost 12 seconds of processing time on a CPU for a grid of 125k (50^3) voxels, so increasing the grid's resolution quickly proves problematic. Furthermore, the strict separation of interior and exterior planning doesn't fully accommodate the case of a robot mapping an outdoor environment, which can contain geometry from both classes.

View planning algorithms operate on spatial data structures, and searching these structures leads to memory-bound programs. Because memory throughput hasn't increased as fast as required by the integration of the third dimension, a common strategy is to reduce the dimensionality of the learned map. In simulation and real-world experiments, Strand and Dillmann (2010) condense a scanned point cloud to a two-dimensional grid for navigation and planning. At the same time, the ability of exploring unstructured environments with slopes and overhangs is lost.

Blaer and Allen (2007) research ground-based outdoor reconstruction based on building voxel-grids from acquired data and ray traversal for NBV computation, requiring an a-priori "2-D map with which it plans a minimal set of sufficient covering views". To manage large-scale outdoor scenes, the voxel's sizes are increased to one meter cubed and they are marked as seen if they contain at least one point from the iteratively acquired point cloud. Candidate NBV locations are computed by finding occupied voxels that intersect the ground-plane, but are marked as free on the a-priori two-dimensional map. New sensor poses are then generated by using ray-tracing to count the number of "boundary unseen voxels" (unseen voxels adjacent to at least one empty voxel) visible from all candidate locations.

Three-dimensional environment mapping is often implemented using laser scanners and time-of-flight cameras, so point clouds are a very common type of sensor data. Unfortunately, information about exploration boundaries is hard to generate from point clouds. Applying a plain spatial subdivision to point clouds to form a 3D occupancy grid map might appear as a logical next step, extending Mobarhani's definition (see page 30) into the third dimension. Constantly updating such a grid quickly becomes a burden on the processing pipeline, as all rays scanned by the laser scanner have to update all the cells they travel through. This makes resolutions in the centimeter range quickly become unfeasible. Furthermore, a height limit has to be imposed manually to keep the robot from mapping unknown (and empty) regions in the sky.

Because the presented experimental airborne platform features a flight-time of only 15 minutes, NBVs need to be determined quickly. In contrast to generating a single NBV for a given input, computation of multiple NBVs, sorted by achievable information gain, is preferred. This enables creation of trajectories that include all NBV-derived waypoints in an order optimized to allow the robot to reach all of them in minimal time.

As the UAV is required to navigate through unstructured and complex 3D environments, approaches that reduce the problem to two dimensions (for frontier generation or reduction of complexity) were avoided. Other approaches failed to meet requirements, e.g. by requiring a-priori knowledge of the map, by limiting applicability to either the interior or exterior NBV problem, or by necessitating computationally intensive pre-computations that are slow to execute on graphics cards. For this reason, section 4.2 describes how a custom approach to next-best-view generation was first tested using a CPU-based proof-of-concept, then implemented on the GPU to work in real-time.

2.6 Autonomous exploration

Autonomous exploration, finally, is the combination of localization, mapping, path planning, obstacle avoidance and motion control.

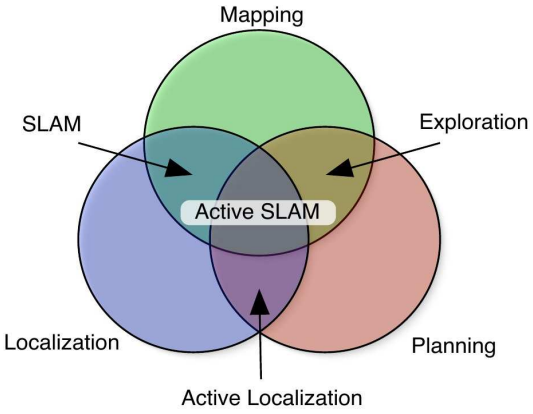


Figure 2.10: Subtasks of automatic model learning, from Fairfield (2009)

More constraints in exploration emerge when taking the limited maneuverability of most UAVs into account. Due to the limited bandwidth and authority of trajectory-controlling actuators, path-planning and motion control must be suitable for the respective UAV (Singh and Fuller, 2001; Machmudah et al., 2010).

When localization is performed through SLAM, a compromise must be made between reobservation of known landmarks (optimizing localization quality) and exploration of unknown space. Bryson and Sukkarieh (2006) define a mutual information gain resulting from traveling to a goal as "the difference between the entropies of the distributions about the estimated states before and after making the observations". This definition requires knowledge about the probability distribution after making the observations at a given goal in unmapped environment, which cannot be foreseen and thus, must be simulated under the assumption that "there exists a certain available feature density [...] in terms of average number of features per map grid area". Although the authors later present a solution for fixed-wing UAVs using visual/inertial SLAM that performs well in simulation (Bryson and Sukkarieh, 2008), the system simulates application on a macro-scale (grid-map with 100m cell size) and essentially requires a collision-free environment that can be well abstracted into two dimensions.

As shown in figure 2.10, Fairfield (2009) groups the basic tasks of localization, mapping and motion planning into the higher-level activities SLAM, exploration, active localization and, unifying all three tasks, active SLAM. Although the basic tasks are rather well defined, both names and definitions for the higher-level activities vary between authors (compare e.g. Makarenko et al. (2002)). In the end, localization, mapping, motion planning and motion control are strongly interdependent and one cannot reasonably expect a diagram of three circles to capture the complexity of autonomous robotic exploration. Despite these and other shortcomings (e.g. it is doubtful whether localization and planning (i.e., "active localization") can exist without a map), the diagram still serves as a helpful guidance for classification of the different approaches presented.

There have been numerous publications surveying active perception planning for reconstruction and inspection. Makarenko et al. (2002) include all subtasks of autonomous exploration into the problem domain and - while focusing on localization quality - attempt to find a balance with information gain and motion cost. However, the algorithm relies on assumptions about unknown parts of the map and was evaluated using only simulation of a robot in a two-dimensional environment. The text further notes that "while undoubtedly the optimal plan must take into account the expected integral payoffs (e.g. information gain along the path), the complexity of the problem effectively precludes this approach".

Reasoning over yet-unexplored spaces using probabilistic methods, like in Potthast and Sukhatme (2014), yields helpful output as long as the environment to be scanned follows the assumptions made beforehand, e.g. flat table-tops and non-degenerate shapes. Unfortunately, real-world outdoor scenarios are not necessarily flat and often more complex than table-tops with cutlery.

Scott et al. (2003) classify methods as either surface-based, volume-based or global. As detailed in chapter 4.2, we detect the absence of information (missing geometry) using a surface-based approach, while rating possible information gain of sensor poses using volumetric data structures.

When using GNSS, the quality of localization does not depend on the density of sensed features in the surroundings and the ability to associate them, as it is commonly the case with SLAM algorithms. However, this does not mean that localization quality is constant throughout the environment. Indeed, the number of satellites in view as well as their geometric constellation is an important factor for the accuracy of the computed position. When mapping very close to buildings or in-between high-rise architecture, obstructions of satellites become so important that motion planning must consider both satellite orbits and local geometry. This is a good example of additional constraints that must be considered when implementing active localization for a mobile robot.

As experiments in chapter 5 show, the platform's self-localization is sufficiently precise and reliable in outdoor environments. Thus, we can optimize the exploration strategy towards map completeness and mapping duration instead of optimizing for localizability.

2.7 Summary

As presented in figure 1.2, a wide range of functionality must be integrated in order to implement autonomous airborne exploration. This chapter critically assessed the state of the art in related fields.

Regarding the most fundamental part of the platform, many unmanned aerial vehicles are available on the market today. Fortunately, a few remain when requirements re-

garding maximum weight, payload, flight-time and extensibility are considered. For this thesis, a Mikrokopter Okto2 multirotor drone is selected to carry the sensor payload. This UAV is presented in the next chapter, section 3.1.1.

Besides discussing advantages and disadvantages of different range sensing technologies, section 2.2 also serves as an explanation for the selection of LIDAR sensors, which are available as commercial off-the-shelf hardware. Section 3.1.4 will describe the properties and technical integration of the chosen sensors into the platform. The challenges of 3D scanning, registration and self-localization, as well as current concepts for appropriate solutions were introduced in section 2.3. As explained above, localization of the platform is achieved using a commercially available GNSS/Inertial navigation system. Its integration is discussed in section 3.1.3 and chapter 5 evaluates its reliability and accuracy.

Map representation is an important foundation for any robotic mapping system, as computer memory capacity and throughput are limited in real-world systems. The choice of data structures is one of the most fundamental, as it has strong influence on runtime properties such as speed, energy efficiency, and scale limitations. As such, section 2.4 also is a helpful introduction to chapter 4, because it motivates the choice for point clouds as the data structure backing most algorithms presented later.

Numerous techniques for view planning in indoor and outdoor environments, represented using either two- or three-dimensional datastructures, have been discussed in section 2.5. Limited by the computational power of today's hardware, all of them must compromise between speed, robustness against more complex environment geometry, correctness, and other desirable aspects. The possibility to improve on this state by leveraging the availability of newly-available, highly-parallel computing architectures has motivated a core part of this thesis. Even though some parts of the NBV algorithms presented in chapter 4 quite obviously took inspiration from their predecessors, others differ significantly. And while compromises must still be made, this approach allows for real-time airborne exploration in outdoor environments.

The next chapter presents the system's architecture on a lower level than given in chapter 1, and describes the integration of its components into the experimental platform. The software architecture is presented in the second half of chapter 3, with the employed algorithms being described in the following chapter 4.

Chapter 3

Experimental Platform: Concept and Architecture

The first chapter started with a discussion of this thesis' motivation and its goals. A diagram was presented (see figure 1.2), showing the abstract functional components required, as well as the interaction between them. The second chapter analyzed the state of the art in related fields and concluded that some functionality can be provided by off-the-shelf components and pre-existing algorithms. However, other functionality is not provided by the state of the art and thus, requires research and development.

The first part of this chapter will introduce the hardware that was integrated to become the experimental platform. Building a robot capable of autonomous exploration requires a combination of many components: a UAV, computers, networks, sensors and countless smaller parts to integrate them. All of these must interoperate as planned, and because many of them are part of modules that are critical to safety in flight, they must work reliably not just most of the time, but always. This is an important realization for researchers that, like the author of this text, previously worked with ground-moving robots: there is no emergency-stop switch for UAVs.

Once the hardware was delivered, it was integrated and tested, revealing that most components did not behave as specified. As an example, the on-board computer emitted strong electromagnetic interference (see section 3.1.5), preventing the navigation system from performing carrier-phase measurements and thus, providing accurate positioning. Once this problem was resolved, positioning was accurate, but then the navigation system turned out to be unable to perform kinematic alignment. Weeks later, its manufacturer confirmed that an error in the system's firmware prevents kinematic alignment in accurate GNSS modes, and quickly fixed the problem. In the end, the time invested paid off, as all parts sourced at the beginning of the project are used in the final system. Indeed, only replacement parts, batteries and mounting materials were purchased in the years following.

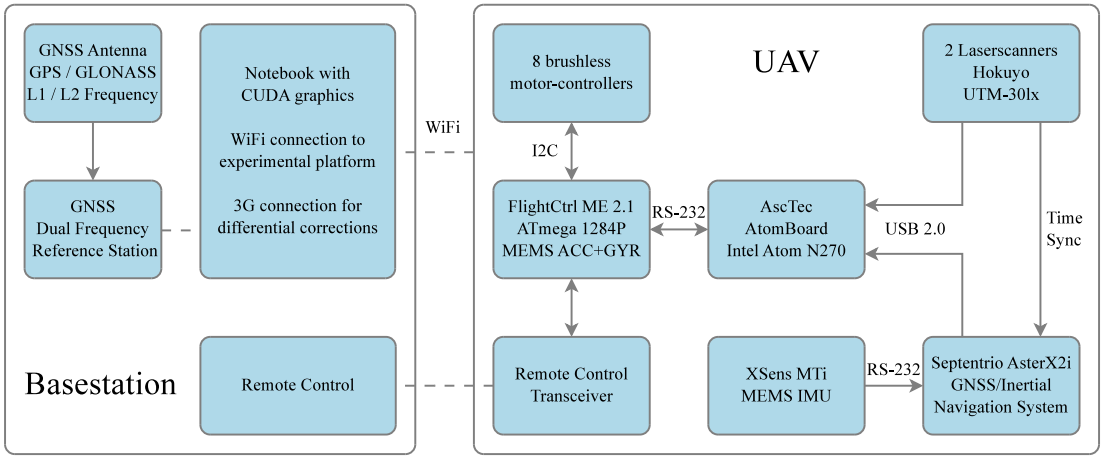


Figure 3.1: Diagram of hardware used for both base and rover as well as communication channels connecting them.

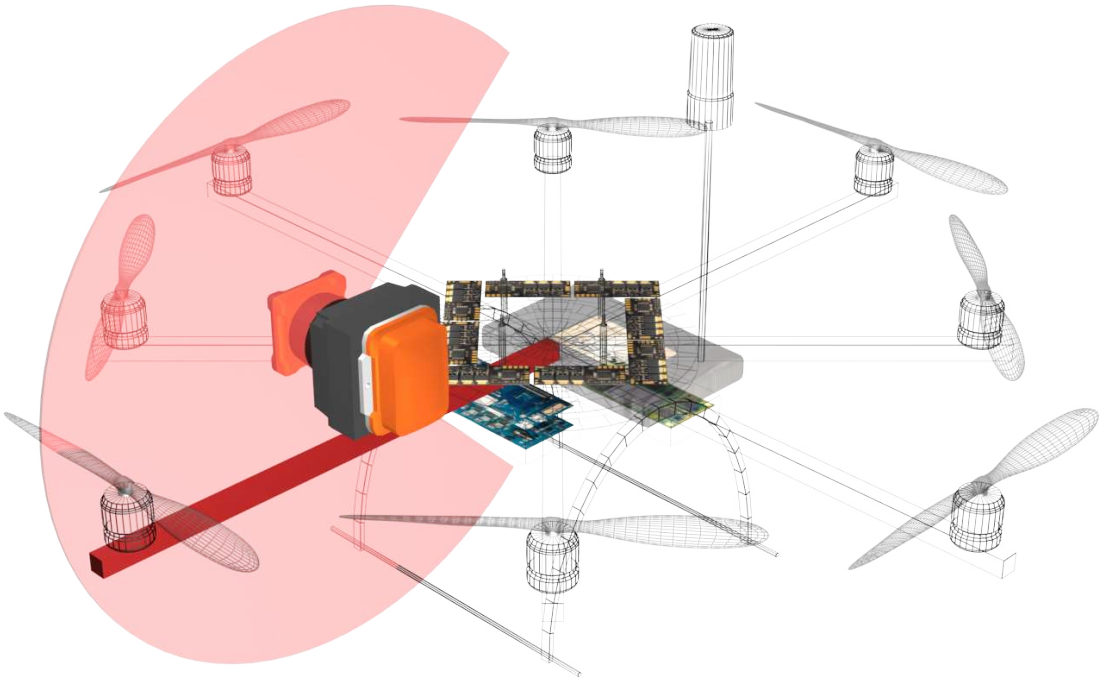
The following sections portray each hardware component, giving a detailed account of its properties, application, limitations and how it integrates into the experimental platform.

At the same time, a set of software was developed: first, a proof-of-concept implementation was created, allowing a virtual UAV to fly and map in simulation. With delivery of the hardware, integration and software development were worked on in parallel. Device drivers were written and another program was created to control the sensors and the UAV. The resulting software architecture is discussed in section 3.2.

3.1 Hardware

In chapter 1, the platform's functional goals were defined and diagram 1.2 presented an abstraction of necessary components and their interaction. Figure 3.1 presents a more low-level diagram of the hardware used in this thesis. The left hand side shows the GNSS reference station and base station notebook, which are necessary parts of the equipment on the ground. The right hand side presents the UAV and its payload, which is described in detail in the following sections. Some functionality is implemented directly through hardware components, e.g. range sensing is realized through laserscanners and self-localization by GNSS reference station and GNSS/Inertial navigation system. Other functionality, like motion control, cannot be mapped to hardware as easily, as it is implemented using software running on multiple hardware components.

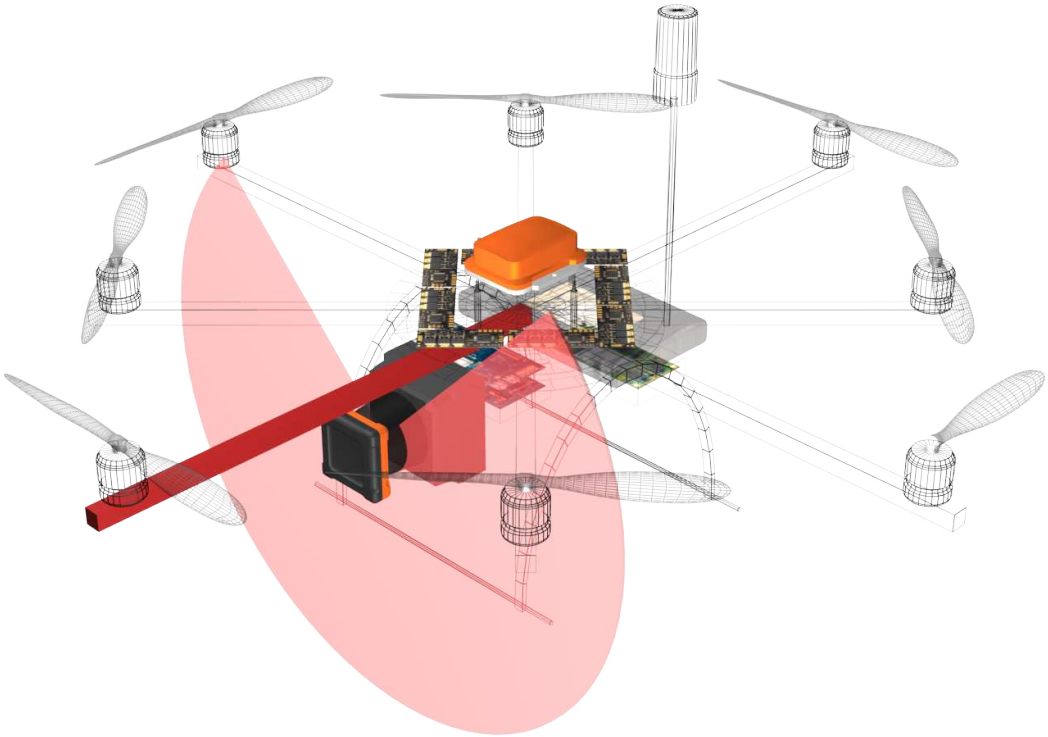
(a) The first configuration: the laser scanner's field of view is oriented vertically and the inertial measurement unit is fixed to the scanner for precise alignment. This arrangement supports both mapping and obstacle avoidance during flight with a single scanner by constantly yawing during flight. Unfortunately, the inertial navigation system's fusion algorithms cannot solve the vehicle's heading reliably when heading and bearing differ during large parts of the flight.



(b) Visualization of the scanner's field of view in the above configuration, showing occlusions induced by vehicle geometry. As reflections with a distance below the vehicle's radius are ignored, reflections from parts of the UAV (including its propellers) do not harm point cloud quality in the form of outliers.

Figure 3.2: First hardware configuration of the experimental flying platform with mounted GNSS-antenna and -receiver, laser scanner, inertial measurement unit and processor-board. The red boom points forward.

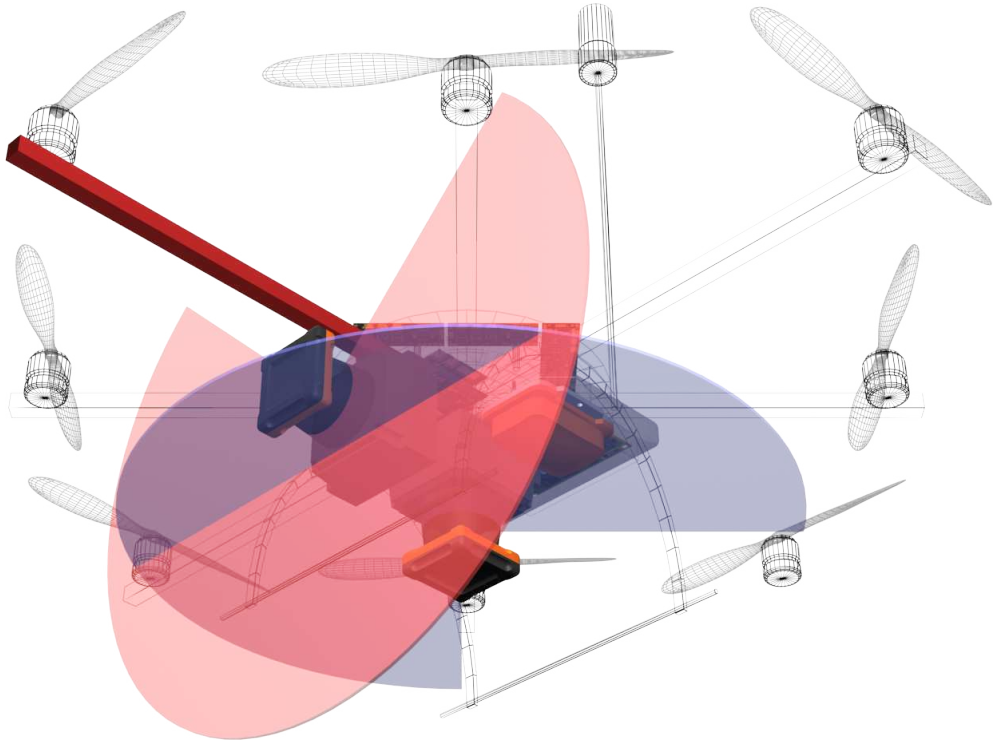
(a) Second configuration with the scanner's FoV oriented downwards and perpendicular to principal direction of travel. The inertial measurement until is mounted using vibration dampeners for improved GNSS/inertial fusion performance.



(b) Visualization of the scanner's field of view in the above configuration, showing occlusions induced by vehicle geometry.

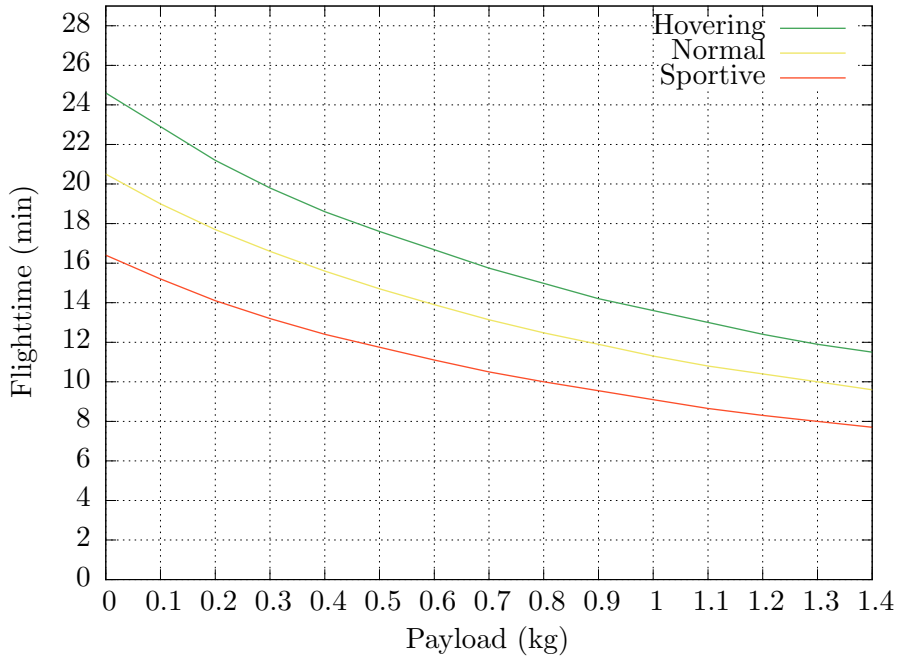
Figure 3.3: Second hardware configuration of the experimental flying platform with mounted GNSS-antenna and -receiver, laser scanner, IMU and processor-board. The red boom points forward.

(a) The current setup includes a second laser scanner for obstacle avoidance and mapping. Both laser scanners and the IMU are mounted together with the battery on a stiff and lightweight carbon-fiber sandwich-board. The resulting module makes up 40% of the UAV's weight and is fixed to the vehicle's frame using vibration dampeners. This yields less drift in the navigation system solution, more precise sensor alignment and better crash survivability.

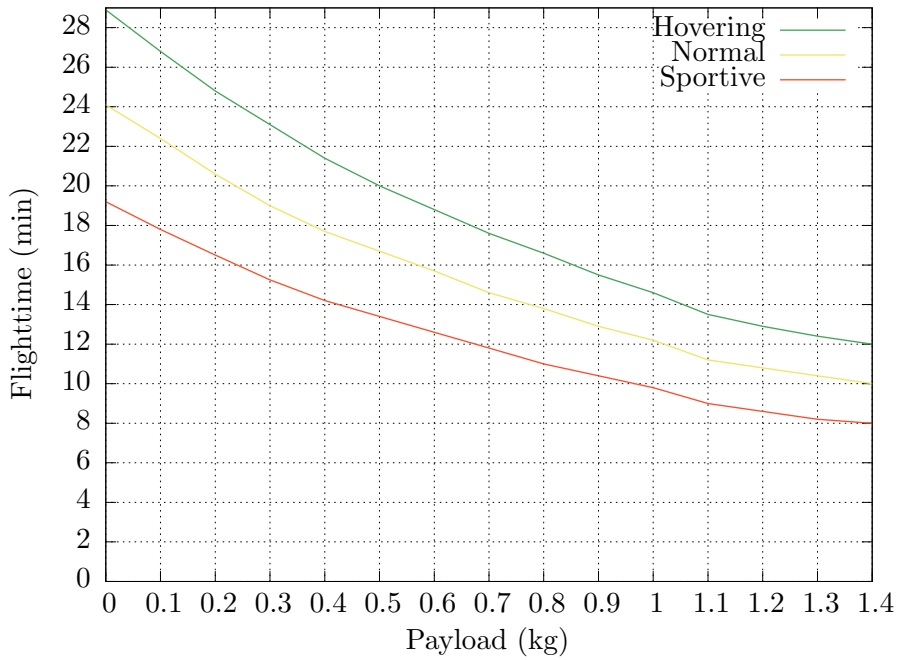


(b) Visualization of the scanner's field of view in the above configuration, showing occlusions induced by vehicle geometry.

Figure 3.4: Current hardware configuration of the experimental flying platform with mounted GNSS-antenna and -receiver, laser scanners, IMU and processor-board. The red boom points forward.



(a) Flight time of an Okto2 model (8 motors, total weight 1770g including a 4-cell 5Ah LiPo battery) versus payload.



(b) Flight time of a QuadroXL model (4 motors, total weight 1480g including a 4-cell 5Ah LiPo battery) versus payload.

Figure 3.5: Graph depicting the achievable flight times versus the payload carried by differently-sized multirotors, courtesy of the Mikrokopter project. While these numbers are of theoretical nature, they match the experienced real-world flight-times very closely.

3.1.1 Unmanned Aerial Vehicle

The UAV is built upon an "Okto 2"-multirotor helicopter from the Mikrokopter project[1] (see figure 3.2(a)). It is propelled by eight drives (i.e., motor/propeller combinations) that are installed on booms of two different lengths. Towards the front, rear, left and right, they are fixed on long booms about 38cm away from the center. In the half angles, the other four motors are fixed 28cm away from the center using short booms, resulting in a diamond-like shape when observed from above.

When rotating, each drive causes two forces to act on its boom and thus, the vehicle: first, the downward-oriented thrust generated by the rotating propeller pushes the vehicle upwards. The relation of electric power flowing through the motor, resulting rotational velocity and thrust is non-linear and will be briefly discussed in section 3.2.1 B. Secondly, the drive also generates a torque that – if not neutralized – causes the platform to start rotating the direction opposing that of the drive.

For this reason, the drives installed on the longer booms rotate clockwise, with the other four rotating counter-clockwise. Due to this interleaved arrangement, the total torque exerted by motors and propellers on the platform is kept neutral during hover, roll and pitch maneuvers. Yawing the platform can be accomplished by decelerating all motors rotating in one direction, while accelerating the opposing group to make up for the loss of thrust. Effectively, this will slowly unbalance the torques of equal magnitude, but opposing directions affecting the platform. The angular velocity of the yawing maneuver can then be controlled by adjusting the difference of rotational velocity between these two motor-groups.

The Mikrokopter is also offered with the drives mounted slightly rotated along the axes of their booms ($+5°$ for one motor group, $-5°$ for the other), so that the drive's thrust no longer points strictly nadir. While this allows the platform to yaw more quickly, it does decrease its efficiency and was not adopted for the UAV presented here.

Using a four-cell 5000mAh LiPo-battery, its eight motors enable the platform to lift a payload of about 600 grams for up to 16 minutes of normal flight. See table 3.1 for a list of components and their weight, as well as figure 3.5(a) for a graph describing the relation between payload and flight time. Judging by the weight of the payload alone, using the QuadroXL four-motor version does seem to be the obvious choice, as cost and expected flight times compare favorably to the Okto2 model. On the other hand, using a larger vehicle with more motors than strictly necessary has shown to be advantageous in many regards: in terms of flight dynamics, the mounted sensors experience a more stable, smoother flight that is less prone to wind and small turbulences. Furthermore, eight slow-running motors have shown to induce less vibrations into the vehicle than four motors of a smaller version that lift almost the same weight. Although the vehicle has never had a motor or controller fail during flight, it has lost single propellers due to contact with trees, walls and ground. As the flight behavior was not visibly affected

[1] `http://www.mikrokopter.de/`

Table 3.1: Platform components and weight

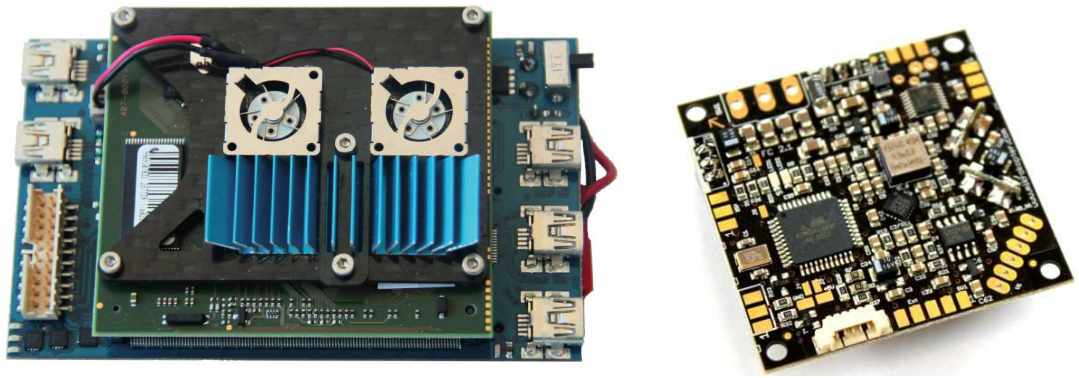
Component	**Weight (g)**
2 Hokuyo UTM-30lx	426
IMU XSens MTi	54
SensorBoard	41
AtomBoard incl. case and WiFi antennas	126
GNSS receiver incl. case	97
GNSS antenna	17
GNSS antenna cable & mount	16
Shielding (aluminum foil)	15
Sum payload	**792**
Payload	792
LiPo 4S 16.8V 5Ah	511
Mikrokopter Okto2	1180
Sum platform	**2483**

by this loss, a missing propeller was often detected only after landing. Especially during development of the high-level flight-controller, the added redundancy in the octocopter's propulsion has saved the vehicle from countless crashes, quickly paying off the higher initial investment.

3.1.2 On-Board computing

For low-level flight-control, we use the "FlightCtrl ME 2.1" flight controller, designed by the Mikrokopter project. It is based on an ATMEGA 1284P microcontroller, features MEMS gyroscopes and accelerometers for stabilization and communicates with all eight brushless motor controllers using an I^2C bus. The board also features other hardware like an atmospheric pressure sensor or servo controllers that are left unused in this project. The source code for the board's control loop is published and free to use for non-commercial purposes.

The platform also carries an on-board computer, weighing just 110 grams including its case. It contains a 32-bit Intel Atom Z530 single-core processor with two logical cores ("Hyper-Threading"), running at a clock of 1.6GHz and connected to 1GB RAM. The board hosts an SDHC card-reader for non-volatile flash memory. In contrast to Atom CPUs of the N2xx series, the card reader is not connected via USB, but directly to the system controller hub (SCH) AscTec GmbH (2010). On the one hand this is beneficial for the time synchronization of sensors (presented in section 3.1.4), as it means that read/write operations on mass storage have no influence on the timing of transactions on the USB. On the other hand, the SCH limits the maximum capacity of the SD card to 8GB, even though the SDHC specification allows for capacities of up to 32GB. Even when Using a class-10 micro-SDHC card, write-throughput it limited to less than



(a) (b)

Figure 3.6: (a): AscTec CoreExpress Carrierboard ("AtomBoard"). (b): Mikrokopter FlightCtrl ME 2.1. Images ©: (a) 2010 Ascending Technologies GmbH, (b) 2011 MikroKopter.de

3MB/s, forming the upper limit of data logging that can be practiced during flight. The device's BIOS is capable of booting from the attached SDHC card, after which the Linux kernel recognizes it as a regular block device. Ubuntu 9.04 was preinstalled on the board; over the course of this project, Ubuntu 10.10 to 13.10 were used, but generally, any x86-based Linux distribution is suitable for the device as well as the software created in this thesis. Eight USB ports are available, but some restrictions apply:

- port 1 is permanently connected to an FTDI FT2232 serial-to-USB converter chip, adding two serial ports to the AtomBoard's interfaces.
- port 3 is not powered and can be configured to be either a client or a host port in the BIOS.
- port 7 and 8 only support USB 2.0 devices, without backwards compatibility for USB 1.1.
- port 8 is shared with the mini-PCIexpress socket - as only few extension cards actually use the USB connections provided through the socket, this port is not necessarily disabled when cards are installed.

Using these USB and serial ports as well as an Atheros Ath5k-based IEEE 802.11n WiFi card in the mini-PCIexpress slot, the AtomBoard connects all devices on board the vehicle and allows for fast wireless communication with the base station, with reduced data rate even when the line-of-sight between both is obstructed.

3.1.3 Navigation System

For self-localization, the UAV is equipped with a *Septentrio AsterX2i OEM* commercial navigation system. Conceptually, this device consists of three different modules, which

are presented in the following subsections.

A GNSS receiver

Global navigation satellite systems (GNSS) are used to determine the vehicle's exact position. Internally, all parts of the GPS infrastructure (control segment, space segment and receivers) use the Earth-Centered Earth-Fixed (ECEF) coordinate system. For reasons that are largely founded in history and usability, values are usually converted to Longitude, Latitude and Altitude before being output to the user. Section 4.1.1 will further elucidate on different coordinate reference systems employed in GNSS as well as this thesis. For now, the term position shall be used without further qualification.

Generally, off-the-shelf GNSS receivers solve positions based on reception of the so-called code-phase of the coarse acquisition (C/A) code emitted by space vehicles on the L1 frequency band. In case of GPS, this signal is emitted on a single frequency ($1575.4MHz$), while GLONASS satellites use a frequency of $1602MHz + (562.5KHz * c)$, where c is the satellite's channel number. GPS satellites generate a stream of pseudo-random numbers (PRN), where the sequence of numbers depends only on the satellite's id (PRN code) and the time. Using the same algorithm, a receiver can generate the same stream of numbers and compare it to the stream received from that satellite. To determine the so-called *pseudorange* between a specific satellite and itself, the receiver simply delays the generation of its own copy of pseudo-random numbers until both streams are synchronized. The resulting delay is the time the signal traveled through space, and, accounting for the speed of light, yields the pseudo-distance between satellites and receiver. Given sufficient pseudoranges from different satellites, the receiver's position and clock can be computed.

Because the C/A code transmits data at a rate of 1.023 million bits/s, each bit amounts to a spatial span of roughly

$$\frac{299,792,458\ m/s}{1.023\ MBits/s} = 293.05\ m \tag{3.1}$$

GNSS receivers commonly achieve an alignment between both streams that is precise to about 1-2% of a single bit width (corresponding to 3-6 meters). This explains the upper bound to the precision with which positions are solved.

With the receiver usually being located at the surface of the earth, the signal has to pass through earth's athmosphere, where it is subjected to noise, dampening and other influences: in the upper layers, the signal is slowed down based on the conductivity of the ionosphere. This metric is named the ionosphere's total electron content (TEC) and it's value is both very dynamic and strongly depends on solar activity. Later, the signal is refracted in the troposphere, which is composed of a dry and a wet portion, with the latter being rather difficult to model (van der Hoeven et al., 2002). Close to the ground, a large part of the signal can also be reflected on local geometry before reaching the

receiver's antenna. The resulting effect is called multipath reception and can cause the receiver to measure imprecise pseudoranges (decreasing positioning accuracy), or fail altogether.

As a result, the positioning accuracy of single-frequency GNSS receivers is rarely better than 5m SEP_{50} in practice, even when using space-based augmentation services (SBAS) like WAAS or EGNOS, which are available only regionally. Clearly, these accuracies are insufficient for the purposes of this project. For precise mapping of outdoor environments, positioning errors must be limited to a range of few centimeters, requiring an error-reduction of two orders of magnitude.

Since the beginning of the 1990s, GNSS positioning with centimeter-accuracy (even millimeter-accuracy in favorable conditions) has become possible. This is an extremely remarkable technological accomplishment, especially considering that GPS was never designed to provide this accuracy in the first place. In summary, the main approaches to reducing positioning-errors are as follows:

First, antenna properties like gain, multipath rejection and phase center stability are critical to precise signal processing in the receiver: for precise GNSS surveys, receivers (especially reference stations) are often connected to choke-ring antennas, which were originally invented at NASA's JPL. These are built on a ground plane that is optimized to reject multipath reception by reducing antenna gain for directions below the horizon, which also improves tracking of low-elevation satellites. Although the geometry of the groundplane has been optimized since, groove depth and -distance must be sized relative to the signal's frequency, resulting in a large metal structure that can weigh up to 10kg (see figure 3.8(a)). A widely used compromise are so-called pinwheel antennas (shown in figure 3.8(b)); this type is much cheaper, smaller and features RF characteristics close to its choke-ring counterpart. Pinwheel-antennas typically weigh from little more than 100 grams (OEM versions without radome and mounting brackets) to 500 grams. Combining size and weight of either design with the additional requirement that GNSS antennas must be mounted with an unobstructed view of the sky complicates the mechanical sensor-arrangement and considerably reduces flight-times on a lightweight UAV with restricted payload capabilities. Thus, a helix-antenna was used: in shape, these antennas closely resemble those used on GNSS satellites, but are available in much smaller sizes. While the Maxtena M1227HCT-A (shown in figure 3.8(c)) does not reach the performance of aforementioned antennas in any discipline, it proved capable of reliable GNSS reception - and weighs only 17 grams (see table 3.1).

Survey-grade GNSS receivers receive satellite-signals on L1 as well as on L2 frequencies (1227.6 MHz). The major benefit of such dual-frequency receivers is that they receive two signals on multiple frequencies from the same satellite. Because the impact of the abovementioned ionospheric delay on a signal depends on its frequency, the delay can be measured indirectly, and thus, be removed.

(a) (b) (c)

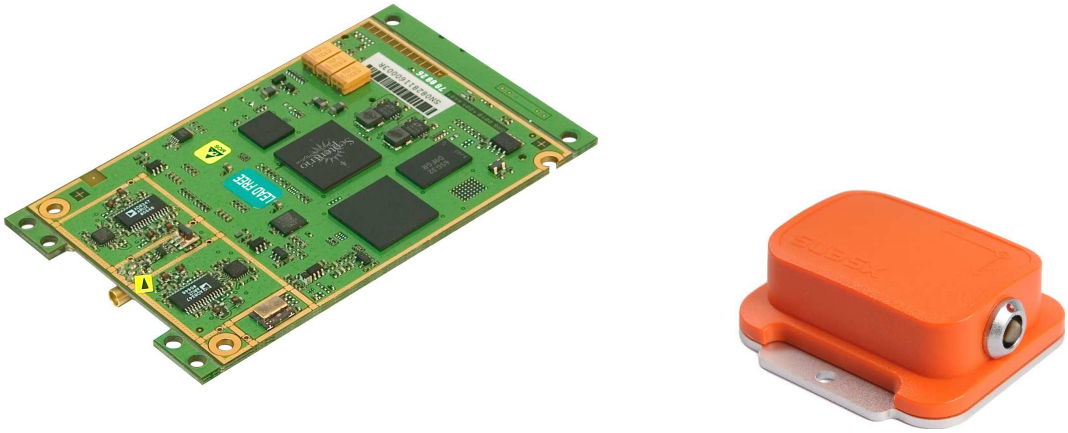
Figure 3.7: GNSS reference station components: (a): NovAtel pinwheel antenna, (b): Septentrio AsteRx2e-HDC GNSS receiver (c): MOXA COM server. Images ©: (a) 2014 NovAtel Inc., (b) 2011 Septentrio nv, (c) 2012 MOXA Inc.

Survey-grade GNSS requires a reference station. Conceptually, this is another dual-frequency GNSS receiver that operates at a precisely known position. After receiving the constellation's almanac and the satellites' ephemerides, it computes the pseudoranges to all the received satellites. Since the receiver's position is known, errors appearing in the pseudorange measurements can be detected and transmitted to the nearby GNSS rover. After subtracting the errors that appeared at the reference station, the rovers position is solved much more precisely - relative to the reference station. This transmission of differential corrections can either be done in real-time (named *Real-Time Kinematic GNSS*) or in post-processing.

By far the largest reduction of positioning-errors stems from the creative exploitation of GNSS signal's properties: instead of just relying on the code-phase alignment to the signal, survey-grade GNSS also performs carrier-phase measurements. The frequency of the L1 carrier-phase is 1575.4MHz which equates to a wavelength of 19 cm, so a phase-alignment with an error of 1% of the wavelength yields a precision of better than 2 millimeters. Unfortunately, while aligning to the carrier-phase is comparably simple, the receiver has no information about *which* phase it has aligned to. Synchronizing to the code-phase was one of the design goals of GPS and is aided by streams of pseudo-random numbers that, for this reason, do not contain large windows of repeating numbers. In contrast to this, there are no measurable differences between consecutive phases of the carrier-signal, leaving the receiver in an ambiguous state. Because the solution is now off by a full multiple of the wavelength, this problem is named the integer ambiguity. Solving this problem (i.e., "fixing the ambiguity") requires very sophisticated signal analysis, which is the primary reason why it took decades from the availability of GNSS signals to the availability of carrier-phase-based GNSS. Frei and Beutler (1990) present one of the first working approaches to RTK-GNSS, while Kim and Langley (2000) give a good overview of the years that followed. Nowadays, fixing the ambiguity takes only seconds in favorable conditions and can even be done while the receiver is moving ("On The Fly" ambiguity fixing).

(a) (b) (c)

Figure 3.8: Different GNSS antenna designs: (a): Leica AR20 choke-ring antenna (⌀32cm * 16cm height) with the choke rings in silver/gold and the antenna element in the center. (b): OEM version of NovAtel pinwheel antenna (⌀13cm * 3cm height, without radome). (c): Maxtena helix antenna (⌀3cm * 5cm height). Images ©: (a) 2014 Leica Geosystems AG, (b) 2014 NovAtel Inc., (c) 2011 Maxtena Inc.



(a) (b)

Figure 3.9: (a): OEM version of Septentrio's AsterX2i inertial navigation system. (b): XSens MTI MEMS-grade inertial measurement unit. Images ©: (a) 2011 Septentrio nv, (b) 2011 Xsens Technologies B.V.

The GNSS receiver used in the *AsterX2i* supports RTK-GNSS by receiving signals from satellites of GPS and GLONASS constellations on L1 and L2 frequencies, solving position, velocity and time at a rate of 10Hz.

A *Septentrio AsteRx2e HDC* dual-frequency GPS/GLONASS receiver (shown in figure 3.7(b)) is used as a reference station. Its antenna is mounted to a mast that is located the roof of the highest building on campus, guaranteeing a permanently free view of the sky.

B Inertial Measurement Unit

Inertial measurement units (IMUs) contain two kinds of sensors: accelerometers and gyroscopes. Strictly speaking, most IMUs also contain thermometers; since these are used only to correct the data sensed by the former, temperature-sensitive devices, they will not be described in further detail.

Accelerometers measure accelerations, usually on a single axis. Internally, they use a proof mass attached to a lever arm (that behaves like a spring), which is displaced according to the accelerations experienced by the sensor. This architecture forces the designer to compromise between sensitivity and input range by selecting suitable combinations of proof mass weight and spring constant. This displacement can be detected indirectly by measuring the capacitance between structures attached to the lever arm and other, fixed structures in close proximity. Accelerometers integrating this architecture in a microelectromechanical system (MEMS) are very small and comparably cheap, but produce comparably noisy values. Similarly, servo-accelerometers also contain hardware to sense the displacement of a ferromagnetic proof mass. However, this output is not used to measure accelerations, but to power a coil until the mass is moved back into its neutral position. The required current is proportional to the acceleration experienced by the mass, leading to a less noisy signal that will exhibit less drift in the integrated position. Combining three sensors that are oriented perpendicularly to each other into a 3-axis accelerometer, accelerations can be sensed as a vector quantity. The first integration of this vector over time yields a 3D-velocity, the second integration (velocity over time) returns 3D-displacement. Assuming that the accelerometer's bandwidth and sample rate satisfy those imposed by the Nyquist sampling criterion for the body's motion, the current position of the body can be reconstructed - as long as the body is not rotated, only translated. As with every sensor, the measurements of acceleration contain noise; double-integrating this noise into the object's position causes drift that - within few seconds - renders the derived position too imprecise for mapping.

Because the UAV's configuration space is six-dimensional, the IMU will be subjected to rotations, which can be measured using gyroscopes. Single gyroscopes can be combined into a 3D-gyroscope analogous to the accelerometers described above. There are two different ways of measuring rotations that are in common use today: the first approach detects rotation by measuring the coriolis force applied to a micromechanical mass, similar in notion to MEMS accelerometers. Accordingly, the setup exhibits similar characteristics in terms of price and accuracy. Far more precise (and expensive) than MEMS rotation sensors are solid-state Sagnac interferometers: these are based on coherent light being split and traveling in opposite directions through a ring-like path, made of either fiber optic cables (the resulting sensor consequently being called a fiber-optic gyro) or a mirror-setup (named ring-laser gyroscope). In both setups, both beams exit the ring at the same position they were injected and are allowed to interfere with one another. When the ring's platform is rotated along the normal of the plane representing

the light's path, the relative phases of the beams will change, creating constructive or destructive interference proportional to the angular velocity. Using a photo-diode, the intensity of the resulting beam can be measured, delivering exceptionally precise, low-noise measurements of angular velocity.

By combining accelerometers and gyroscopes into an inertial measurement unit and fusing data of both sensor-triads, the unit's position can be tracked even when it is subjected to rotations. The characteristics of the resulting data make IMUs ideal for short-term, high-frequency motion tracking.

An *XSens MTI* MEMS IMU (figure 3.9(b)) is used to collect inertial data on the UAV. This specific IMU also contains an internal Kalman filter ("XSens Kalman Filter", XKF-3) that can be used to compute a 3D orientation of the IMU in space. To compensate for orientational drift, this filter also uses a built-in 3D magnetometer. However, nearby electrical currents of up to 40A to each of the UAVs motors do not provide a basis for precise analysis of the earth's magnetic field. Since XKF-3 further requires that accelerations due to translation of the IMU are zero on average, the filter turns out to be unsuitable for application in a navigation system. For this reason, XKF-3 is disabled and the IMU is setup to output temperature-corrected (but otherwise raw) inertial data from accelerometers and gyroscopes.

The IMUs data sheet (Xsens Technologies B.V. (2009)) specifies a temperature range of -20 to $+55°C$. Its gyroscopes are limited to an input range of $\pm 300°/s$ and a bandwidth of $40Hz$, whereas the accelerometers have an input range of $\pm 50m/s^2$ and a bandwidth of $30Hz$. The Nyquist theorem dictates that signals (representing the platform's motion) containing frequencies more than half these specified bandwidths cannot be captured and reconstructed without aliasing effects causing loss of information. Thus, undersampling this signal can result in inaccurate solutions in later stages of the processing pipeline. On the other hand, although measurement rates of up to 512Hz are supported by sensor's on-board circuitry, sampling with higher frequencies than the specified bandwidth will not yield additional information. For this reason, the INS reads angular velocities and accelerations at a rate of 50Hz.

C GNSS/Inertial Navigation System

The are three different approaches to fusing GNSS and IMU sensor data:

1. loose coupling: in this approach, GNSS data processing uses a separate Kalman filter. Once GNSS-position, -velocity and -attitude solutions are determined, these values are fed into a second Kalman filter, designed to fuse GNSS and IMU data. This approach is the most simple to implement, as it conceptually keeps both sensors separate as much as possible. It also is the most robust, because a strongly drifting IMU can always be corrected and does not have a detrimental impact on the GNSS solution.

2. tight coupling means that raw GNSS measurements are fed into the GNSS/IMU integration filter. This increases the filter's complexity considerably, but allows for continued GNSS tracking when fewer than four satellites are available. On the other hand, this requires the inertial data to be of higher quality than in loose coupling applications, as low-quality IMU data can impact precision and reliability of GNSS tracking.

3. deep coupling describes the tightest integration of both sensors: the IMU-based velocity is used in the GNSS tracking loop, enabling the system to reacquire signals much faster after a signal has been lost. While this approach strongly depends on inertial data quality, it has proven very useful especially in environments where GNSS signals are often lost due to obstructions.

The Septentrio AsterX2i fuses data from GNSS and IMU sensors using a loosely coupling filter. Although the source of the code used in the INS itself is not public, it's control loop will likely run every $20ms$, reading the PVT solution originating from the GNSS receiver at it's maximum rate of 10 Hz and the inertial measurements at 50 Hz. It is important to mention that data received from both sensors at the same time does not guarantee that the underlying measurements were executed synchronously. While GNSS measurements are inherently well timestamped, this is not true for data digitized by the IMU, since it does not necessarily contain a precise clock. Unfortunately, documentation of sensor time-synchronization procedures are proprietary to the manufacturer.

Next, the data must be converted into a common coordinate reference system (which usually matches that of the IMU). This process is straightforward, because the virtual lever arm between IMU reference frame and antenna reference point is specified as part of the configuration procedure after INS startup.

In a following step, the INS filters will derive synchronous position and attitude from both datasets. Because of the low update rate of the GNSS receiver (instead of 10Hz, 20Hz is a common update rate nowadays), the filter is forced to rely on inertial data for 4 consecutive iterations after having fused GNSS and IMU data. This means that for the next 80 ms, accelerations are integrated twice to compute the distance traveled since the last fused pose, whereas angular velocities are integrated once to derive the vehicle's updated orientation.

In an attempt to reduce the drift introduced in these intervals, the system also features a so-called zero-velocity update (ZUPT). Using GNSS-derived velocity, it is able to detect that the antenna is static: in this case, the knowledge about the system's state allows the filter to limit the error growth of the fused solution. Furthermore, one of the 80 pins in the INS connector can also be configured to serve as an input for external zero-velocity update indications. While this is very useful in many applications (e.g. ground-based robots are usually static when their drives are not powered), it obviously is of no help in airborne applications.

Table 3.2: Configurable intervals for position and orientation packets (SBF: INTPVAAGEOD) and the resulting sequence of (F)used and (I)ntegrated poses.

Interval	**Frequency**	t_0	t_1	t_2	t_3	t_4	t_5	t_6	...
20 ms	50 Hz	F	I	I	I	I	F	I	...
40 ms	25 Hz	F	I	I	I	I	F	I	...
50 ms	20 Hz	F	I	F	I	F	I	F	...
100 ms	10 Hz	F	F	F	F	F	F	F	...

Using data of inaccurate poses for sensor fusion with the scanned range data is somewhat detrimental to mapping accuracy, as the spatial drift in the solved poses directly translates to offsets in the point cloud. Much worse, when such poses are used for flight-control, the spikes in acceleration (that result when a drifted position is being corrected using GNSS measurements) can have disastrous effects when used in a flight controller that contains a derivative component to correct for errors in position, or similarly, computes its output values based on accelerations. For a more detailed description of the high-level flight controller used for motion control, please see section 4.4.

The INS can be configured to output a wide variety of measurements between intervals of varying length. Legal intervals are 10, 20, 40, 50, 100, 200 and 500 milliseconds, as well as longer intervals up to one hour. Given that sensor data is only retrieved every 20ms, the shortest interval of 10ms is only useful for timing or debugging information. Longer intervals are helpful when requesting configuration and system state information (e.g. GNSS positioning mode or the age of the supplied differential corrections), while shorter intervals are used to retrieve position and orientation packets. When requesting these packets at less than the maximum rate of 50Hz, the phase of the fused/integrated poses is adjusted within the constraints of the filter loop interval of 50Hz: table 3.2 shows how the output is always configured to include as many fused poses as possible.

After acquiring a GNSS PVT solution on startup, the INS performs a calibration. This procedure lasts about 30 seconds, during which it is of paramount importance that the UAV does not move. Most likely, the filter estimates the IMU's biases at this time.

Following this stage, the INS is capable of determining its own position and - assuming it is static - roll and pitch angles. However, the heading of the platform remains unsolved. Sagnac interferometer-based gyroscopes usually drift less than 15 degrees per hour (earth's angular velocity), so they are able to measure earth's rotation and, consequently, determine their own heading. This process is named *static alignment* and is often the default alignment method when fiber-optic gyros are used. Even though static alignment can simplify a navigation system's initialization, it requires its vehicle to remain completely still for a period of up to 30 minutes, sometimes necessitating that the whole platform be rotated by e.g. 90 degrees multiple times in the process. In

some scenarios, platforms simply cannot be held static for the required periods of time, making static alignment impossible. These include initialization on maritime vessels as well as re-initialization during flight.

To solve the system's heading after startup using MEMS-based IMUs, trajectories of IMU and GNSS have to be aligned while the vehicle moves. This approach is known as *kinematic alignment* and is conceptually supported by all IMUs. Using the Septentrio Asterx2i INS, kinematic alignment happens during the first seconds of flight, and works best by moving forward in a straight line. After the heading has been determined roughly (this state-change is indicated in a packet that is logged frequently), covariances of the fused solution are minimized by flying curves and figure-eights, accelerating and stopping. The covariances of the solved poses are also logged frequently, so that poses of inferior accuracy are not used to fuse range data and consequently "pollute" the point cloud. The duration of this phase depends on the desired covariance thresholds and motions experienced by the INS, but usually takes less than 30 seconds. Given sufficient satellite reception, the system's position is solved to a precision of 5 centimeters in RTK fixed mode, while roll and pitch angles exhibit maximum errors of about 0.5°. The precision of the heading angle depends on the amount of motion the vehicle experienced in the preceding seconds, but usually converges to a maximum error of less than 1.0°.

The INS also offers two event-marker pins: these are electrical connections that can be set low or high (LVTTL levels, 5V tolerant) using external circuitry. Depending on their configuration, the pins will cause an event-packet to be generated when either a rising or a falling edge is detected. The event-packet contains a field that indicates on which pin the state-change was detected. Most importantly though, the packet also contains a timestamp from the same GNSS-synchronized clock that is used to timestamp the position and orientation packets. Here, the manufacturer guarantees a maximum timing error of 20ns.

Altogether, the GNSS-receiver, inertial measurement unit, navigation system, -antenna, breakout board, cables and case weigh just 184 grams.

3.1.4 LIDAR sensors

Initially, a single Hokuyo UTM-30lx laser range finder was mounted to the front arm with its field of view aligned vertically (as depicted in figure 3.2(b)), and connected to the aforementioned on-board computer. Its laser scanned a front-facing planar field of view of 270°, resulting in a 90°blind spot in its back that aligned well with the available field of view at that mounting position. To fully take advantage of this setup, a flight controller was created to reach waypoints by pitching and rolling towards them, while constantly yawing at the same time. This way, the scanner would have been capable of scanning the ground below as well as obstacles around the UAV. While this approach worked satisfactorily in simulation, the actual INS was not capable of

Figure 3.10: Hokuyo UTM-30lx laser range scanner. Image © 2008 Hokuyo Automatic CO., Ltd.

delivering precise heading information by fusing the sensor trajectories of GNSS receiver and IMU, because its proprietary filters were created under the assumption that the vehicle would move mostly forward. Also, with LIDAR scanner and IMU mounted directly to the boom, both were subjected to strong vibrations originating in the UAV's motors, possibly reducing the scanner's life expectancy and adding noise to the data recorded by the IMU.

For the second attempt, the laser scanner remained mounted to the vehicle's forward arm, but with its front side facing downwards and the blind spot directed upwards (as shown in figure 3.3(b)). The flight controller was adapted to fly forward, while yawing and rolling towards given waypoints. This resulted in precise attitude solutions from the INS, but made detection of obstacles ahead of the vehicle difficult. The IMU was moved to the vehicle's center and installed using vibration dampeners. While this reduced the noise in its readings considerably, it decoupled the laser scanner from the IMU, resulting in a detrimental effect of unknown magnitude on the registration accuracy. Because the scanner was mounted about 12cm towards the front of the UAV's geometric center, it also shifted the vehicle's center of gravity towards the front motor. Although the resulting bias would have been compensated by system's low-level flight controller automatically, it would have caused more stress for the respective motors and propellers as well as an unbalanced heat production on the motor-controllers. Instead, the other components (especially the battery) were shifted slightly in the opposite direction, restoring the vehicle's balance.

After the system was capable of creating point clouds in real-time, the focus shifted

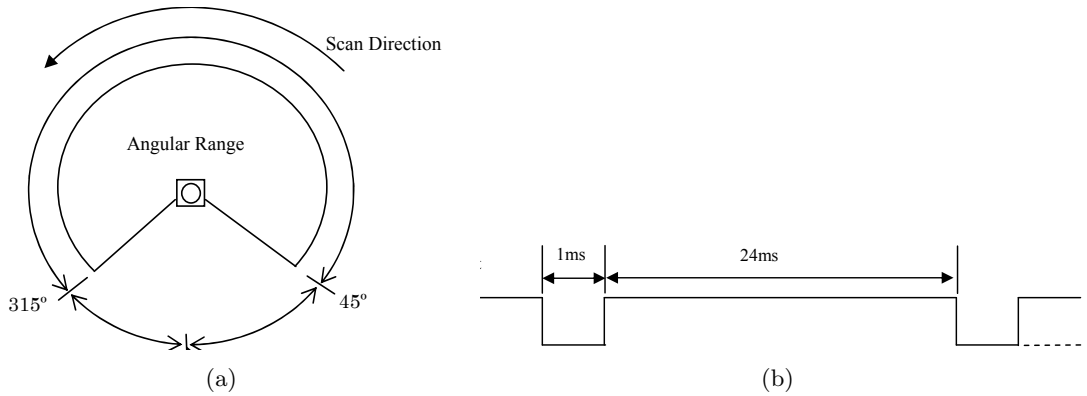


Figure 3.11: (a): The field of view of the Hokuyo UTM-30lx laser scanner, seen from the top. The 270°horizontal FoV is sampled at 40Hz, with the ray rotating counterclockwise. One rotation takes 25 ms, of which 6,25 ms are used for data processing and output. (b) shows the temporal behavior of the SYNC signal. A falling edge indicates that the ray is at the 0°position in the rear, which can be exploited for time synchronization. Both images reprinted from product's specification sheet, © 2008 Hokuyo Automatic CO., Ltd.

towards autonomous exploration: the high-level flight controller was programmed to roughly orient the UAV towards the general direction of the next waypoint, then adjust yaw and roll during the following approach. This behavior resulted in frequent changes in direction, meaning that most of the geometry around the UAV was scanned by the time the vehicle could have collided with it. However, this arrangement only worked out most of the time and was thus deemed too unreliable to be entrusted with a task as essential as collision avoidance. So the final setup (figure 3.4) was created in order to accommodate a second, forward-looking laser scanner for better obstacle detection. Both scanners are constantly operating during flight, with all recorded points being transformed into the same point cloud. This not only allows for more reliable collision avoidance, but at the same time accelerates the scanning procedure substantially. The physical arrangement was modified so that the comparably lightweight on-board computer and INS were now installed above the UAV's frame. At the same time, the heavy combination of both laser scanners, IMU and battery (which together amount to more than 40% of the platform's total weight) were coupled into a sensor-module: the base of this module is a carbon-fiber sandwich board (produced by placing a foam-core in between two thin layers of carbon fiber pre-preg material), offering an extremely lightweight *and* extremely stiff surface to mount the sensors. Due to the warp resistance of the board and the close proximity between IMU and laser scanners, the orientational sensor-alignment is far more precise compared to the previous setup. Even though the high weight of the sensor module makes it inherently more tolerant against vibrations induced from the outside, it was installed below the UAV's center using four vibration dampening rubber fasteners.

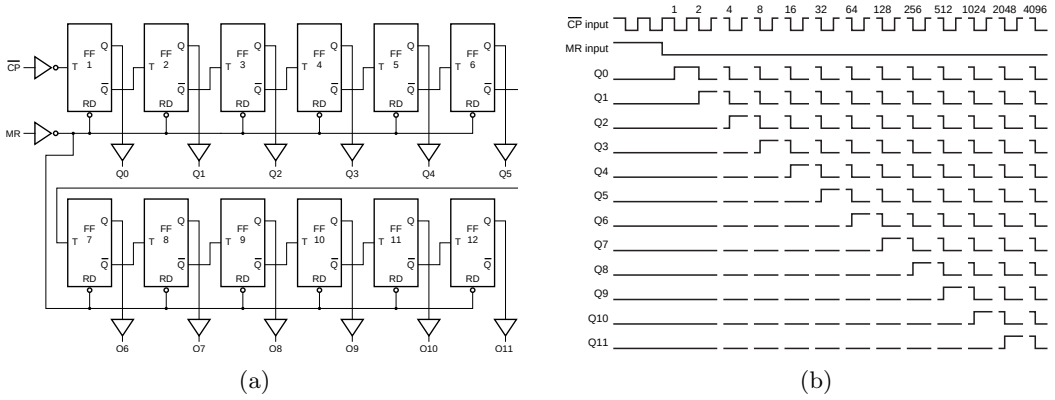
Each scanner delivers distance values of 1080 rays with 0.25°angular resolution at a rate of 40Hz and a range of up to 30 meters. According to Demski et al. (2013),

the precision of measured distances depends on reflectivity and color of the sampled material but generally shows relative errors of less than 1%. Interestingly, the scanner additionally suffers from drift within the first 50 minutes of operation: after startup, returned distances are up to 10mm shorter than ground truth.

Although every scan of 1080 rays contains a timestamp, it's value is generated using a local clock that is initialized to zero when the laser scanner is powered up. The manufacturer describes methods of synchronizing clocks, relying on the assumption that data is transferred on the USB with relatively low jitter. But because communication is implemented using USB bulk transfer modes, no timing guarantees are given by the USB specification. In effect, synchronization is precise only to a few milliseconds and depends on bus utilization. A synchronization error of e.g. $5ms$, a scanned distance of $25m$ and an angular vehicle-velocity of $100°/s$ leads to an angular error of $0.5°$, and, by solving the length of the third leg of the resulting isosceles triangle, leads to a positional error of about $0.22m$ for that point. These numbers also serve as both an example for the imprecisions induced by the INS error specifications of 1.0 and 0.5 degrees for yaw, pitch and roll as well as the reason why using LIDAR sensors with extended range is not always helpful.

To provide a more precise time synchronization, the scanner also features a SYNC-signal on a dedicated pin. It is pulled low for 1 millisecond whenever the laser traverses the scanner's rear position (shown in figure 3.11). This signal is connected from each of the scanners to one of the event-in pins of the GNSS receiver. After configuring both pins to emit a timestamp on falling edges, the beginning and end of each scan is timestamped with nanosecond-precision. Because the timestamp in event-, position- and orientation-packets is specified using the same timescale, almost perfect temporal data-association can be achieved. In effect, every single ray can be temporally associated to the received poses, allowing precise interpolation of position and orientation to the moment a range was scanned. In our experience, the advantage of extremely precise timing measurements for highly dynamic mobile platforms (even in temporary absence of satellite coverage) is often overlooked.

In total, both installed scanners cause 80 falling edges and thus, 80 event-packets to be generated every second. According to the manual, the INS is capable of processing up to four events on all pins within each 50ms interval, as long as the minimum time between any two events exceeds 5ms. Even though the described setup adheres to these limitations, the generation of this many packets does affect the CPU load of the INS, which is a critical parameter and must be kept below a vendor-specified threshold. When further INS output is enabled (e.g. for diagnostic purposes), higher CPU load can be a reason for the INS being unable to reach precise positioning modes or successfully complete kinematic alignment. To alleviate this problem, a 74HC4040 binary counter was installed between each laser scanner and the respective event-in pin. By attaching the SYNC signal to CLOCK and the event-in pin to one of the counter's parallel output pins, the SYNC signal's frequency can be divided by 2 (pin Q1) to 4096 (pin Q12).



(a) (b)

Figure 3.12: (a): Logic diagram of the custom circuit included to lower the inertial navigation system's CPU load through frequency-division of the laser scanner's SYNC-signal (connected to $\overline{CP}$). Subfigure (b) shows selectable frequency divisors and corresponding timing. Reprinted from Philips 74HC4040 datasheet, © 2005 Philips Electronics N.V.

This circuitry enables registration of the scanner's rotational phase while conserving the limited computational resources of the INS' embedded processor. Please see figure 3.12 for a detailed schematic of the setup.

3.1.5 Electromagnetic interference

Flight controller, laser scanner and especially the on-board computer emit electromagnetic interference on the L1 (1575.42 MHz) and especially L2 (1227.6 MHz) bands, which is picked up by the GNSS antenna, often causing the GNSS signal-to-noise ratios to drop below acceptable thresholds. This forces the INS to rely on double-integrated IMU sensor readings for the duration of GNSS dropouts, introducing considerable drift into the solutions of position and attitude.

Determining electromagnetic interference as the reason for unreliable GNSS reception and the AtomBoard as the major source of this interference took up a considerable amount of time, as many components are required for a GNSS rover to reliably solve a position in RTK-fixed mode, and failure in a single component can be the cause for floating ambiguities to remain unsolved:

- the GNSS reference station is required to be correctly configured to emit differential corrections. The station's own position, the data format of corrections (RTCM2, RTCM3, CMR, CMR+ etc.), the emitted messages (different for every data format) and their intervals depend on the feature-set of the rover as well as the datarate and latency of the communication channel between reference station and rover.
- as explained in section 3.1.3, compromises had to be made regarding the GNSS antenna. Using a helix-antenna that cannot support accuracies as high as its

survey-grade counterpart increased the chances that solving the carrier-phase ambiguities on the rover fails.

- a sufficiently large intersecting set of satellites received by both reference station and rover. Limitations in the rover's firmware reduce this set even further, as fixing carrier-phase ambiguities is only supported for GPS signals, while signals from the GLONASS constellation are only drawn into a pre-established fixed mode solution for increased robustness. Of course, geometry obstructing a clear view of the sky ("local horizon" in surveyor parlance) must also be considered and can make GNSS mission planning mandatory.

Because the department of computer science currently has no access to the equipment required for better diagnosis of the interferences, the problem was rectified by wrapping all sources of interference in aluminum foil. Wires to and from the on-board computer were also found to act as antennas emitting further interferences, which was fixed by running the affected cables through ferrite rings. As a result, only small parts of the interferences remain, as large parts of the laser scanner's surface had to remained unshielded for obvious reasons. Moving the GNSS antenna upwards and away from the scanner has further reduced the effect of this problem so much that GNSS reception has become reliable. On the downside, the thermal insulation caused by the added shielding has imposed the need to monitor temperatures of on-board computer and motor controllers during flight.

3.2 Software

During development, a custom software stack was created, consisting of three main packages: a simulator, a base station and a rover program to run on the UAV. Because all algorithms must run in real-time and some parts (like the high-level flight controller) are run on rather constrained embedded hardware, all software is implemented using compiled languages, specifically C++ for software implemented on CPUs and CUDA for algorithms implemented on NVIDIA GPUs.

The software architecture was conceived and first implemented towards the end of 2009. Today, the Robot Operating System ("ROS") is commonly used to implement complex software architectures in robotics, as it offers a robust framework for message passing, software development, maintenance and, most importantly, reusability of other author's code. However, ROS saw its first release on January the 22nd of 2010 and became more widely known by the end of that year. It is built on the idea of loosely-coupled *nodes* running in separate processes, exchanging information in well-defined formats using so-called *topics*, which use TCP/IP connections for data transport. The resulting advantage is that nodes can be deployed in distributed fashion, as long as the hosting machines are connected in a network. On the downside, when data is sent from one node to another, it has to be serialized, sent over the network (or the loopback device

when both nodes reside on the same host) and unserialized. When small records (e.g. position and orientation) are exchanged at low rates, this overhead remains negligible even on most embedded hardware. But for data of larger size, e.g. laser scans that are emitted at higher frequencies, this marshaling incurs considerable cost compared to a simple in-process exchange of pointers. Other techniques like buffer-sharing, which is intensely used in the base station for GPU computing and visualization, cannot be facilitated by ROS with reasonable effort. The aforementioned overhead and other design problems have lead to the inception of other robotics-focused programming frameworks like MIRA (introduced in Einhorn et al. (2012)) as well as some attempts of solving some of these problems in ROS, e.g. by the use of *nodelets*.

However, because hardware *requirements* of this project were largely unknown during the earlier stages of development – and the hardware had already been applied for – it was decided to employ a custom solution that could be better adapted to work around bottlenecks in the given architecture. This solution is composed of three programs, which are introduced in the following sections.

3.2.1 Simulator

The simulator (see figure 3.14) was implemented first in order to test how different scanner setups and flight controller implementations affect the speed and quality of mapping. Technically, it employs Qt for the user interface and network modules, the Bullet library for physics simulation, OpenGL for visualization and Ogre3D for a 3D scene graph, material management and model loading.

A configuration dialog allows the user to quickly define environmental conditions such as wind, as well as various vehicle setups, including different batteries and unlimited numbers of arbitrarily oriented laser scanners and cameras.

A LIDAR sensors

The range readings of the simulated laser scanners are implemented using Ogre3D's RaySceneQuery class. Queries derived from this class quickly return the first entity whose axis-aligned bounding box (AABB) collides with a given ray. In a next step, the triangles of a colliding entity's mesh are iterated to find the colliding triangle closest to the ray's origin. In case no collision is detected (because the ray only traverses the entity's bound box, but not its mesh), the next colliding entity is determined. When using the lightweight listener interface provided by RaySceneQueryListener, further collisions with entities are only determined when no collisions with the mesh were detected. Using this technique and the entity's materials, even simulation of LIDAR sensors featuring multiple returns and waveform digitization can be realized.

The importance of efficient ray scene queries becomes obvious when simulating a vehicle

setup with two Hokuyo UTM-30lx laser scanners as shown in figure 3.4(b): in real-time, two scanners each sample their field of view with a frequency of 40Hz, with each sweep requiring a minimum of 1080 queries. In total, this results in at least

$$2\,scanners * 40\ \frac{scans}{s} * 270\ \frac{degrees}{scan} * 4\ \frac{ranges}{degree} = 86400\ \frac{ranges}{s}$$

If a ray only traverses the first colliding entity's AABB, but not it's mesh, further ray scene queries are required. As a first optimization, ray scene queries can be accelerated by reducing the spatial search space through octree-based scene managers. After a second laser scanner was added to the real platform, the LIDAR-simulation was updated to perform scene queries and collision detection between triangles and rays in threads specifically allocated for every instanced laser scanner. Aside from the vehicle itself, the scene's geometry is rarely updated, so sharing of scene geometry between threads is uncomplicated and does not require mutex locks. In practice, when simulating multiple scanners, the operating system's scheduler assigns each scanner's thread a separate CPU core even without manually specifying thread affinity, allowing the simulation speed to scale favorably with both clock speed and the number of available cores.

In order to simulate realistic range readings, every ray's orientation is offset on yaw, pitch and roll using a normal distribution using a Mersenne twister pseudo-random number generator (PRNG). The computed distance to the ray's first collision with a triangle is also overlaid using a normal distribution with the standard deviation depending on the computed range, similar to the specifications of the Hokuyo UTM-30lx laser scanner.

B Motion Control

For meaningful tests of motion control, both low-level and high-level controllers had to be implemented in the simulator. One of the preconditions to simulating realistic flight behavior and battery discharge is a mapping of current vs. thrust of the used motors. Such a mapping was determined for the UAV's motors (Robbe ROXXY 2827-35) by measuring and is publicly available at Mikrokopter (2013) and reprinted in table 3.3. This data was used to generate fitting function 3.2. Converted into a method in C++ code, motor-speeds are first mapped to thrust, then fed into the physics simulation engine.

$$f = (current^{\,0.3} + \frac{current}{135})/14.0 \tag{3.2}$$

The low-level controller present in the FlightCtrl board is a simple PID controller accepting setpoints for attitude angles (roll and pitch), angular velocity (yaw) and thrust. The individual gains were not directly copied from the UAV, but tuned until the simulated UAV behaved similarly to the real one in flight. Exactly like the real UAV,

Table 3.3: Current vs. thrust for the Robbe ROXXY 2827-35

Current (mA)	**Thrust (g)**	**Thrust (N)**
0	0	0
500	100	1.02
1000	172	1.75
1500	242	2.47
2000	309	3.15
2500	364	3.71
3000	412	4.20
3500	440	4.49
4000	495	5.05
4500	530	5.40
5000	566	5.77
5500	620	6.32
6000	640	6.52
7000	700	7.14
8000	770	7.85
9000	800	8.15
10000	860	8.77

the setpoints can be defined either using a manual control method (the remote control is replaced with a joystick in simulation) or by the high-level flight controller.

The high-level flight controller's inputs are the current time, vehicle position, orientation and the position of the next two waypoints. Using this data, it can compute the vehicle's velocity and is responsible for generating setpoints for the low-level flight controller appropriate to reach the first given waypoint.

When operating in outdoor environments, the UAV is constantly exposed to wind. To better understand the effects of wind on airborne platforms, a dataset recorded by Lange and Brümmer (2010) at a local weather research institute was analyzed. The weather mast is about 300m high and collects wind velocity and direction at heights of 10, 50, 110, 175, 250 and 280 meters above gound level. By design, simple mechanical cup anemometers suffer from a low measurement bandwidth, induced by their rotary inertia. To avoid these limitations, ultrasonic anemometers were employed, recording three-dimensional vectors of air-velocity at a rate of 10Hz. Although the given data only spans a single day, several conclusions were drawn from it's visualization (see figure 3.15):

- average wind velocity increases with height above ground.
- average wind velocity strongly depends on the time of day.
- wind direction is most constant at higher altitudes and much more variable at lower heights.

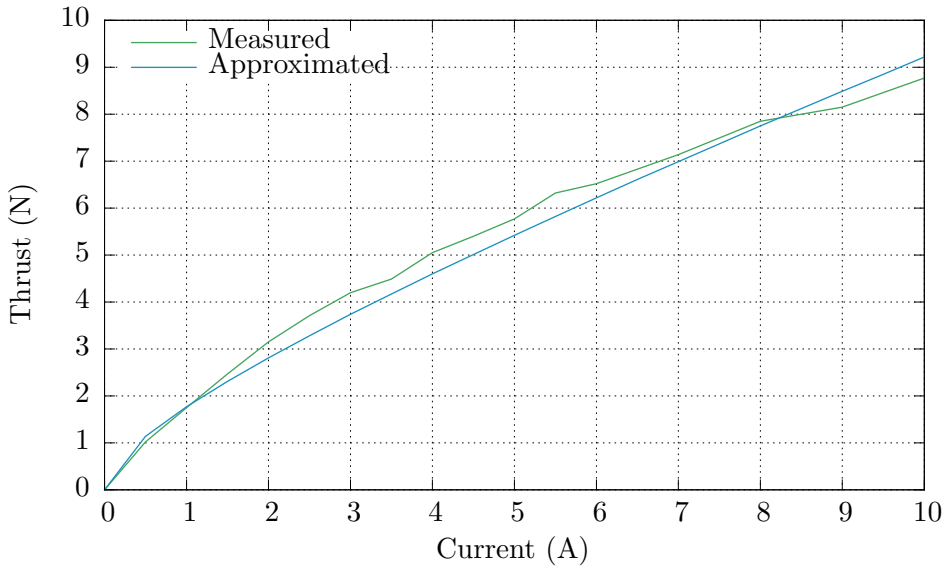


Figure 3.13: A graph showing the current vs. thrust on a Robbe ROXXY 2827-35 motor with 10 inch EPP propellers.

- wind direction is almost constant during most of the day, but can change greatly within an hour.

Given that the laser scanner's range limits the UAV's service ceiling to an altitude of about 30m, only wind data from the sensors at 10m height was integrated into simulation. As a result, enabling wind in the simulator's configuration exposes the high-level flight controller to realistic drift, gusts and turbulences, which brought about further optimizations to the controller's gains.

During this part of development, the availability of a simulator was extremely valuable, as it saved both time and physical damage to the UAV. Only simulation allows testing in predefined and repeatable conditions, which are very important for fine-tuning controller output. For debugging, the simulator's ability to run the simulation much slower than real-time has also proven to be very helpful.

3.2.2 Base Station

Implementation of the base station started shortly after that of the simulator. The program is meant to run on a computer in the field, which is monitored by an operator during all phases of flight. Because it runs on hardware offering far more computational resources than those available on-board the UAV, it hosts almost all of the algorithms presented in the following chapters. As a result, the base station is the most complex program, making use of the *Qt* library for its user interface, networking modules and serial port management, *OpenGL 4 core* for visualization, *NVIDIA CUDA* for GPU computing, *DevIL* for texture and *assimp* for 3D model loading as well as *SimpleDi-*

Figure 3.14: Screenshot of the simulation environment. Simulation speed can vary from a minimum of 0.01x to about 10x real-time, the UAV can be controlled manually via joystick as well as autonomously using a custom high-level flight-controller that approaches generated waypoints.

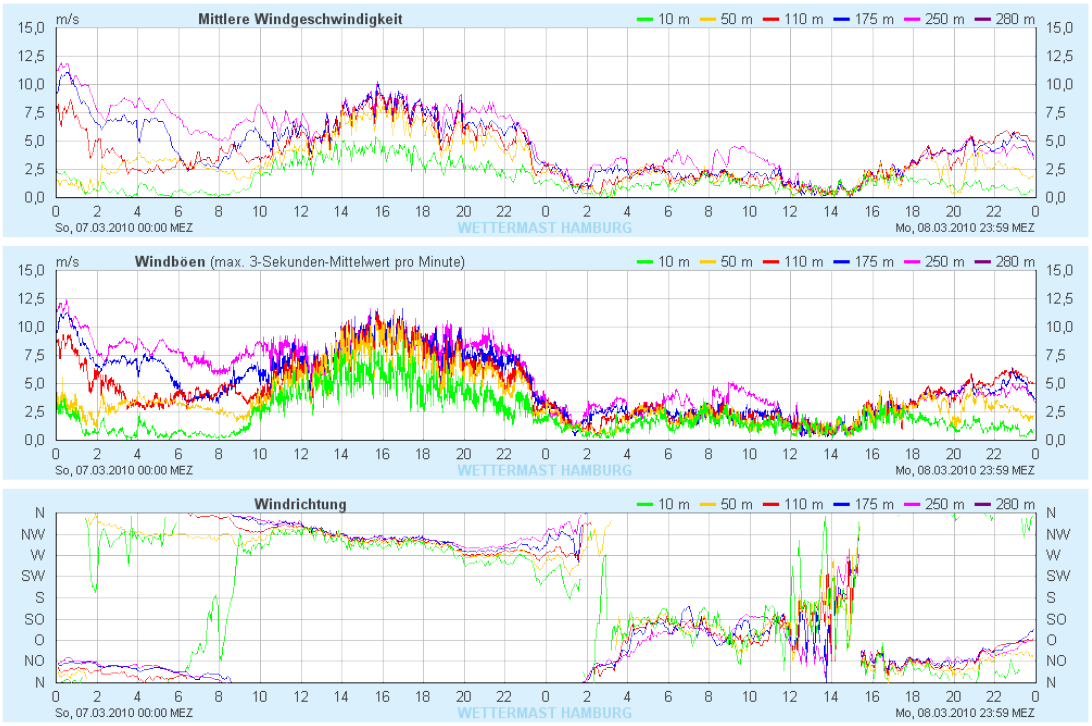


Figure 3.15: Wind data from Lange and Brümmer (2010) for more realistic simulation and tests of the flight controller.

rectMedia for audio output. The base station implements the same communication protocol as both the simulator and the rover programs, so it can connect to both, operating on simulated or real data (see figure 3.16).

Figure 3.17 shows a screenshot of the base station replaying a previously recorded flight. The central user-interface component shows the vehicle position and orientation, as well as the growing point cloud in a live 3D visualization. On the left, the currently orbiting satellites of GPS, Glonass and Galileo constellations are displayed in a list and a so-called sky plot. The right hand side informs the operator of the system's state, which consists of flight-time and battery voltage, state of the high-level motion controller, WiFi reception strength, GNSS reception quality, GNSS/inertial fusion status, age of

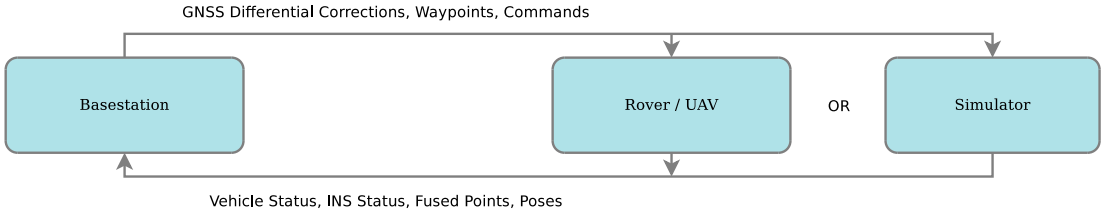


Figure 3.16: Communication between the main software modules. The base station connects to either the rover (UAV) or a simulator, permitting tests of the employed algorithms in simulation before they are tested on the UAV.

differential corrections and others. Just below the status fields, a list shows the currently queued waypoints and allows their manual addition and removal. The bottom row contains a window with log messages on the left, as well as controls for log-file replay on the right.

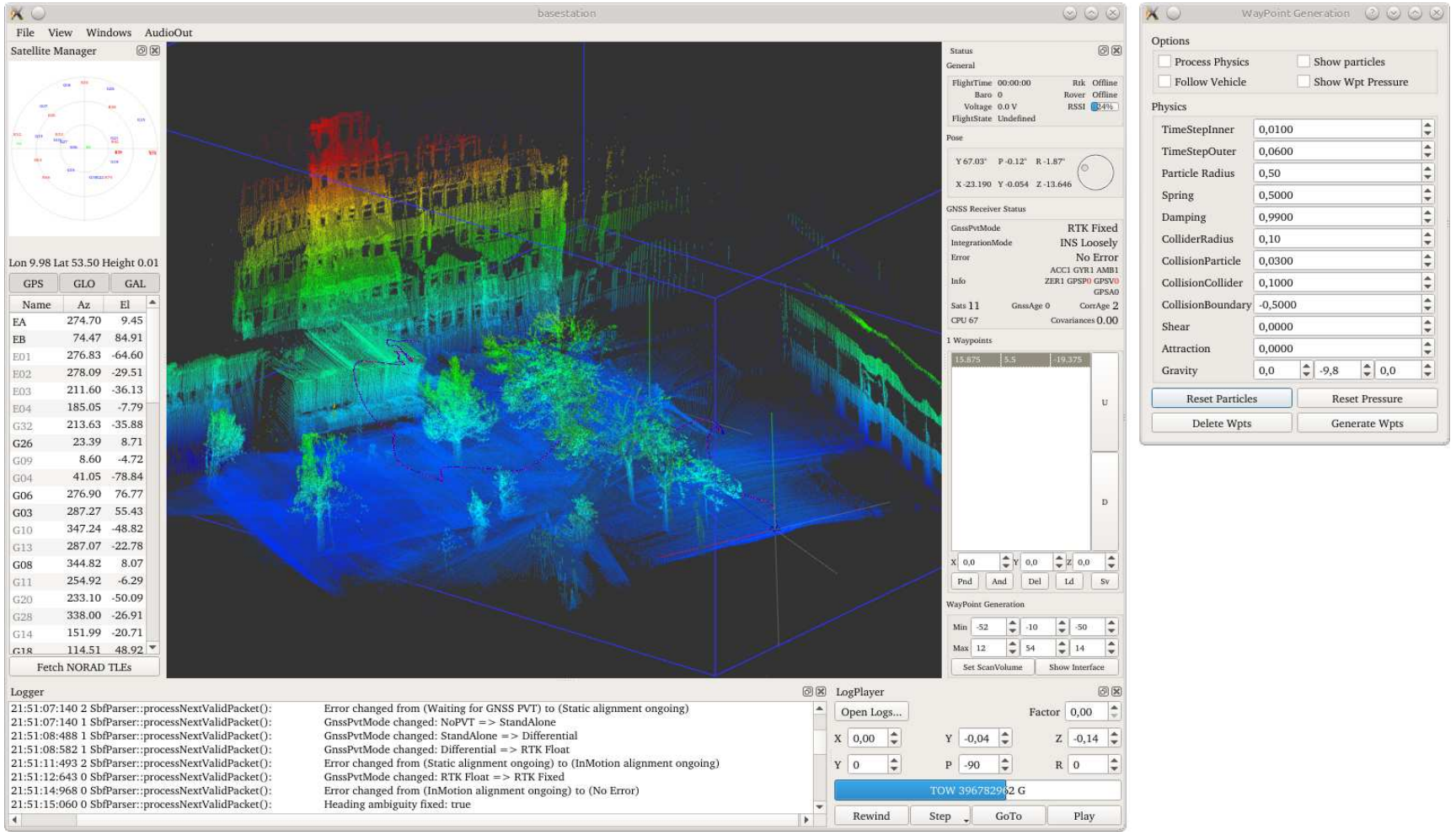


Figure 3.17: Screenshot of the base station program, running on a notebook computer in the field. The point cloud, streamed in real-time from the vehicle, is displayed in the center. Satellite positions are shown on the left, log messages in the bottom view. Status of vehicle, flight planner, and navigation system are shown on the right. Waypoint generation options are located in a separate window on the right.

In terms of functionality, the following modules are hosted in the base station:

As explained in detail in section 3.1.3, differential correction data has to be transmitted from the GNSS reference station to the rover for it to solve it's position with centimeter-level accuracy. One module is responsible for connecting to the GNSS reference station using a separate serial connection or via TCP over a 2G or 3G cellular data network. This connection will forward incoming data (from about 0.5 to 3 kb/s, depending on the number of GNSS constellations supported and thus, satellites observed) to the rover program using the protocol specified in table 3.5.

For reasons explained in more detail in the following chapter, two point clouds are managed on the ground: a dense cloud is used for visualization and later surface reconstruction, while a sparse version of the cloud is managed for path planning and obstacle avoidance. Triggered by a timer, incoming points are inserted into the dense cloud, which is then slightly downsampled to remove redundant points. Once this process completes, the remaining new points are copied into the sparse cloud, which is then downsampled further, building the data structure that backs all following flight planning algorithms.

Whenever the operator allows the vehicle to fly autonomously, but no waypoints are present, they must be computed: the next-best-view algorithm is started to find gaps in the point cloud. Honoring a safety margin around all obstacles in the point cloud, these gaps are then converted into reachable waypoints and ordered into the shortest possible path. The sequence of steps completes with the base station sending the ordered list of waypoints back to the rover. During flight, incoming points are constantly appended to both point clouds and used to check that the remaining path is still safely traversable. If not, the rover is commanded to clear the list of waypoints, forcing its motion controller to switch to hover-mode. While the UAV hovers, a new collision-free path is planned.

Lastly, the base station can open log-files created by the rover for later replay, greatly improving post-flight visualization, validation and debugging.

3.2.3 Rover

The rover program was created last. When the embedded x86 computer (described in section 3.1.2) starts up, the Linux system init daemon starts the program automatically. *Hostapd* is also launched and sets the WiFi card into master mode, creating an access point that the base station notebook can connect to. Rover then listens for incoming TCP connections from the base station and opens the virtual serial ports provided by the USB connections to both laser scanners, the navigation system and the low-level flight-controller.

On startup, rover initializes the GNSS/inertial navigation system, configuring options required for time synchronization with the laser scanners or setting the lever arm be-

tween IMU and antenna reference point. Logdata is requested at different intervals and kinematic alignment is started.

Once precise time has been determined by the GNSS receiver, it is synchronized to the host clock. After this, the scanner's sync signal and the resulting event-logs can be used to correctly reference LIDAR and GNSS time.

As soon as kinematic alignment is complete and the navigation system solves time, position and attitude precisely, a submodule fuses data from all sensors in real-time, creating the point cloud that is streamed to the base station.

The same high-level flight-controller used in the simulator is also compiled into the rover. Using the connection to the FlightCtrl-board on the Mikrokopter, it is constantly aware of the flight-state selected by the pilot (explained in detail in section 4.4). When in autonomous mode, the controller computes setpoints for yaw, pitch, roll and thrust to reach the next waypoint, and emits these values to FlightCtrl's low-level controller at 10Hz. For later analysis and debugging, all sensor data and generated outputs from the high-level motion-controller are logged to disk.

A Data Logging

When airborne, the vehicle spends the largest part of the consumed energy to counter the acceleration exerted by gravity and most of its operational time in configurations where halting the drives would cause extensive damage to the platform. This means that in case of problems during flight, the system cannot be halted using common mechanisms like emergency-stop switches.

Even though the operator can instantly regain manual control over the UAVs motion by toggling a switch on the remote control, experience in the field has shown that pilots are typically unable to respond to unexpected, critical flight situations in a timely manner. In the rare cases that motion control is overridden before the UAV collides with either the ground or an obstacle, the operator must then estimate the UAV's three-dimensional orientation and velocity and mentally align to these vectors as a prerequisite to issuing sensible control commands. As a general rule, the resulting control commands often let the vehicle overshoot into configurations that are even more dangerous than the ones that caused the operator to override autonomous motion control in the first place. Not only does this cause the same or even worse degrees of damage, it also means that the reaction and effects of the now-deactivated flight-controller to this otherwise rare flight-state will remain unknown. In other words, after realizing that the UAV has entered a configuration with dim prospects of survival, the pilot may elect to let the UAV complete its crash. By doing so, it becomes possible to diagnose the reason why such a state was reached as well as the reason why the flight controller remained unable to recover the UAV.

Minimizing the incidence of critical flight states and crashes requires regular mainte-

nance of the hardware platform. On the software side, two strategies proved particularly important: first, new flight control algorithms were thoroughly tested in simulation, detecting misbehaviour that wasn't obvious during design and programming. After those tests were successful, the second strategy was the implementation of extensive logging. Being able to reconstruct the system's detailed state is of utmost importance to ensure that after problems manifest during flight, they can be diagnosed and fixed correctly. Less critical to safety, but also important for offline optimization and data analysis is logging of all LIDAR sensor data.

Table 3.4 shows the data being logged during flight, along with the required throughput of the underlying data storage. The latter is located on an SD-card, whose interface is connected to the on-board computer's system controller hub. This configuration allows higher throughput and lower latency compared to a connection via USB, but does not prevent problematic behavior when the interface's throughput is reached. When data is written to plain log files using standard POSIX *write()* semantics, the kernel of the embedded Linux system first buffers the data, then flushes those buffers non-deterministically within a subsequent *write()*-call. This specific call blocks the calling thread until all data is flushed, which lasted up to $300ms$ during experimental trials. This obviously conflicts with the real-time requirements of the high-level flight-controller running on the same system. In the end, the problem was solved by implementing threaded logging using mutex locks and buffer swapping.

3.2.4 Wireless Communication

The UAV's computational resources are too constrained to host all of the required algorithms, so high-level planning and safety features were implemented in the base station. This necessitates a fast and reliable connection between rover and base. As of 2013, German law demands that a line-of-sight between vehicle and both operator and ground station must be maintained at all times. Although this constraint does not guarantee the first Fresnel-zone between the two stations to be unoccupied, it does allow using common off-the-shelf IEEE 802.11 (WiFi) equipment for communication instead of more specialized wireless communication devices operating in lower frequency spectra. Using a 3x3 MIMO spatial multiplexing setup to increase spectral efficiency, the net TCP/IP data throughput in areas with low WiFi-saturation reaches 3.5 Mb/s.

Ascending the OSI model, layer 2 was first devised to employ an adhoc-connection between rover and base in order to facilitate small overhead from infrastructure and encryption as well as fast reconnections. In practice, it turned out that even though this mode of operation is officially standardized in IEEE 802.11, it is seldomly used, untested and thus, unreliable in most driver/chipset combinations. Instead, the rover was configured to act as a common wireless access point (AP) in the 2.4GHz band, with the base connecting as a regular wireless station (STA). Although it might at first seem intuitive to swap the roles of base and rover, the ability to manually reconnect

Table 3.4: An overview of data being logged during flight.

Data	**Description**	**Frequency (Hz)**	**Datarate (kb/s)**
INS commands	ASCII commands sent to the INS for initialization before and configuration during flight.	-	-
INS pose	Current position, velocity and attitude. See Septentrio (2012), *IntPVAAGeod* block	20	1.2 kb/s
INS status	Mean differential correction age, receiver's time and CPU load, orientation covariances, external events and other data, depending on configuration.	1	5 kb/s
IMU raw values	Raw IMU data for vibration and damping analysis. See Septentrio (2012), *ExtSensorMeas* block	50	3.6 kb/s
GNSS reception status	SiS[1] reception status and complex baseband samples for electromagnetic interference analysis. See Septentrio (2012), *BBSamples* block	2	120 kb/s
Differential corrections	Differential corrections required for real-time kinematic GNSS.	1	3 kb/s
Laser scanner data	Distance to reflector of 1080 rays, specified as 16bit unsigned integer. Up to 86.4kb/s per scanner.	40	172.8 kb/s
Flight controller data	400 bytes containing information about a single controller iteration. Consists of timestamp, PID-controller gains, input and output values, the resulting motion command (thrust/yaw/pitch/roll), the flightstate (-restriction), trajectory-start and -goal, hover-position, last known pose and height over ground as determined by LIDAR.	10	4.0 kb/s

the base to the rover during flight has proven much better than waiting for the rover to reconnect after realizing that the connection is lost.

A simple TCP/IP-based application-layer protocol was designed, with the rover program and simulator acting as a server and accepting connections from the base station. The protocol implements streaming of scanned points to the base station and allows exchange of other information. See table 3.5 for a complete listing.

Table 3.5: The TCP/IP-based protocol for communication between rover and base.

Message	Direction	Description
UAV status	Base ← Rover	battery voltage, motion-controller state, air pressure etc.
INS status	Base ← Rover	number of visible/usable satellites, positioning mode, integration mode, CPU load, age of differential corrections, solution covariances etc.
differential corrections	Base → Rover	differential correction data for RTK-GNSS in RTCMv3 format.
lidar points	Base ← Rover	points from one scanner's sweep in float4 format (x/y/z/w), where the w-component indicates the squared distance between scanner and point.
pose	Base ← Rover	the UAV's current position and attitude
controller gains	Base → Rover	sets PID gains of the high-level motion controller
controller gains	Base ← Rover	notification that the high-level motion controller has changed the gains (due to a request from the base station)
waypoint reached	Base ← Rover	notification that the UAV has reached the next waypoint
waypointlist	Base → Rover	a list of generated waypoints, to be navigated by the UAV
flightstate	Base ← Rover	a new flight state from motion controller due to changed waypoint availability or flight-state restriction
flightstate restriction	Base ← Rover	flight-state restriction has changed (because the pilot actuated the switch on the remote control)
flightcontroller values	Base ← Rover	new debug-values from the high-level motion controller, visualized in base station
scanner state	Base → Rover	enables/disables the on-board laser scanners
logmessage	Base ← Rover	a message from the UAV, to be displayed in base station user interface
ping	Base ↔ Rover	used to test connectivity

Chapter 4

Experimental Platform: Theory and Methods

The previous chapter described the engineering and development necessary to construct an experimental airborne platform capable of autonomous exploration. After testing communication, maneuverability, endurance and point-cloud capture in the field using manual remote control had validated the UAV's operational readiness, work related to hardware design and engineering phased out slowly, with optimization, repair and maintenance remaining the only regular tasks in this arena.

The next phase of work was focused on the estimation, planning and control approaches and algorithms necessary to actually implement autonomous exploration in software. As discussed in chapter 2, existing approaches existed for some functionality, but had to be ported to execute on a graphics card. In other areas, like generation of next-best-views, there was no algorithm that promised satisfactory performance even on GPUs, so a new approach was invented, tested in a proof-of-concept implementation, and ported to the GPU. The following sections describe the algorithms needed to reach goals 3, 4 and 5 as defined on page 6.

4.1 Georeferencing Measurements

Because mapping is such an important skill in autonomous robotics, research in this area has been intense, and as chapter 2 has shown, impressive results have ensued from these efforts. It is interesting to note, however, that almost all of the contributions discussed have produced maps that are valid only in the robot's local coordinate reference system (CRS). Since a coherent map is sufficient for localization and navigation within its bounds, research in this field rarely looks beyond local correctness. Even multi-robot mapping approaches - which require a "global" coordinate system for

registration of sub-maps - mostly employ cartesian coordinate systems originating in arbitrary positions on earth's surface.

Inherent to the operating principles of global navigation satellite systems, a precise and globally unique position relative to the earth's surface is solved. The remaining parts of this section discuss how this information can be used to produce georeferenced data. Georeferenced maps, which are a part of this thesis' results, fall into the domains of both geodesy and robotics.

4.1.1 Spatial Reference Systems

Due to the long history of and different requirements in geodesy, many different Spatial Reference Systems (SRS) (or Coordinate Reference Systems, CRS) have been invented and employed. In this thesis' context, SRS are used to unambiguously describe positions relative to the surface of the earth.

The most fundamental classification of SRS is based on the property of projection: map projection is the process of transforming coordinates given in 3D cartesian or polar format into locations on a plane. After projection, a 2D or 3D cartesian CRS can be placed on the resulting plane, creating a projected coordinate system that better lends itself to interpretation by humans. However, all representations of the complex-shaped earth on a plane (i.e., "map") introduce distortions of different kinds, making map-projection a science in itself. Maps can be optimized to preserve true direction, local shapes, area, distance and many other properties, giving rise to countless projected SRS. Depending on the shape of the developable surface used in projection (most often cylinders, cones or planes), only small regions of earth are represented with relatively small errors, while other parts can become heavily distorted.

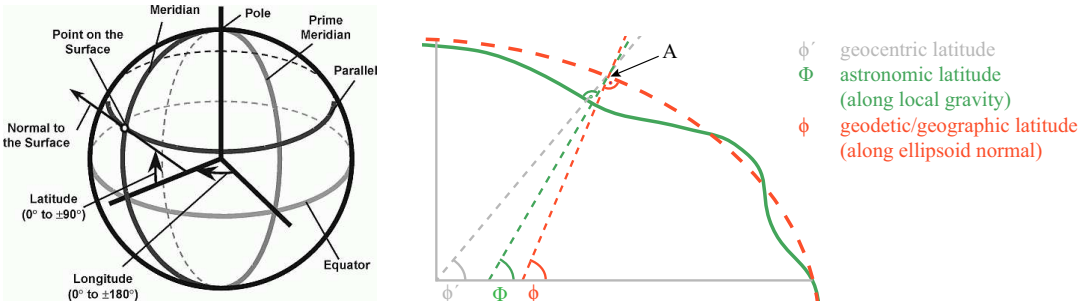
For this reason, precise measurements and computations are most-often executed in unprojected SRS. Interpretation of the produced data by humans, on the other hand, almost always necessitates map projection. As a consequence, using projected SRS, unprojected SRS or a combination of both can be required.

The following paragraphs shall introduce the different coordinate system that are needed to create georeferenced maps.

A Geodetic Spatial Reference Systems

The geodetic SRS is probably the most well-known reference system for locating objects close to the surface of earth. It is a polar coordinate system where locations are specified using latitude ϕ, longitude λ and height h (detailed in figure 4.1(a)). It is often neglected that a triple of these values alone still allows room for interpretation.

In geodesic parlance, the information required to interpret unique and globally consistent positions from aforementioned data is called a "geodetic datum", which defines



(a) (b)

Figure 4.1: (a): Terms and conventions of the geodetic spatial reference system, reprinted from Schuh and Kutterer (2013). (b) illustrates the difference between geocentric, astronomic and geodetic/geographic latitudes using a highly exaggerated geoid and flattening of the reference ellipsoid.

the coordinate system and additional conventions for data format and processing. It consists of:

- the origin of the SRS
- the orientation of the SRS
- a gravitational model
- a model of the shape of the earth

The emergence of ever more precise surveys of planet earth have spawned over one hundred such geodetic datums. Today, the datum used in GPS is the "World Geodetic System 1984" (WGS84), which was defined in 1987 based on data collected through Doppler satellite surveying. Over the course of the last few decades, it has displaced most other datums that were older or used only regionally. The positional offset that arises when coordinates given in reference to one datum are interpreted using another datum is called "datum shift", and can amount to several hundred meters.

The origin of the WGS84 datum is meant to coincide with earth's center of gravity, first established indirectly by observing satellite orbits using satellite laser ranging. According to Lemoine et al. (1998a), the positional error in alignment to the center of gravity of earth (including the mass of the atmosphere) is considered to be in the sub-decimeter range. On several occasions, WGS84 was updated to reflect more precise knowledge of earth's geometry due to more advanced measurement instrumentation: in 1994, GPS measurements were used to enhance the datum's precision, whereas in 1996, it was better aligned with the International Earth Rotation and Reference Systems Service (IERS) reference frame ITRF 94. Because these changes coincided with the GPS-weeks 730 and 843, these datum's fully qualified names are "WGS84 (G730)" and "WGS84 (G843)", respectively. Nowadays, origin, orientation and scale of WGS84 is determined relative to a world-wide network of NAVSTAR/GPS tracking stations, justifying the need for precise information about plate tectonics and continental drift.

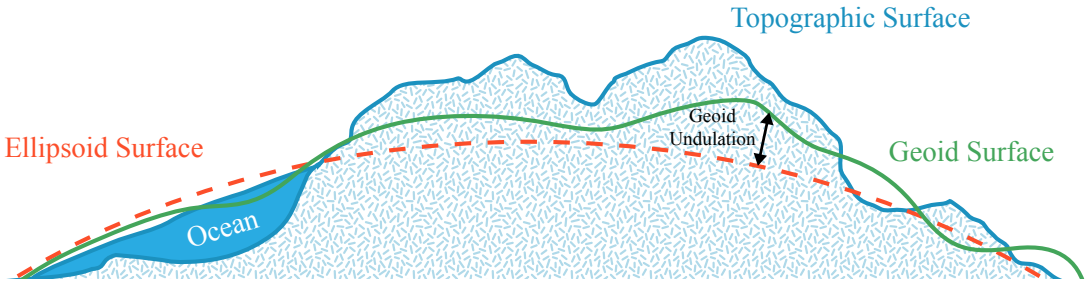


Figure 4.2: Heights can be specified relative to different surfaces, also referred to as the vertical datum. Shown here are the datum's ellipsoid, the geoid and the topographic surface as possible references.

The aforementioned measurements of both strength and direction of gravity on earth's surface were formally standardized as the "Earth Gravitational Model 1996 (EGM96)" Lemoine et al. (1998b). This model defines a geoid, an irregularly shaped surface that matches the shape of the oceans with influence from gravity and earth-rotation, without influence from tides, winds and currents. In other words, EGM96 defines an equipotential surface, meaning that any two points on this surface share the same gravitational potential.

A datum also defines a model of the shape of the earth. For almost all datums, this is an oblate spheroid, formed by rotating an ellipse around its minor axis and optimized to best approximate all of earth's surface. Since earth's radius at the equator is greater than its radius at the poles, the datum is oriented such that the minor axis of its ellipsoid matches earth's polar axis and the major axis lies on earth's equatorial plane. By further introducing a flattening of about $\frac{1}{298.257}$, the geoid undulation (i.e. the distance from the reference ellipsoid's surface along the surface's normal to the geoid, see figure 4.2) on the surface of the earth was globally minimized.

Given a datum, there are several ways of determining the latitude of any point A close to earth's surface: *geocentric* latitude is defined as the angle ϕ' between the datum's equatorial plane and a line connecting the datum's origin with A. Although this is a very intuitive definition of latitude, it is used seldomly and interpretation of latitudes produced using modern GNSS equipment in this manner leads to large errors in positioning.

Instead, coordinates are commonly given in *geodetic* latitude: here, locating A requires computing the normal N to the datum's ellipsoid running through A (N will coincide with the datum's origin only when A lies on the poles or the equator). Then, the angle ϕ between the datum's equatorial plane and N is solved. Figure 4.1(b) illustrates the difference between geocentric and geodetic latitude using an ellipsoid with highly exaggerated flattening.

A similar quirk is to be considered for interpretation of longitude: In WGS84, the meridian at zero degrees of longitude does not pass through the Royal Observatory in

Greenwich. Instead, the datum aligns to the International Reference Meridian (IRM) as defined by the IERS, located about 102 meters west of the Royal Observatory. This offset is a legacy of TRANSIT (a predecessor to NAVSTAR/GPS) and just one example that studying the history of geodesy helps understanding modern positioning procedures and the resulting data.

Without further qualification, measures of height are usually specified in meters relative to the datum's reference surface.

Most GNSS-based inertial navigation systems deliver coordinates in geodetic format using the WGS84 datum, using geodetic latitude, longitude relative to the IRM and height in meters above the reference ellipsoid.

B Geocentric Spatial Reference Systems

As the name implies, geocentric spatial reference systems originate at the center of gravity of earth and its atmosphere. They are cartesian systems and by convention,

- the X axis lies on the equatorial plane and intersects the International Reference Meridian (IRM),
- the Z axis is parallel to the mean earth rotation axis and points towards the International Reference Pole (IRP, North Pole).
- the Y axis is aligned to complete a right-handed orthogonal coordinate system.

The geocentric SRS, shown in figure 4.3, is also named ECEF ("Earth-centered, earth-fixed"), ECR ("Earth Centered Rotational") and "conventional terrestrial" coordinate system. It is similar to the geodetic SRS in that it is fixed to earth in both position and orientation. Because of this property, it is convenient for specifying positions on earth's surface. Other SRS, like ECI ("Earth Centered Inertial") are also of cartesian nature and originate in earth's center of gravity, but do not rotate with earth, so they are more suitable for locating objects that move independently.

Because of its cartesian nature, the geocentric SRS is much better suited for most measurements involving spatial units than the geodetic SRS (e.g. distances, areas, volumes and velocities can be computed directly) and allows for much simpler computations involving vectors. Since LIDAR sensors deliver measured ranges using vectors in their own cartesian CRSs, the geocentric CRS is a commonly used intermediate reference system, as it simplifies the addition of these offsets to the scanner's own position.

ECEF as such is well suited for interpretation of collected point cloud data, with the notable exception of orientation: by nature, point clouds presented in geocentric coordinates will preserve their orientation with respect to earth, so that only point clouds of the north pole would seem "correctly" aligned when visualized together with a ground plane. Moreover, geocentric coordinates are hard to interpret by humans: while estimating an LLA-triple of 50° latitude, 10° longitude and 120m altitude to represent a

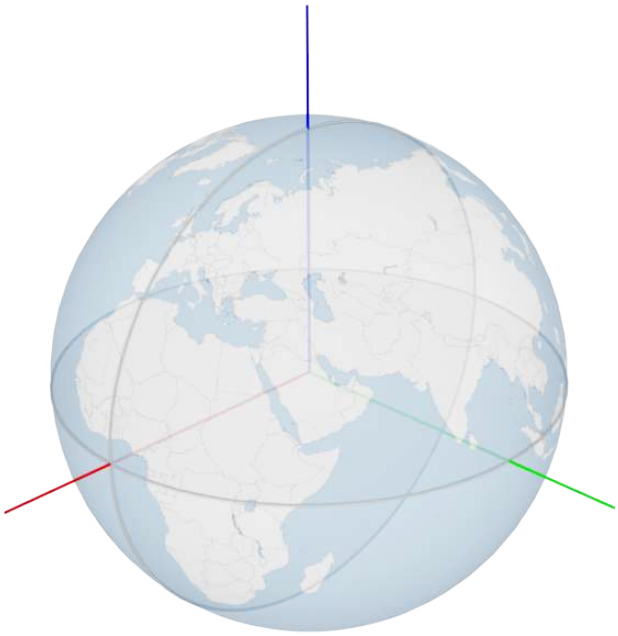


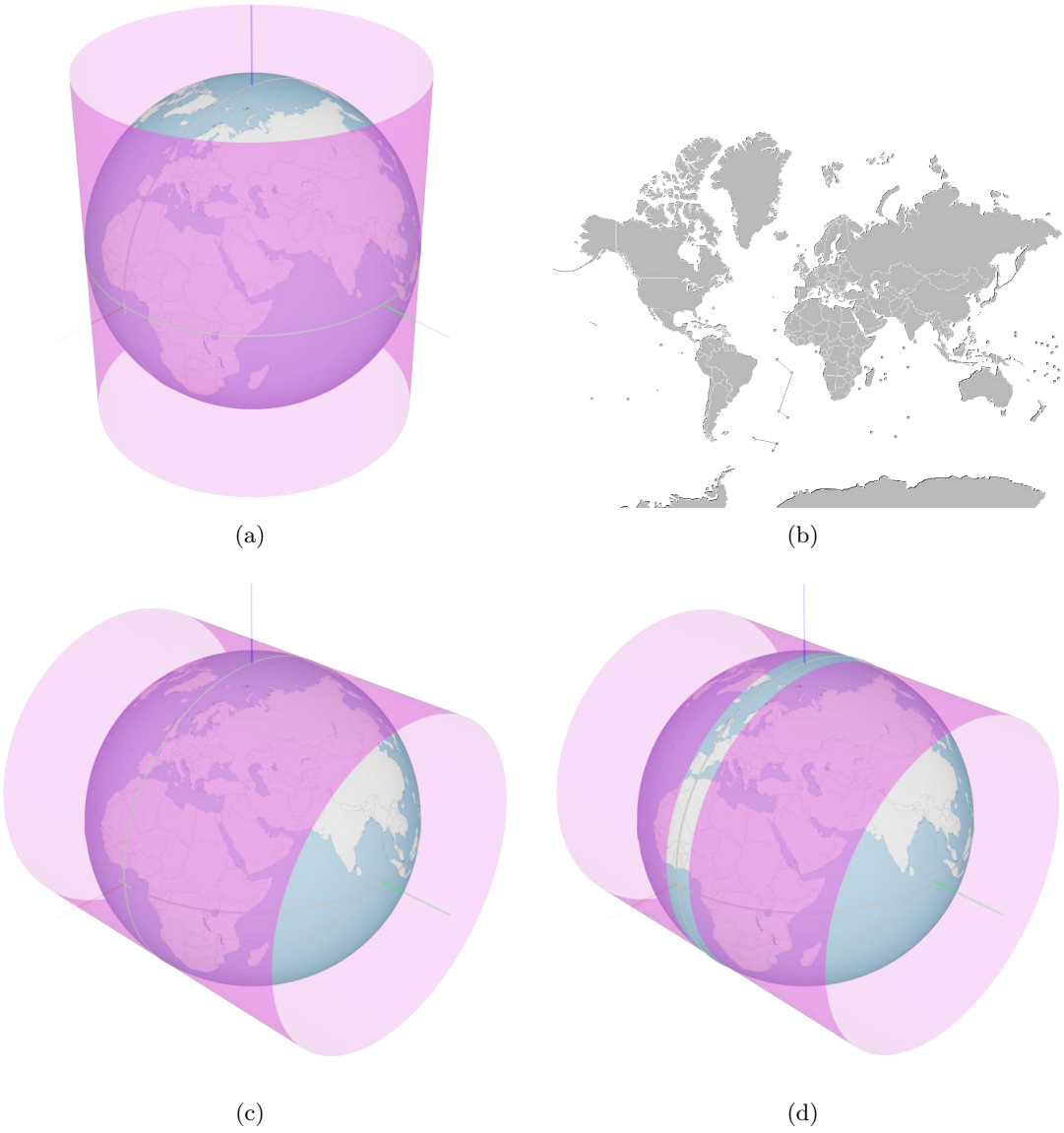
Figure 4.3: The geocentric spatial reference system originates at the center of gravity of earth and its atmosphere. The Z axis (blue) points towards the International Reference Pole (IRP, North Pole), X (red) intersects both the International Reference Meridian (IRM) and the equatorial plane and Y (green) is aligned to complete a right-handed orthogonal cartesian coordinate system.

location in western Europe is relatively straightforward, the corresponding geocentric coordinates of $X = 4,045,532.36m$, $Y = 713,336.50m$ and $Z = 4,862,880.96m$ do not lend themselves well for this task.

C Mercator Projections

The Mercator projection was named after Geradus Mercator, and has been the most common projection used for world maps from its invention in 1569 well into the 1960s; it is an orthomorphic (i.e. angle-preserving) mapping, produced by placing an imaginary cylinder around earth, with both shapes touching at earth's equator (see figure 4.4(a)). In opposition to popular belief, the projection is not achieved by placing a light source at the center of the earth and projecting it's surface onto the developable surface (cylinder), because this would cause extreme distortion at the poles. In fact – like most other projections in use today – the Mercator projection cannot be produced with a real-world, physical light source.

At the projection's central parallel (the equator), map distortion is zero, but it grows towards the poles so much that Greenland appears to be about the same size as Africa (see figure 4.4(b) although it has only about 10% of it's area). By rotating earth on the geocentric Y-axis, any region with an east/west orientation can be aligned to become



(a) (b)

(c) (d)

Figure 4.4: (a): The Mercator projection allows distortion-free mapping at its central parallel (the equator, also called the *Standard Parallel*). (b): Distortion of Mercator projections grows towards the poles. (c): The Transverse Mercator Projection is re-oriented for low distortions in regions extending further north/south than east/west. (d): By downscaling the cylinder to 99.96% of its original size, the cylinder cuts through earth and two rings of zero distortion appear about 180km east and west of the projection's central meridian, creating a *Secant* Transverse Mercator Projection. Mercator projection map of the world courtesy of Geordie Bosanko.

the projection's central parallel, allowing accurate projection. About 200 years after Mercator, Johann Heinrich Lambert invented the *Transverse* Mercator projection: for many applications, it is desirable to re-orient the distortion-free ring, so that it is not aligned with earth's parallels, but with it's meridians instead (figure 4.4(c)). Depending on the area to be mapped, the cylinder can be rotated around earth, making any line of longitude become the projection's central meridian.

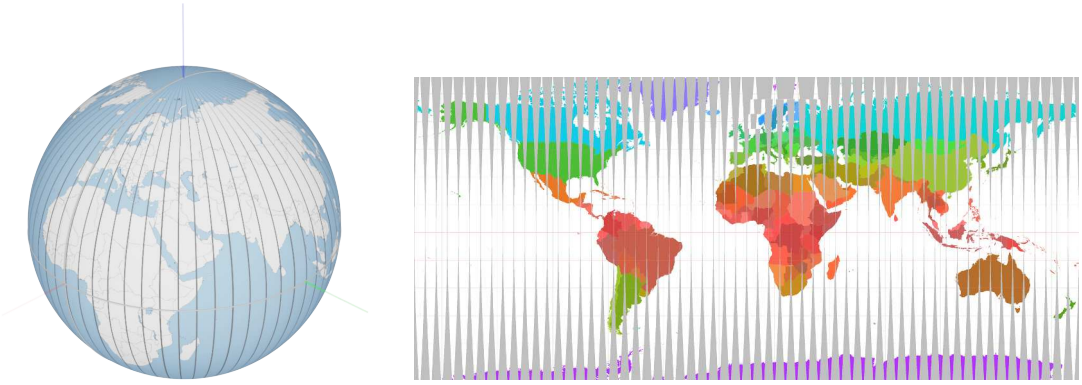
The only remaining disadvantage was the increasing distortion in areas further away from the central meridian. In order to minimize this distortion, a *Secant* Transverse Mercator projection is produced by shrinking the imaginary cylinder: it starts to cut through earth ("to cut", latin: *secare*), increasing the distance between it's surface and the central meridian. Consequently, distortion at the central meridian increases, but two distortion-free rings east and west of the central meridian start to emerge, extending outwards as the cylinder is shrunk. As a result, the amount of overall distortion is reduced.

D UTM Spatial Reference System

The Universal Transverse Mercator projection system was developed in 1947 by the North Atlantic Treaty Organization (NATO). After realizing that Mercator projections are highly accurate within 5° of their central parallel/meridian, NATO extended a single Transverse Mercator projection into a *projection system* of 60 Transverse Mercator projections. UTM divides earth into 60 longitudinal zones, each 6° of longitude wide. A central meridian is placed into the center of each zone, creating a system of 60 Transverse Mercator projections (pictured in figure 4.5(a)). Using this system, no place on earth is further than 3° from its central meridian. UTM also uses a secant cylinder, scaled down to 99.96%, which results in two rings of zero distortion appearing 180km east and west of the central meridian. Together with the large number of zones, this enables virtually distortion-free mapping in earth's non-polar regions.

After projection into a plane, each UTM zone hosts a cartesian coordinate system. The X axis denotes the easting and originates at the central meridian, while the Y axis specifies the northing and starts at the equator. All values are specified in meters. By convention, the X-value of the central meridian is set to 500.000m. This is known as *false easting* and eliminates negative coordinates. In similar fashion, the equator remains at 0m on the northern hemisphere, but on the southern hemisphere, negative Y-values are prevented by assigning a *false northing* of 10.000.000m.

UTM is often confused with UTMREF/MGRS (the Military Grid Reference System), which extends UTM by subdividing its zones every 8° of latitude. These grid cells are then further subdivided into squares with 100km sidelength and assigned combinations of letters, allowing for very compact coordinates. In this thesis, UTM and MGRS are treated as different SRS.



(a) (b)

Figure 4.5: (a): The Universal Transverse Mercator projection system subdivides earth's surface into 60 vertical zones, each of which spans 6° of longitude and hosts its own cartesian sub-coordinate system. (b): the resulting zones in projection (image courtesy of Wikimedia Commons).

Projecting point coordinates into UTM yields the following advantages:

- point clouds can be visualized directly in OpenGL without further scaling or translation. Thus, all data remains georeferenced even in GPU memory.
- coordinates are specified in meters, which eases computations of distance, area and volume.
- as opposed to geocentric data sets, no rotation is required in order to align the point cloud's ground plane with a cartesian CRS used for processing and visualization.

4.1.2 Algorithms for Transformation and Conversion

In the context of LIDAR-generated point clouds, we define the property *georeferenced* to mean that every single point in the point cloud is specified in globally unique coordinates. These can be either geodetic (i.e. longitude, latitude, altitude relative to a given datum), geocentric coordinates or a projected coordinate system like UTM/UPS (i.e. hemisphere, zone, false easting, easting, false northing and northing). Other SRS can also be used (although this is becoming rare in practice), provided that conversions to other, more common, SRS are defined.

Figure 4.6 illustrates the conversions and transformations required to produce a georeferenced point cloud:

In step 1, the geodetic position with respect to the WGS84 datum as solved by the INS is converted to the geocentric coordinate system (using the same datum), which is achieved using the closed-form approach given in equations 4.1 to 4.4 (a and e define the shape of the datum's ellipsoid: a is the length of its semi-major axis, e is its

eccentricity). At the same time, geodetic latitude and longitude imply an orientational offset of the vehicle's local zero-orientation (0° pitch and roll, heading north) with respect to the geocentric frame. This rotational offset is best understood geometrically (and presented in figure 4.7) – it remains to convert this interpretation into arithmetic. For accuracy and computational efficiency, the implementation concatenates the corresponding sequence of quaternion multiplications instead of employing matrix arithmetics.

$$X = (N(\phi) + h)cos(\phi)cos(\lambda) \tag{4.1}$$

$$Y = (N(\phi) + h)cos(\phi)sin(\lambda) \tag{4.2}$$

$$Z = (N(\phi)(1 - e^2) + h)sin(\phi) \tag{4.3}$$

where

$$N(\phi) = \frac{a}{\sqrt{1 - e^2 sin^2(\phi)}} \tag{4.4}$$

Step 2 converts the cardan-angles output by the INS into a quaternion. The vehicle's zero-orientation is defined as pointing north, with the vehicle-down direction aligning with the local gravity vector. Applying the given cardan-angles for heading, pitch and roll at an arbitrary location on earth is presented graphically in figures 4.8 (a) to (d).

Because LIDAR data is commonly given in cartesian coordinates relative to the scanner itself, a transformation into the INS frame is performed in step 3. Simple multiplication with a classic 4*4 transformation matrix comprised of 32bit floats yields sufficient accuracy, as the LIDAR sensor's maximum range is comparably small.

Next, the vehicle's local orientation generated in step 2 is merged with the orientation of the local coordinate system (computed in step 1). The result of this quaternion multiplication provides another quaternion representing the UAV's current orientation relative to the geocentric frame (shown in figure 4.8 (d)).

The scanned point, which is now given in the INS frame, is transformed using this quaternion. Geometrically speaking, this yields a 3D vector representing the offset from the UAV to the point in the geocentric frame.

Thus, step 6 simply adds the point's geocentric offset to the UAV's geocentric position as computed in step 1, arriving at the point's geocentric position. At this time, the point is correctly georeferenced and could be exported into GIS (geographic information system) applications. For reasons stated in section 4.1.1 D, point's coordinates shall be solved in UTM coordinates, which requires further processing.

Step 7 transforms the point's position into the geodetic frame. As opposed to the opposite transformation performed in step 1, converting geocentric into geodetic positions is not a straightforward procedure. Traditionally, geodetic coordinates were solved ap-

proximately and then refined using e.g. Newton's method until convergence within less than ten iterations. An early solution using a closed-form equation is presented in Heikkinen (1982), which was later simplified by Vermeille (2002). Still, Vermeille's approach had a few shortcomings, such as numerical instabilities around earth's center. Some of these problems were resolved in Karney (2011), who also maintains a C++ library implementing solutions for modern computer architectures. Focusing on robustness and speed, Karney (2012) supports multiple geodesic conversions required in the code accompanying this thesis.

Finally, the computed longitude is used to determine the UTM/UPS zone surrounding the point. Once the zone is computed, it shall remain static for all following points, which becomes important in the rare case that a point cloud spans multiple zones. Given this information, a forward projection from the geodetic SRS into UTM/UPS is performed as defined in Hager et al. (1989). Depending on the application, the resulting coordinates can now be uploaded into GPU memory or saved into a file. Starting in 2003, the American Society for Photogrammetry and Remote Sensing established the free and open LAS file format ASPRS Board (2003). Similar to the RINEX format in the GNSS industry, LAS is currently supported by most GIS applications.

4.2 Computing Next Best Views

The algorithm for generation of next-best-views is inspired by other researcher's contributions concerned with creating watertight 3D models of real-world environments (like e.g. Holenstein et al. (2011)): watertightness is not only a desirable property for completely reconstructed models, but also a helpful test to find gaps that have remained throughout the mapping process. The following sections focus on executing this test as quickly as possible in order to find gaps of a desired minimum size and then derive useful waypoints from the results in the following stages.

As an initial setup, the algorithm requires a predefined bounding-box b that contains both the UAV and the environment to be mapped. A 3D uniform grid G_{IG} of information gain subdivides b, with each cell carrying a scalar value indicating the information gain achievable by scanning it. Few seconds after take-off, the point cloud that has been streamed to the base station is downsampled (see section 4.2.3) into a sparse version, called the collider cloud C. Gaps in this cloud are then detected by using a particle system which simulates pouring water in the form of N_P particles P over C (see figure 4.9). We can postulate that whenever a particle $p \in P$ first collides with a collider $c \in C$ and later arrives at b's bottom plane, it has successfully passed through a gap in C. The algorithm stores the position P_{col} of every particle p's last collision with C. Whenever a particle p reaches the bounding box's bottom plane, P_{col} is looked up, and, if present, the information gain value of G_{IG}'s cell containing p_{col} is increased. Leaving reachability concerns aside, cells of G_{IG} in which many particles slide through gaps in the point cloud intuitively represent possible waypoints.

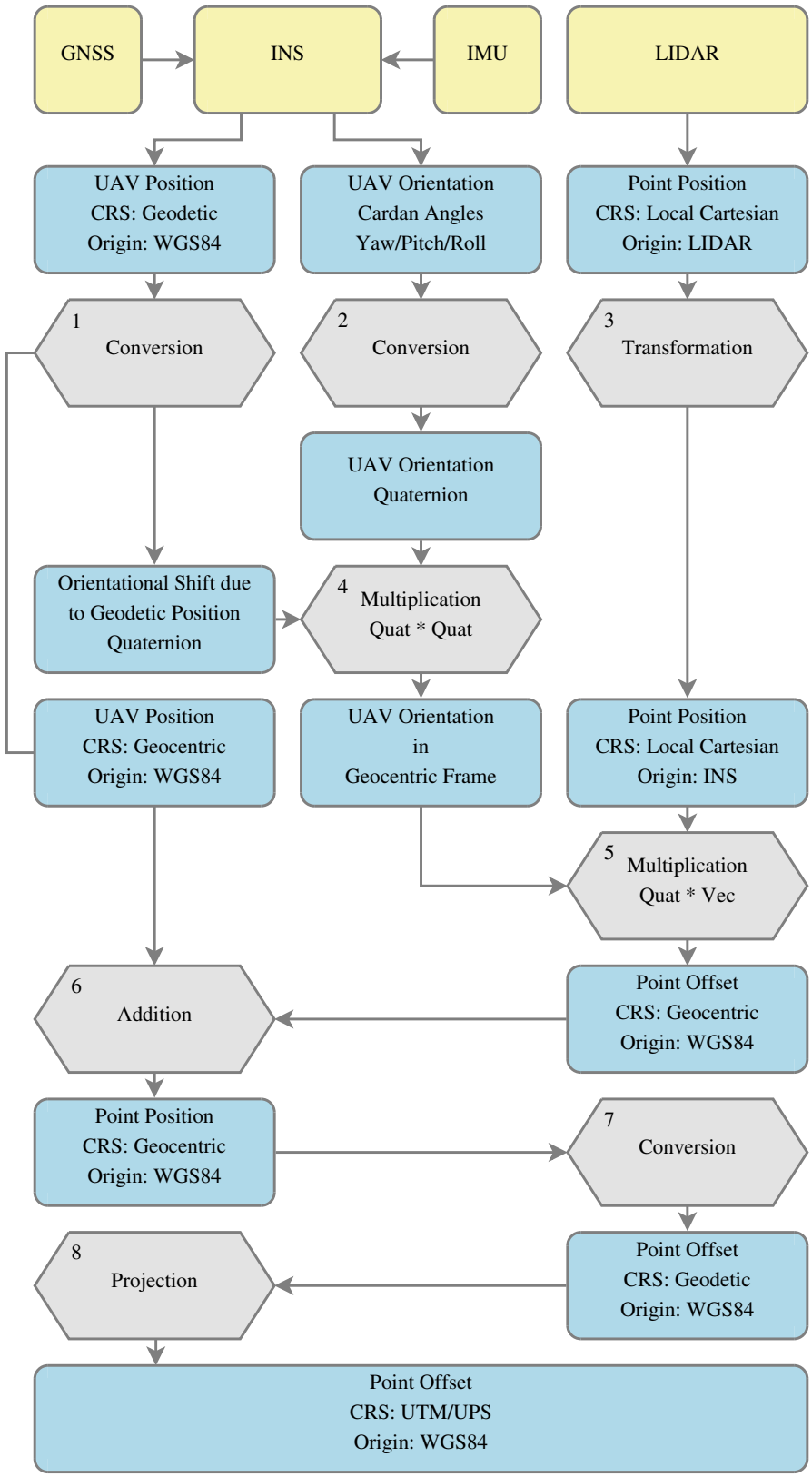


Figure 4.6: A diagram depicting the conversions and transformations required to produce a georeferenced point cloud. The whole process must be repeated for every point.

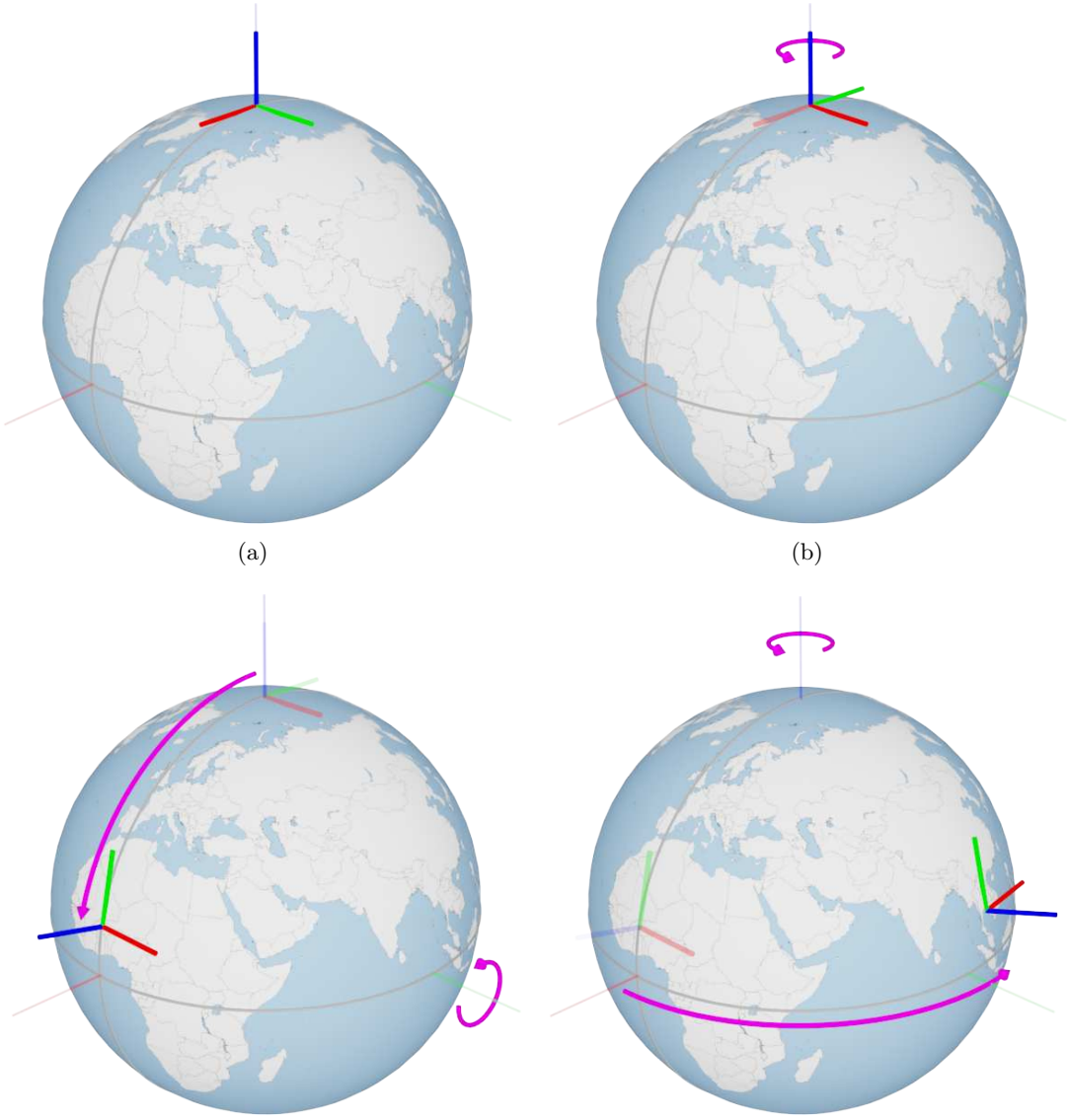
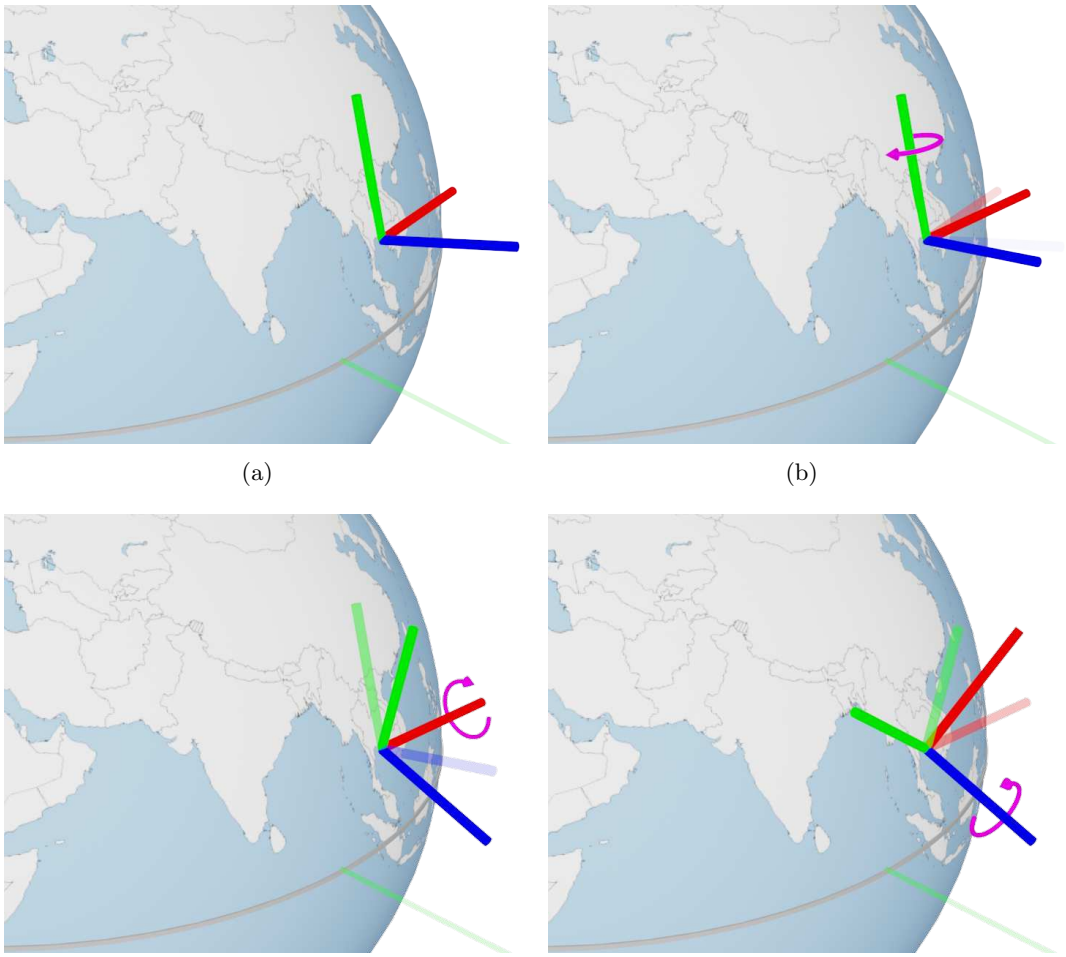


Figure 4.7: A part of step 1 in figure 4.6 is computing the orientation of a vehicle-fixed CRS in zero-orientation (0° pitch/roll, heading north) in Bangkok ($\phi = 13.79°latitude$, $\lambda = 100.5°longitude$) with respect to the geocentric frame: (a): the initial orientation matches the geocentric frame. (b): a rotation of 90° around the geocentric Z axis prepares the local CRS to align with the geodetic convention (X points east, Y point north and Z is up). (c): a rotation of $(90 - \phi°)$ around geocentric Y, followed by (d): a rotation of $\lambda°$ around the geocentric Z axis results in the desired orientation.



(a) (b)

(c) (d)

Figure 4.8: Step 2 in figure 4.6: after deriving the orientation of the UAV's CRS with respect to the geocentric frame ((a) equals 4.7(d)), the vehicle's local orientation is converted from cardan angles specified by the INS into a quaternion. This requires matching conventions with regard to rotation order and axis directions: in case of the Septentrio INS used in this thesis, rotations are applied in the order roll (b), pitch (c), yaw (d)

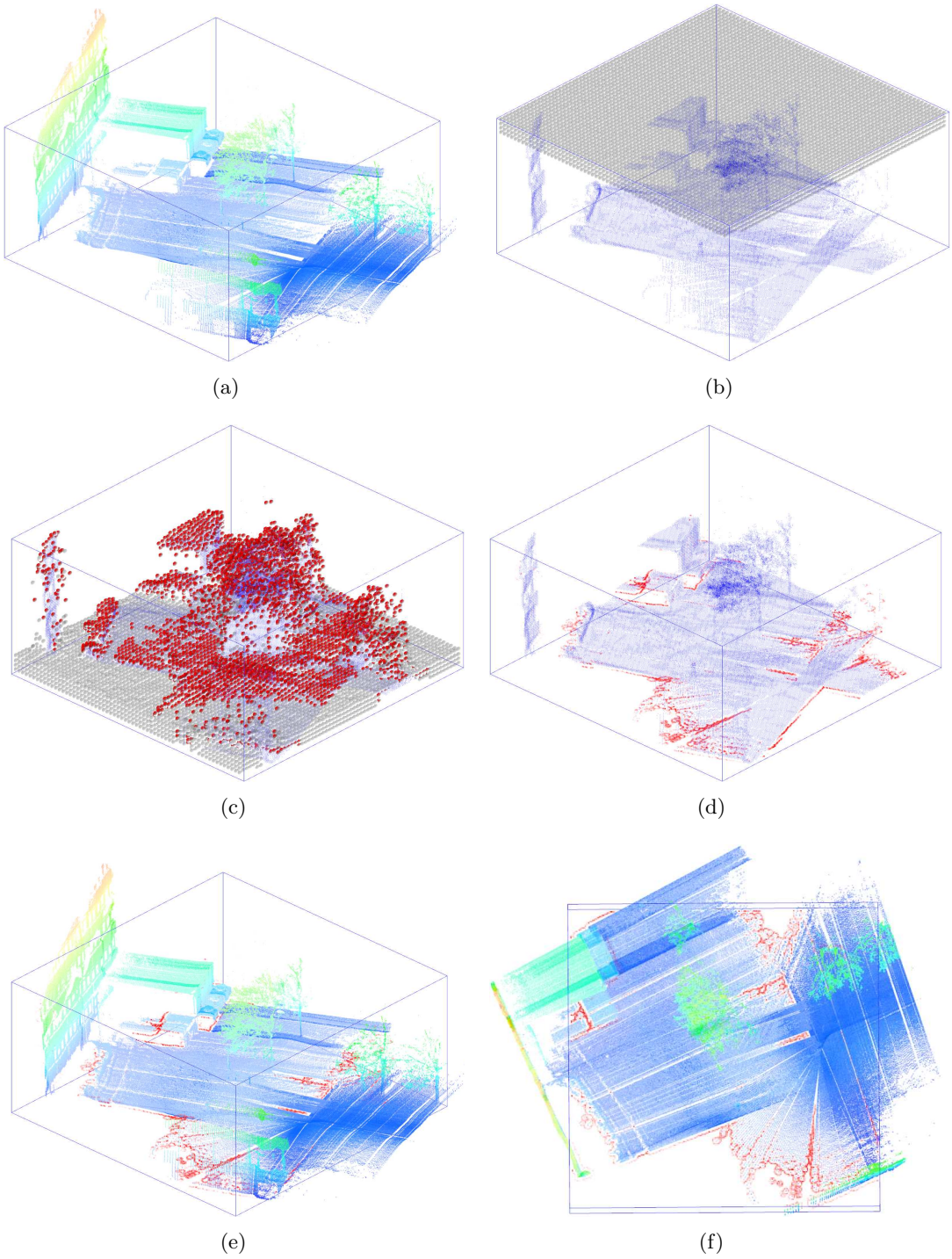


Figure 4.9: Overview of the process: a) shows predefined bounding box b and the initial point cloud from the onboard laser scanner. b) shows 16k particles $p \in P$ in gray being poured over downsampled cloud of colliders $c \in C$ in blue. c) depicts one timestep of simulation, with falling particles $p \in P$ that have collided with at least one $c \in C$ in red, others remaining gray. d-f) overlay a visualization of G_{IG} over sparse and dense point clouds, showing cells promising higher information gain in more saturated red.

4.2.1 Proof-of-Concept implementation on the CPU

In a first step, this approach has been tested using a proof-of-concept implementation. Focusing on fast development instead of fast runtimes, the *bullet* physics library (Coumans, 2013) was employed to execute collision detection and handling on the CPU. Using the *bullet* API, the point cloud was managed using a fast dynamic bounding volume tree based on axis aligned bounding boxes for the broadphase collision detection. When collision between particles and colliders were detected, they were processed using a callback mechanism that saved the particle's current position into a vector. The process was visualized using legacy/immediate-mode OpenGL, meaning that for every frame rendered, all colliders and particles had to be re-uploaded into GPU memory space using separate OpenGL function calls. Colliders were drawn rather efficiently as single points, but particles were displayed as spheres, which necessitated triangulation on the host.

While the implementation proved the potential of the concept, both physics simulation and visualization turned out to be far too slow to handle the enormous number of collision-tests necessary for rapid detection of small gaps. At the time, *bullet* did offer support for parallel broadphase execution on some CPU architectures, using e.g. "Sweep and Prune" approaches to find potentially overlapping pairs (Coumans, 2008). However, an investigation of how the *bullet*-based approach scales with the number of objects simulated (see figure 5.13) resulted in the realization that even a linear speedup on an eight-core CPU with disabled visualization would not provide sufficient performance for real-time operation. As of December 2010, the rigid body engine in *bullet* did not provide robust GPU-backed algorithms. Even though physics processing on graphics cards has been on the project's development roadmap for more than four years by now, implementation of this goal has been very slow and in fact, is not complete so far.

Because the approach is intrinsically geometry-based, an interactive visualization is vitally important to understand limitations, problems and tuning of parameters. As opposed to *bullet*, the complexity of a custom physics simulation between particles and colliders can be far lower – after all, a full-featured physics engine has to simulate more complex objects like concave meshes, whereas this thesis' simulation merely requires simulating interactions between static points, moving spheres and axis-aligned planes.

Algorithms involving massive throughput of operations of comparably simple complexity are well suited to GPU implementation, especially when the data's interdependencies are low. Thus, the proof of concept was ported to CUDA, running an NVIDIA graphics card, and optimized for real-time applicability. Porting to an architecture-agnostic language such as OpenCL would principally have enabled the code to run on any supported hardware. In practice, though, support of OpenCL on Linux has traditionally been sparse and did not compare favorably to the performance, robustness and support

offered by CUDA. OpenCL's advantage of exploiting all available processors at the same time (i.e., CPU *and* GPU) is unlikely to improve performance either: today, powerful GPUs are always discrete units, connected to the CPU via PCI-Express bus. Balancing the computational load between all devices introduces algorithmic complexity and further limitations stemming from memory-latency and limited -througput. The theoretical peak throughput of a 16-lane PCI Express 3.0 bus is 128 GBits/s, whereas the memory-throughput of the GDDR5 built into the mid-range NVIDIA GTX 670 (Kepler-based GK104 with a 256bit memory-bus) graphics card lies just above 192 GBits/s, or about 150% of the host/device connection.

4.2.2 Handling Point Clouds

When the INS is delivering precise poses, all points from the laser scanner are fused into a global coordinate system on the rover and streamed to the base station in batches. Given a maximum of 1080 points per scan at a scan-rate of 40Hz, the theoretical point rate is 43.200 points per second per scanner. Some of the rays are reflected by the vehicle's own booms, propellers and landing gear, returning distances below 0.5m. Other rays directed upwards rarely receive reflections, as they often point into the sky. After filtering values that have very close or no reflections, an average of 600 points per scan remain to be fused, yielding a point rate of about 48.000 points per second from both scanners. As a single point is currently represented using four IEEE754 single-precision floats, this translates to a memory requirement of 0.77 Mb per second. Given the maximum flight time of 15 minutes, allocating 700 megabytes of memory on the graphics card for the dense point cloud is sufficient, even when downsampling is disabled. Thus, on the base station, points are saved to disk and uploaded into the GPU's memory space by appending them to a vertex buffer object VBO_{dense}. OpenGL's VBOs can be mapped into CUDA address space, avoiding copies between interleaving visualization and processing stages.

4.2.3 Data Reduction on the GPU

As shown in section 5, the particle simulation's bottlenecks are collision-detection and -processing. In every iteration of the simulation, this process is first executed between the simulated particles themselves, then executed between the particles and the collider cloud. Thus, optimizing it becomes key to detecting gaps and generating waypoints as quickly as possible.

In the first stage, particles are collided against each other in order to simulate a fluid. While optimizing for speed by reducing the number of particle/particle collisions is possible, it would mean changing the particle system's behavior: if collision checks were skipped between some neighboring particles, they would be allowed to penetrate each other, making the simulated fluid lose the desired property of being non-compressible.

Trying to save collision tests by executing them in comparably coarse time steps would first allow the particles to interpenetrate more deeply, which would cause large repelling forces to throw them apart in the next collision detection phase. For this reason, all neighboring particles must have pair-wise collision checks applied to them and decreasing the total number of particles remains the only way of reducing the computational effort of particle/particle collision checking.

In the second stage, particles are collided against points of the collider cloud, testing its permeability. Depending on the distance between the UAV's range scanner and an object being scanned as well as the number of scan-passes, that object's surface can be sampled using very high point densities. Colliding particles against all of those points (as depicted in figure 4.10(a)) causes many pairwise collision-checks to be executed. Because only the permeability of the collider cloud is tested, it is possible to increase the distance d_{max} between neighboring colliders (reducing the cloud's density) as long as particles of radius r are kept from passing through the cloud (see figure 4.10(b)). Mathematically, this is guaranteed as long as the margin m between particle-center and the baseline (2D) or plane (3D) between colliders remains larger than zero. This geometric constraint for non-permeability can be trivially formulated using the Pythagorean theorem:

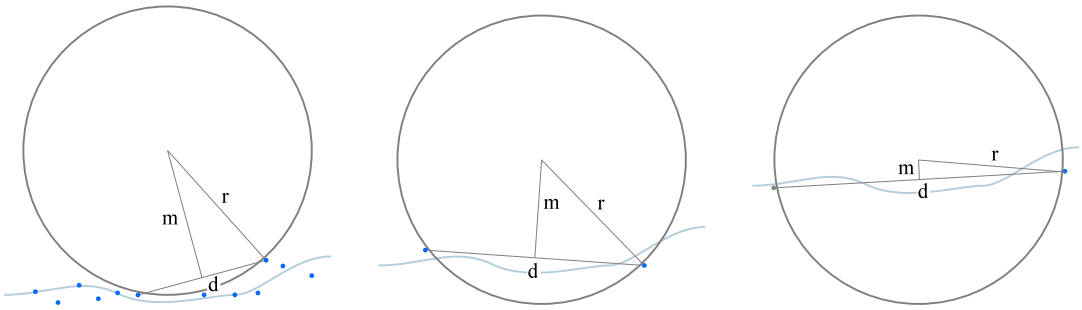
$$m^2 = r^2 - (\frac{d_{max}}{2})^2 > 0$$

Theoretically, the constraint $m >= 0$ could be chosen, effectively setting $d_{max} = 2r$. In practice, a particle of radius r requires the distance d_{max} between two colliders to remain less than $2r$ for two reasons: firstly, particles tend to fall between very sparse colliders and get stuck, meaning that they no longer move and fulfill their purpose of sampling the represented surface. Secondly, the simulation is performed in discrete time steps, meaning that the distance traveled by particle p in-between collision checks is a function of its velocity $p_{\vec{v}}$ and the simulation's time step Δt. As soon as their product exceeds m, a particle can pass through the collider cloud by the offset applied due to its velocity in the integration phase (see figure 4.10(c)). Thus,

$$p_{\vec{v}} * \Delta t < \sqrt{r^2 - (\frac{d}{2})^2}$$

must hold to prevent false waypoints from being generated.

The sparse collider cloud is stored in another, smaller vertex buffer object VBO_C on the graphics card. As previously mentioned, points are stored using four floats instead of three for reasons of better alignment in GPU memory. For points of the dense and collider clouds, the fourth component (w) stores the distance between the scanned point and the range scanner. Because the positional accuracy of the platform's localization is far superior to its orientational accuracy, the distance offers a good metric for the



(a) A particle being collided against an unprocessed, noisy and dense point cloud for tests of permeability. Collision-tests will be performed between the particle and every collider in its cell, resulting in many unnecessary memory transactions, calculations and comparisons.

(b) The same point cloud, sparsed to reduce the required number of collision tests. Because the sparsing algorithm accounts for the particle's size, permeability remains unchanged.

(c) The same point cloud, sparsed too much. Even though the cloud remains mathematically impermeable to the particle, the discrete nature of the simulation can allow the particle to pass through the colliders.

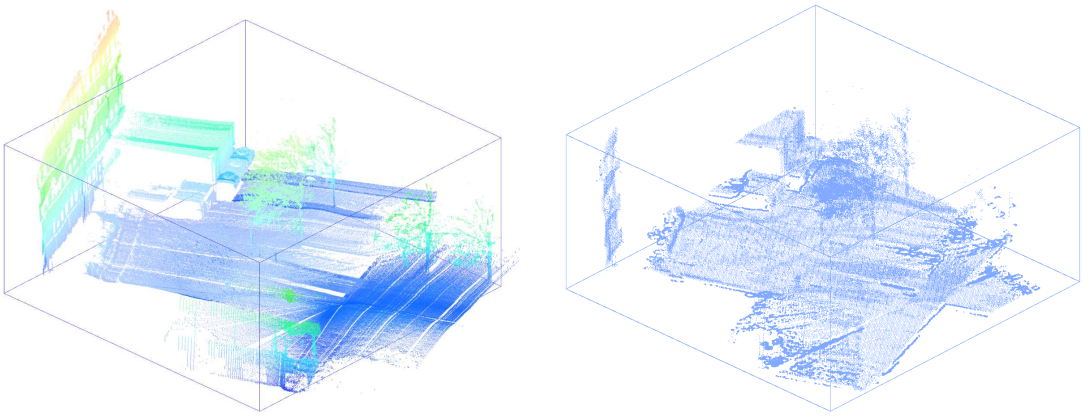
Figure 4.10: Different levels of downsampling applied to the collider cloud C (stored in VBO_C) and its effects on performance and correctness.

point's precision, which will be used in the following processing stages.

When new waypoints have to be generated, the point cloud in VBO_{dense} is downsampled into a sparse version C located in VBO_C, as shown in figure 4.11. An overview of this process is given in figure 4.12: after being cleared, VBO_C is filled with all points in VBO_{dense} that are located within b, until either all points in VBO_{dense} are copied or VBO_C is full.

Next, the colliders in the target buffer are reduced. The classic approach for implementations on the CPU is to sequentially iterate through all the points, deleting close neighbors using algorithms and data structures optimized for radius-based neighbor-queries. Our implementation is similar in that it uses a uniform grid as a spatial decomposition technique (see figure 4.13(a)), but different in that it is done in parallel on all available streaming multiprocessors. A spatial hash table based on the uniform grid G_{SHT} (the subscript SHT denotes "spatial hash table") with $N_{G_{SHT}} = N_{G_{SHT}}(x) * N_{G_{SHT}}(y) * N_{G_{SHT}}(z)$ cells is created to enable efficient access to neighboring colliders.

The SHT requires two vectors, C_{index} and $C_{cellhash}$, associating the collider's index in C_{pos} with the hash of G_{SHT}'s cell containing it. This is shown in figure 4.13(b) and allows finding all particles within a given grid cell during the following downsampling stage. The hash table for C_{pos} is constructed in step 3 by launching one thread for every $c \in C$. Afterwards, it is sorted according to the CellHash column. This step is performed by a radix sort algorithm implemented on the GPU, which is described in detail by Merrill and Grimshaw (2011).



(a) (b)

Figure 4.11: (a): A dense point cloud, as streamed from the rover during flight. (b): The same point cloud, cropped to the region to be mapped (pre-defined bounding box b) and sparsed to reduce the required number of collision tests.

Step 4 sorts colliders according to their grid cell's hash value, thereby moving geometric neighbors into adjacent memory locations.

Afterwards, step 5 uses one thread per collider to populate a pre-allocated vector $CellFirst$ of $N_{G_{SHT}}$ integers, where $CellFirst[i]$ stores the first row of the sorted spatial hash table referencing a collider in grid cell with hash i. If the cell does not contain any colliders, its value is set to $UINT_{max}$. In analogous fashion, $CellLast$ is populated so that $CellLast[i]$ stores the last row of the sorted spatial hash table referencing a collider that is contained in the grid cell with hash i. An example is shown in figure 4.13(c). On completion, finding all colliders in grid cell i now boils down to accessing C_{pos} using all indices found in $C_{index}[CellFirst[i]]$ to $C_{index}[CellLast[i]]$

For optimization, a locality preserving hashing function is used to assign hash values to G_{SHT}'s cells. Sorting positions according to the hash values of the cells containing them increases the probability of fetching positions of other particles in the same and neighboring grid cells from neighboring memory locations, maximizing the memory bandwidth utilization by using coalesced access patterns. Pre-sorting colliders also best leverages the L2 caches for global memory access that emerged with CUDA compute capability 2.0, as collider positions will already be cached when neighboring threads need to fetch their positions in order to execute collision tests against them.

Now, one thread is launched for every collider: it iterates all other colliders in the same and neighboring cells (operating in adjacent memory locations to exploit the GPU's cache) and checks whether they are located closer than the threshold distance (usually defined to be slightly less than the particle's diameter, as explained above). If so, the collider with the larger w-component (which is likely to be less precise) gets overwritten with values of (0/0/0/0).

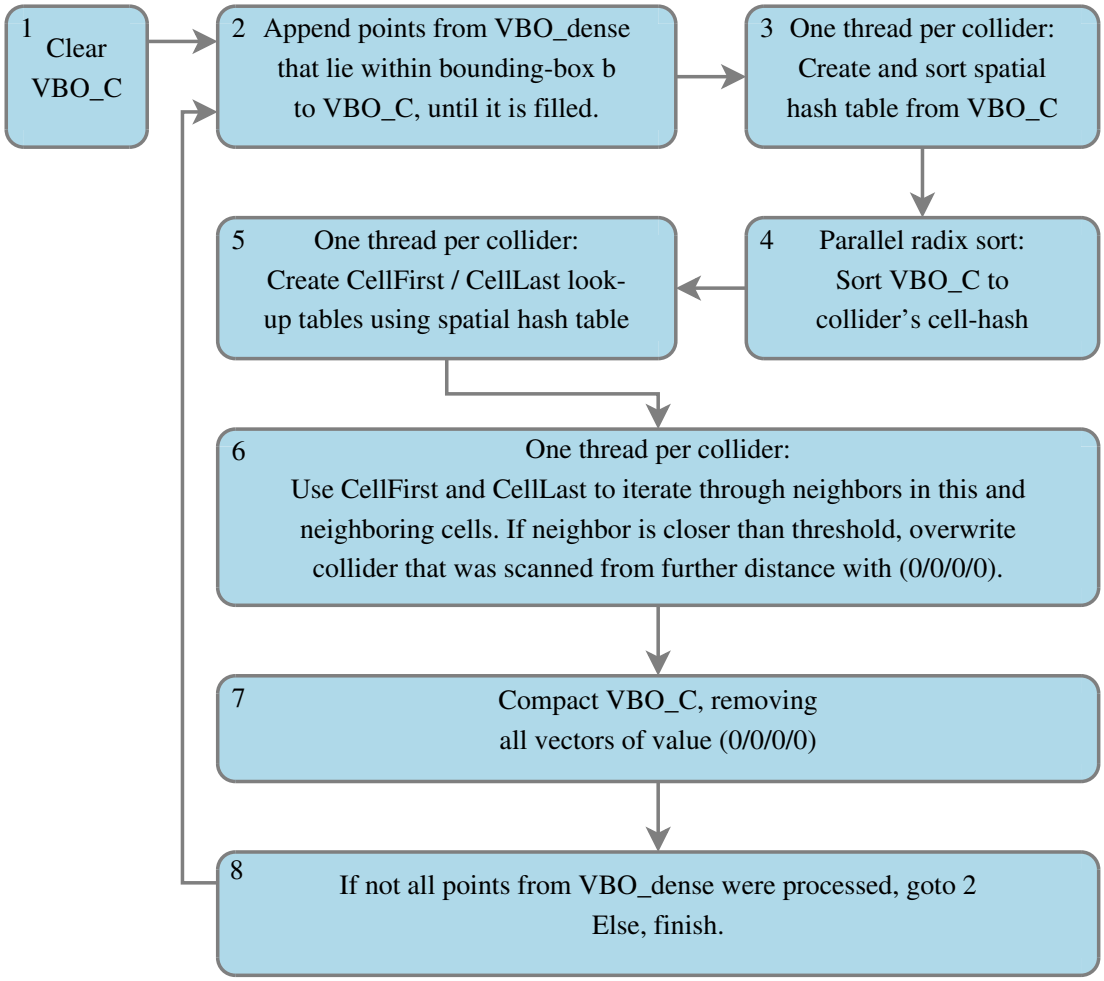
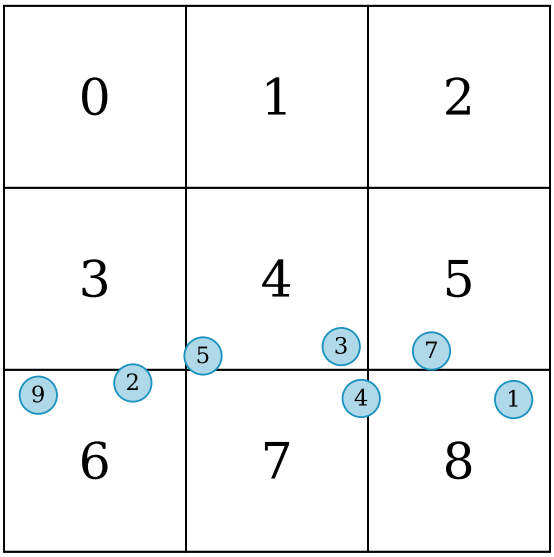


Figure 4.12: A flow-chart describing the process of downsampling the point cloud in VBO_{dense} to a sparse version C stored in VBO_C.

In step 7, the values in VBO_C can finally be compacted, in effect removing all zero-points. This free space allows more points from the dense point cloud to be appended and reduced again, as decided in step 8.

In contrast to the following particle simulation, which needs to be executed many hundred times, this code is executed only once in preparation for waypoint generation, totaling less than 20 calls during typical flights. Although the procedure contains many steps that process large amounts of data, the massively parallel execution allows downsampling of the point cloud typically within less than 100 milliseconds. For benchmarks, please see section 5.

Independent of the threshold distance for neighbor removal, there is no guarantee that the holes developing during downsampling are still small enough to keep particles from passing through. Ensuring this would require a computationally involved analysis of each point's neighborhood, slowing down execution. Given that particle size defines the minimum size of gaps to be detected, we have found that using their radius as



(a) Diagram of colliders $c \in C$ from the dense point cloud, located in cells of uniform grid G_{SHT} (diagram is inaccurate in that it is showing only a two-dimensional grid for clarity and visualizing points as spheres). Cells are labeled using their hash values $C_{cellhash}$ (in this case, the hash value is simply the cell's row-major index). Colliders are labeled using their index C_{index} in C_{pos}.

$Index$	$Collider\ CellHash$ $(C_{cellhash})$	$Collider\ Index$ (C_{index})
0	4	5
1	4	3
2	5	7
3	6	2
4	6	9
5	7	4
6	8	1

N_C

(b) Spatial hash table, associating every collider to its cell's hash value in the uniform grid. After construction, this table is sorted according to the CellHash column.

$Cell\ Hash$	$C_{index}First$ $(CellFirst)$	$C_{index}Last$ $(CellLast)$
0	$UINT_{max}$	*
1	$UINT_{max}$	*
2	$UINT_{max}$	*
3	$UINT_{max}$	*
4	0	1
5	2	2
6	3	4
7	5	5
8	6	6

$N_{G_{SHT}}$

(c) The collider cell lookup table, populated using the spatial hash table: for each cell's hash value, $CellFirst$ and $CellLast$ store the first and last row of the spatial hash table referencing colliders in that cell. This allows a fast, memory-coalesced access to all colliders in a grid cell. The value $UINT_{max}$ in $CellFirst$ denotes empty cells, while * in $CellLast$ consequently denotes an undefined value.

Figure 4.13: Data structures in GPU memory, required for downsampling the dense point cloud into the collider cloud C in VBO_C.

Idx	C_{pos}	P_{pos}	P_{vel}	P_{col}	G_{IG}
0	xyzw	xyzw	xyzw	xyzw	0
1	xyzw	xyzw	xyzw	xyzw	0
2	xyzw	xyzw	xyzw	xyzw	0
3	xyzw	xyzw	xyzw	xyzw	0
...	...	...	...	...	...

Figure 4.14: Four vectors of float4 are allocated, storing N_C collider positions C_{pos} as well as N_P particle positions P_{pos}, the same amount of particle velocities P_{vel} and particle/collider collision-positions P_{col} in GPU memory. Also, memory for $N_{G_{IG}}$ scalar cell-values of a grid G_{IG} of information gain is allocated.

threshold distance for neighbor removal prevents those degenerate cases, as long as the source point cloud is sufficiently dense and evenly sampled in this region. Since the latter condition is exactly what the algorithm is designed to test, there is no reason to implement aforementioned neighborhood analysis.

4.2.4 Testing for watertightness on the GPU

The downsampling of the collider cloud described in the previous section might seem to be a performance optimization conceptually unrelated to the particle simulation. In fact, it is not: downsampling point clouds and simulating large amounts of particles share a large and important part of computational logic: neighbor searching. Thus, the highly-parallel data structures supporting fast access to neighbors for downsampling can also be used to enable high-speed particle simulation, as seen in figure 4.15. Instead of removing colliders that are closer than a threshold distance, we add repelling forces to particles that have approached up to a distance closer than their diameter.

Section 4.2.3 already introduced the sparse collider-cloud stored in VBO_C, holding a vector of collider-positions C_{pos}. Testing this point cloud for watertightness involves simulating N_P particles being poured over it to detect the remaining gaps.

This requires further data structures to be allocated in the graphics card's memory, as shown in figure 4.14. Before simulation, P_{col} and G_{IG} are initialized with zeros, indicating that no particles have collided yet and no information gain has built up. Using these structures, a single iteration of the particle simulation is processed as described in Algorithms 1 and 2:

To build the spatial hash table, N_P threads write their thread-id into $P_{index}[threadId]$ and the hash of the cell containing $P_{pos}[threadId]$ into $P_{cellhash}[threadId]$ in lines 2-5. Afterwards, both vectors are sorted according to $P_{cellhash}$ using a parallel radix sort.

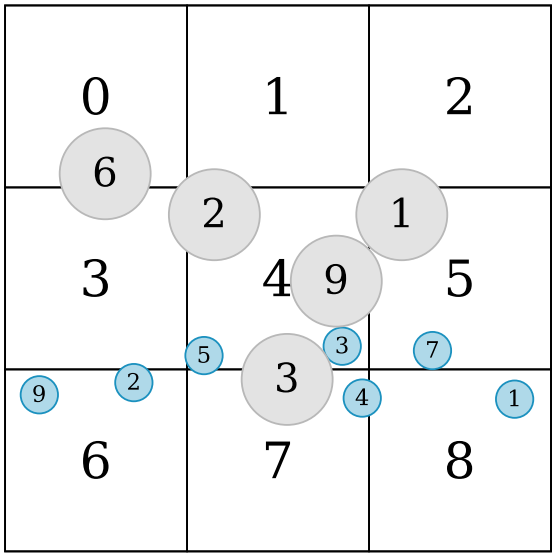
The cell lookup table is populated as listed in lines 10-20: one thread per particle reads $P_{cellhash}[threadId]$ into the temporary $cellHash[threadId]$, located in the given thread-block's shared memory space. In this way, each cell hash is fetched from global memory

Algorithm 1 Build support datastructures

```
1: // Particles move, so rebuild their spatial hash table (SHT) in every iteration.
2: for each core i < N_P do in parallel
3:     P_index[i] ← i
4:     P_cellhash[i] ← getCellHash(G_SHT, P_pos[i])
5: end for
6: radixSortKeyValue(P_cellhash, P_index)
7:
8: // Using the spatial hash table, build the cell lookup table for particles.
9: // Allocate sharedHash in shared memory to coalesce global memory access.
10: allocate sharedHash[N_P]
11: for each core i < N_P do in parallel
12:     sharedHash[i + 1] ← P_index[i]
13:     synchronizeThreads( )
14:     if sharedHash[i] ≠ sharedHash[i − 1] then
15:         P_cellFirst[P_cellhash[i]] ← i
16:         if i > 0 then
17:             P_cellLast[sharedHash[i + 0]] ← i
18:         end if
19:     end if
20: end for
21:
22: // If points were inserted into the collider cloud, also
23: // rebuild their spatial hash and cell lookup tables
24: if ColliderCloudChanged then
25:     for each core i < N_C do in parallel
26:         C_index[i] ← i
27:         C_cellhash[i] ← getCellHash(G_SHT, C_pos[i])
28:     end for
29:     radixSortKeyValue(C_cellhash, C_index)
30:
31:     // Build cell lookup table for colliders
32:     allocate sharedHash[N_C]
33:     for each core i < N_C do in parallel
34:         sharedHash[i + 1] ← C_index[i]
35:         synchronizeThreads( )
36:         if sharedHash[i] ≠ sharedHash[i − 1] then
37:             C_cellFirst[C_cellhash[i]] ← i
38:             if i > 0 then
39:                 C_cellLast[sharedHash[i + 0]] ← i
40:             end if
41:         end if
42:     end for
43: end if
```

Algorithm 2 Execute massively parallel test for watertightness of C

```
1: // Collide all particles
2: for each core i < N_P do in parallel
3:     force ← (0, 0, 0)
4:     // Retrieve hashes of all 26 neighboring cells
5:     cellHashes ← getNeighborHashes(P_pos[i])
6:
7:     for each h ∈ cellHashes do
8:         // Collide particle p against colliders in all neighboring cells
9:         for j ← C_cellFirst[h], C_cellLast[h] do
10:            force += collideDEM(P_pos[i], C_pos[j])
11:        end for
12:        // Save p's current position in case of collision with collider
13:        if force ≠ (0, 0, 0) then
14:            P_col[i] ← P_pos[i]
15:        end if
16:
17:        // Collide particle p against other particles, updating it's velocity
18:        for j ← P_cellFirst[h], P_cellLast[h] do
19:            force += collideDEM(P_pos[i], P_pos[j])
20:        end for
21:        P_vel[i] ← (force + P_vel[i])
22:    end for
23: end for
24:
25: // Collide every particle with the bounding box b's walls, then integrate
26: // the resulting motion over time, moving the particles.
27: for each core i < N_P do in parallel
28:    P_vel[i] ← damping * (P_vel[i] + (g * Δt));
29:    P_vel[i] ← collideWithBoundingBox(b, P_pos[i])
30:    P_pos[i] ← P_pos[i] + (P_vel[i] * Δt)
31:    if P_pos[i].y < b.min.y then
32:        P_pos[i].y ← b.max.y
33:        // If the particle reached the floor, increment
34:        // the corresponding cell's value in G_IG
35:        if Pcol[i] ≠ (0, 0, 0) then
36:            G_IG[getCellHash(Pcol[i])]+=1
37:            Pcol[i] ← (0, 0, 0)
38:        end if
39:    end if
40: end for
```



(a) Particles $p \in P$ (with radius, shown in gray) and colliders $c \in C$ (i.e. points from downsampled point cloud, blue) located in cells of uniform grid G_{SHT} (showing a two-dimensional grid for clarity). Cells are labeled using their hash values, while particles and colliders are labeled with their index in P_{pos} and C_{pos}, respectively.

Index	*Particle CellHash* ($P_{cellhash}$)	*Particle Index* (P_{index})
0	0	6
1	4	9
2	4	2
3	5	1
4	7	3

N_C

(b) Spatial hash table, associating every particle to its cell's hash value in the uniform grid. After construction, this table is sorted according to the CellHash column.

Cell Hash	$P_{index}First$ (*CellFirst*)	$P_{index}Last$ (*CellLast*)
0	0	0
1	$UINT_{max}$	*
2	$UINT_{max}$	*
3	$UINT_{max}$	*
4	1	2
5	3	3
6	$UINT_{max}$	*
7	4	4
8	$UINT_{max}$	*

$N_{G_{SHT}}$

(c) The particle cell lookup table, populated using the spatial hash table: for each cell's hash value, *CellFirst* and *CellLast* store the first and last row of the spatial hash table referencing particles in that cell. This allows a fast, memory-coalesced access to all particles in a grid cell. The value $UINT_{max}$ in *CellFirst* denotes empty cells, while * in *CellLast* consequently denotes an undefined value.

Figure 4.15: Additional vectors in GPU memory, required for particle simulation. Their structure is completely analogous to that described in figure 4.13.

only once. After synchronization of all threads in the warp (ensuring that all hashes have been loaded), they are compared against the cell-hash of the previous particle in $cellHash[threadId - 1]$. Because $P_{cellhash}$ is sorted, a failed comparison means that the previous particle is located in a different cell, allowing $CellFirst$ and $CellLast$ to be populated.

To detect and process collisions, N_P threads fetch $P_{pos}[threadId]$ and compute the hash value of G_{SHT}'s respective cell. They then iterate through its own and all $3^3 - 1 = 26$ neighboring cells in the grids of the SHTs for *both* other particles and colliders (lines 2-23), re-using the vectors built in section 4.2.3. To ensure that particles in non-neighboring cells cannot collide, their diameter must be less than the grid cell's smallest side. For every cell visited, $CellFirst$ and $CellLast$ are used to quickly access the indices of contained particles and colliders. When collisions occur, $P_{vel}[threadId]$ is updated using forces computed by the discrete elements method (Harada, 2007) and $P_{pos}[threadId]$ is copied to $P_{col}[threadId]$ (line 14), allowing this particle's last collision to be retrieved in case it reaches the bounding box's bottom plane in the next step.

The particle-motion is integrated by launching N_P threads: each kernel first updates $P_{vel}[threadId]$ according to a given time step t, gravity $g \in \mathbb{R}^3$ and a global damping value. It also collides $P_{pos}[threadId]$ against the inner sides of b, confining the particle to the bounding box. Then, $P_{pos}[threadId]$ is updated according to $P_{vel}[threadId]$ and t and used to check whether $P_{pos}[threadId]$ has reached b's bottom plane. If so, that particle's last collision is looked up from $P_{col}[threadId]$ and is, if not zero, used to increment the information gain of G_{IG}'s cell containing it in line 36.

After multiple iterations, particle simulation is terminated (see section 5 for a discussion of termination criteria). G_{IG}'s cells are sorted in order of decreasing information gain values, and their respective positions in $\mathbb{R}^3$ are computed. After close waypoints are merged, the results are passed to the path planner.

4.3 Computing Trajectories

As input, the path planner requires the UAV's current position, the sparse collider cloud and a list of waypoints.

In the first step, another uniform 3D grid is created, whereby one byte per cell is allocated in GPU memory. Each cell's value is initialized to zero. The sparse collider cloud's points are then processed in parallel manner, so that every cell containing a point is set to a value of 255 (depicted in dark gray in figure 4.16, left), in effect creating a three-dimensional occupancy grid (figure 4.16, left). When using a 26 (8 in 2D) neighborhood for path finding, generated paths can traverse diagonally between occupied cells, causing the vehicle to pass unsafe parts of the environment. As seen

in the middle of figure 4.16, dilation circumvents this problem by wrapping those cell-patterns with occupied cells, also adding a safety margin between vehicle and geometry. Thus, for increased safety, the occupancy grid is dilated in the next step: each thread processes one cell, looks at the 26 (8) neighboring cells and assigns a value of 254 if it finds an occupied neighbor. Because the spatial grid usually contains more cells than the GPU has streaming multiprocessors, the grid will be processed in (spatial) batches. If dilated cells were also assigned a value of 255, threads executed in consecutive warps would be unable to discriminate between occupied and dilated neighbor-cells, in effect dilating the occupied cells even further.

Next, the list of waypoints is processed: because waypoints are generated near edges of the point cloud, they are usually placed in occupied cells of the occupancy grid and must be moved to neighboring, free cells: because the main LIDAR's field-of-view is oriented downwards, the UAV is expected to map the gaps by flying above them. All waypoints are processed in parallel fashion: while its containing grid-cell is occupied, the cell above its current position is checked. If it is also occupied, its 8 horizontal neighbors are probed. This process is repeated until either a free cell is encountered (and the waypoint is placed in its center), or the distance between the original waypoint and the current position exceeds the LIDAR's range of 30m, in which case the waypoint is deleted. The result is an occupancy grid with a list of waypoints in its free cells.

Ignoring the occupancy grid, a CPU-based TSP-solver is used to reorder the waypoints into the shortest path. Because we produce less than 20 waypoints at a time, this computes in few milliseconds.

The occupancy-grid-cell containing the UAV's current position is marked as the start cell and its value is set to 1, while the grid-cell of the first waypoint becomes the goal cell. The path from start to goal is found by launching one thread per grid-cell to look up its own cell's value, and, if zero, the neighboring 26 (8) cell's values. If their minimum is non-zero, that minimum is incremented by one and saved in the cell. This process is repeated until the goal-cell has received a non-zero value (shown in figure 4.16, right) or a maximum number of iterations has been reached, indicating that no path was found. In case a path was found, a single thread starts at the goal cell and collects the 3D world-positions of the cells it passes while hopping towards neighboring cells with smaller values, eventually reaching the start cell. If a path was found, the world-positions of *all* visited cells will be converted to waypoints and path planning continues with start and goal assigned the previous goal and the next waypoint, respectively. If no path was found, the first waypoint is deleted and path planning continues with the next waypoint. When path planning completes, the collected waypoints are sent to the rover's high-level motion-controller.

During autonomous flight, the occupancy grid is constantly updated using the collider cloud's new points. At a configurable interval (currently 2 seconds), all waypoints' grid-cells are checked. If one cell is found to be occupied, the path is replanned. An example of this scenario is shown at the end of the accompanying video.

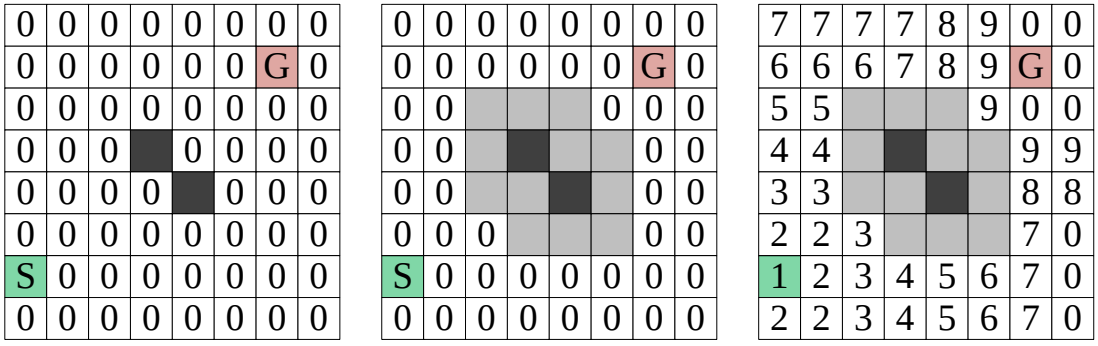


Figure 4.16: Parallelized path planning using a uniform occupancy grid. Left: Initialized occupancy grid with start and goal cells marked and cells containing colliders being marked as occupied. Center: After dilation, searching for traversable cells in the diagonal neighborhood is safe. Right: Populated occupancy grid for a path between start and goal after 8 iterations. In the next iteration, the goal cell will receive a value of 10, terminating the search.

4.4 Motion Control for Autonomous Flight

Multirotor UAVs require complex motion control, because they react to control-input in non-linear fashion and can be affected by potentially large disturbances (wind). They also are under-actuated and non-holonomic, as the number of controllable degrees of freedom (DoF) is less than their total DoFs. Since roll and pitch determine lateral velocity and thus position in cartesian space, their attitude cannot be controlled independently, posing additional constraints.

As mentioned in section 3.1.2, the Mikrokopter UAV comes with a FlightCtrl-board that integrates a MEMS-grade IMU and implements a low-level PID motion controller. In the absence of control input, it keeps the platform's heading constant and pitch/roll at zero degrees.

The FlightCtrl can read up to 12 channels from the remote control. Each has a resolution of 8 bit, producing signed values from -127 to 128. Values for thrust, yaw, pitch and roll can be sent using the remote control or using an open protocol (Buss and Busker, 2012) on its serial port at a rate of roughly 20Hz, with acknowledgement-packets being generated for every such motion-control-packet. This is called "ExternControl" in MikroKopter-parlance. Using the values read from the gyroscopes, accelerometers and an internal mixing-table, control values are converted to motor-speeds and sent to the brushless controllers. Documentation on the implementation of the low-level controller does not exist. Although its source-code is open, it is written mostly in German and contains few comments. A deeper analysis can be found in Sa and Corke (2011).

The four channels used for thrust, yaw, pitch and roll are mapped to the respective sticks, while one channel is mapped to a tri-state switch used for flight-state restriction. The FlightCtrl is configured to activate ExternControl when this channel's value is

above 128. Even when activated, for safety reasons the FlightCtrl will use the remote control's thrust value as an upper bound to the thrust specified on the serial port. This configuration makes simple and safe testing of autonomous flying possible, as it allows the pilot to quickly overrule the high-level motion controller in case of problems. The high-level, PID-based motion controller is implemented in the rover program on the UAV. It reads the flight state restriction channel using the same serial port and changes states accordingly - the resulting flight states for both controllers are listed in table 4.1. States transition automatically from *ApproachWaypoint* to *Hover* when there are no more waypoints and back, when new waypoints arrive.

When entering *Hover*-state, the UAV's current position is set as the target and values for pitch, roll and thrust are computed to steer towards that target. For thrust, the controller simply determines the error between current and desired height, while for pitch and roll, a desired velocity over pitch and roll axes is derived from the vector towards the target. This velocity is proportional to the distance, but limited according to another channel's value (between 0 and 2 m/s, set by a potentiometer on the remote control). Then, current and desired velocity are used to compute the error. Velocity-based control was implemented after flying in windy conditions had demonstrated that position-based control for pitch and roll was insufficient to reach the target position, and adding an integral component to counter this introduced complications when the UAV changed orientation. For yaw, a non-linear P-component is used to ensure that the front is always pointed away from the position of lift-off. Thus, when the pilot (assumed to be positioned near the position of lift-off) switches from *Hover* to *User-Control* state, manually controlling pitch and roll does not require knowledge of the UAV's orientation.

ApproachWaypoint-state is similar to *Hover*-state, with the next waypoint being the target. It is different from *Hover*-state in that the controller first yaws towards the next waypoint and then pitches forward, as the INS produces a more precise attitude solution when flying forward. To avoid slow and clumsy cycles of "orient towards target", "fly forward" and "slow down", the controller starts yawing and rolling towards the second queued waypoint shortly before it has reached the first. The approach is only slowed when closing in on the last queued waypoint or when the second queued waypoint will be situated behind the UAV.

The update frequency of the high-level controller is limited by the availability of reliable information about position and attitude. As mentioned in section 3.1.3 C, the inertial navigation system is capable of producing sequences of one fused pose followed by 4 IMU-integrated poses at a rate of 50 total poses per second. In this configuration, the UAV showed unpredictable and dangerous behavior during tests, which was later determined to be caused by the controller's derivative component reacting to the UAVs position seemingly jumping between the last IMU-integrated pose to the next, fused pose by up to 20cm (see figure 5.4). As a result, the INS was configured to deliver only fused poses, albeit at a lower rate of 10Hz. As evidenced in the accompanying video,

Table 4.1: Different states of low- and high-level motion controllers, based on flightstate restriction and presence of waypoints.

Flightstate Restriction Switch	**Channel value**	**Low-Level controller state ("Extern Control")**	**High-Level controller state (waypoints absent)**	**High-Level controller state (waypoints present)**
UserControl	-107	Disabled	UserControl	UserControl
Hover	20	Enabled	Hover	Hover
ApprWpnt	128	Enabled	Hover	ApprWpnt

this rate is still sufficient for flight control.

Proportional and derivative gains for thrust, yaw, pitch and roll have been determined by performing test flights in different wind conditions, with the UAV attached to a long sea-fishing rod for safety.

4.5 Summary

This chapter discussed the algorithms that achieve the platform's key functionality. Computation of next-best-views was presented in detail along with the optimizations needed to reach real-time performance on the available hardware. The resulting codes for georeferencing, generation of next-best-views and trajectory computation were integrated into the base station (see section 3.2.2), while motion control code was integrated into the rover program (see section 3.2.3).

With all functionality now in place, the following chapter 5 documents the experimental phase.

Chapter 5

Experiments and Results

After presenting the UAV in chapter 3 and the most important algorithms in chapter 4, the following sections discuss the tests performed with the system as well as the achieved results. The experimental strategy is laid out to validate the goals set in the first chapter, and thus, roughly follows their order. This means that first experiments were conducted to verify more fundamental system properties (i.e., flight safety, localization, and mapping in open terrain, later in more obstructed environments), then to validate the more complex components (i.e., motion control, computation of next-best-views and trajectory generation) that build on the system's basic functionality.

Further documentation of experiments and results can be found in the form of a video attachment to Adler et al. (2014)[1]. Also, algorithms can be tested in real-time using the simulator provided at `https://www.github.com/benadler/octocopter/`.

5.1 Flight Safety and Environmental Conditions

As opposed to industrial robotics, mobile robots are designed to work in less structured and more dynamic environments. Of those, the outdoors remains the most difficult environment to master: this means that the system should be tolerant to wide ranges of parameters such as temperature, humidity, wind, solar irradiation and air pressure, none of which can be influenced. While robustness against e.g. solar irradiation is necessary for the LIDAR sensor to gather meaningful data, tolerance against wind is a more critical property, as it affects flight safety.

The UAV has been tested at outside temperatures from as low as -10°C up to +30°C. Unsurprisingly, achievable flight times decrease significantly when approaching freezing temperatures. At temperatures below 5°C, the battery can become unable to supply the current required to spin up all motors at the same time, which causes the low-level

[1] Also available at `https://www.youtube.com/watch?v=0Z1UyJnaNhU`

Table 5.1: Listing of the number of experiments performed. These numbers represent only those log files that were archived - many more experiments took place. Also omitted are experiments in simulation, which preceded every outdoor testing. Number of tests is an inclusive figure, e.g. 96 of the >100 flights included mapping.

No. of tests	Functionality tested	Results
>100	Flying	Section 5.1
96	Flying, mapping	Section 5.2
72	Flying, mapping, high-level motion control (hovering)	Section 5.1 and 5.4
44	Flying, mapping, high-level motion control (appoaching waypoints)	Section 5.1 and 5.4
8	Flying, mapping, high-level motion control, waypoint generation, collision avoidance	Section 5.3 and 5.4

flight-controller to abort the start-up sequence. Once start-up succeeds, the ongoing discharge keeps the battery warm enough to allow flying at temperatures as low as -10°C.

In principle, the system could fly in rain or fog after sealing its electronic components. In practice, sealing adds weight and can be irreversible, complicating diagnostics of electronic failures and the ensuing maintenance. Furthermore, with LIDAR being fundamental to the mapping procedure, and given these sensor's tendencies to register reflections from rain drops and suffer shortened range in fog, the benefit of this capability is questionable. Because the goals set in the first chapter do not comprise flying in rain, the UAV was not designed to withstand any form of moisture and thus, it has not been tested in either rain or fog. Nevertheless, one series of flights was performed during snow (for the video attachment to Adler et al. (2013))[1] and yielded satisfactory results after laser-ranging returns reflected from snow flakes were removed using a simple filter.

Classic airborne LIDAR measurements are commonly executed at heights above 1000m, resulting in a nadir-oriented FoV. With the UAV flying at altitudes no higher than 25m, scanning angles more oblique and even sideways is important for mapping. Due to this, the LIDAR sensors are often oriented such that they scan directly into the sun. The Hokuyo UTM-30lx laser scanner has proven to be extremely robust in these situations, producing no false range measurements even when directly exposed to sunlight. This robustness is rather important to the platform's operation, because a false-positive range measurement translates to a false point in the point cloud. Such a point can cause a free cell in the occupancy grid to be marked as occupied, preventing trajectory-

[1] Video available at `https://www.youtube.com/watch?v=tGbgkrS2zdw`

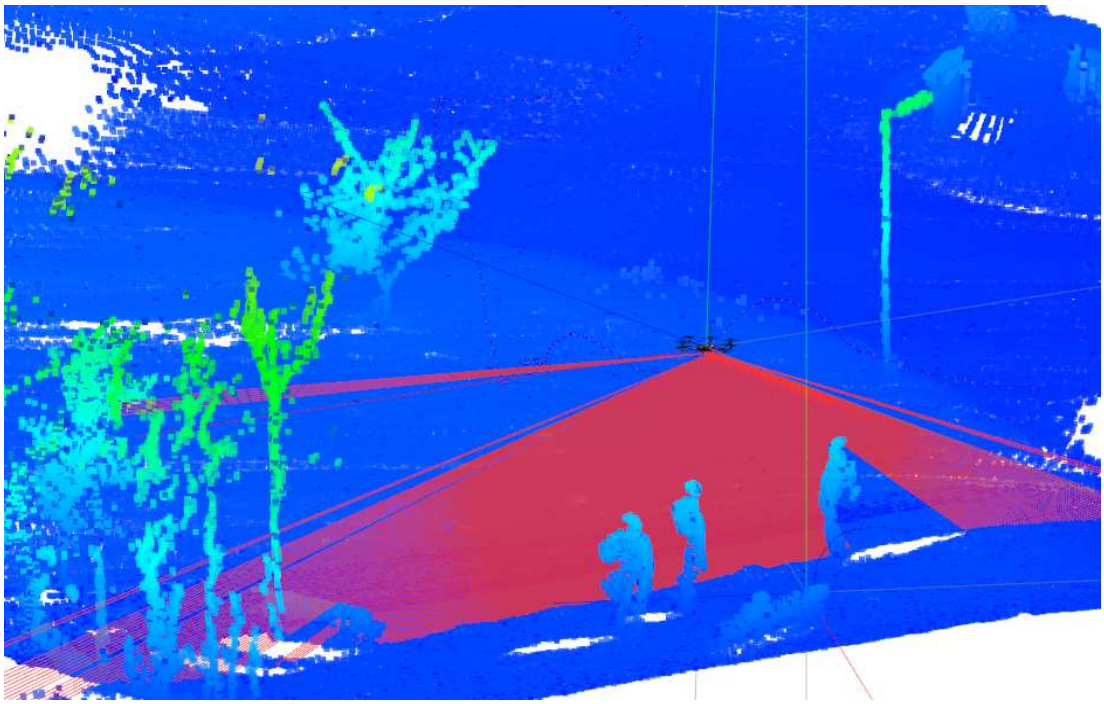


Figure 5.1: Close-up of the UAV scanning nearby persons on the campus. A single scan is shown, visualizing every ray that caused a reflection in red.

planning from traversing the cell. Other scanners (e.g. the Velodyne HDL32e) are more sensitive against false positives, which would then have to be removed in software.

First experiments with regard to flight stability and safety started during construction of the platform, before more expensive components (LIDAR sensors and navigation system) were mounted. About 20 remote-controlled flights with a duration of up to 15 minutes confirmed that the UAV is able to handle payloads of up to 1kg.

Testing the high-level motion controller required a working self-localization, and thus was done after the navigation system was operational. Over 50 flights were necessary to tune the navigation system for suitable output of position and orientation, tune the gains of the PID controllers and implement the possibility to switch from autonomous to user-controlled operation during flight (as required by German law). For some of these flights, the UAV was attached to a long fishing rod held by an assistant, effectively preventing crashes in case of controller failure. Finally, the UAV is able to navigate reliably along specified waypoints, while at the same time accommodating the inertial navigation system's special requirements with regard to flight dynamics. The motion controller is known to handle wind gusts of up to 10.8 m/s (one of the videos shows a flight on October 1st, 2013, when the weather station of the nearby Hamburg airport recorded temperatures of 12°C and an average wind of 4.2m/s with gusts up to 10.8m/s).

5.2 Localization Reliability

Evaluating reliability of the platform's localization requires the definition of an application-oriented metric. The previously postulated goals do not require point clouds with sub-decimeter accuracy, but the system's flight-safety strongly depends on a working collision avoidance, which relies on a constantly updated occupancy grid. This grid is updated from the incrementally growing point cloud, which in turn is generated by fusing the system's position and orientation with range readings acquired by the LIDAR sensors. Thus, flight-safety depends on the availability of localization.

Currently, the system relies on the GNSS/Inertial navigation system to provide position and attitude solutions, which requires reception of signals on different frequencies from a sufficient number of satellites. During startup, achieving centimeter-precise positioning requires at least four GPS satellites to be visible, with each supplying signals on L1 and L2 frequencies for both coarse acquisition and carrier phase measurements. This works reliably under open-sky conditions. Once the carrier phase ambiguities have been solved (see section 3.1.3 C for an explanation of the process), *RTK fixed* mode is entered and GLONASS satellites are added to the solution, making it more robust against loss-of-lock of single satellites. Because of limitations in the firmware, the GNSS receiver is unable to *start* centimeter-precision positioning using signals from mixed constellations, e.g. 3 GPS and 3 GLONASS satellites. Once the solution of position, velocity and time is supported by both GPS and GLONASS satellites, we experienced very robust positioning even close to buildings (about 1m distance), indicating very reliable multipath mitigation.

Of course, loss of GNSS-based PVT is not unseen: combinations of few visible satellites (because of unfortunate orbit parameters and the comparatively high latitude in Hamburg), electromagnetic interference experienced with previous hardware designs and temporary obstructions, e.g. caused by flying below large trees can prove troublesome. The first problem is mitigated by supporting multiple constellations and the second by shielding - only the third problem was only partly solved by using a dilated occupancy grid to keep a minimum distance from potentially shadowing structures. Because the IMU is unable to support precise positioning in case of GNSS outages for more than 5 seconds, the base station notifies the pilot in this case, requesting him to override autonomous motion control.

As discussed in section 3.1.3 B, measurement characteristics of MEMS-based accelerometers and gyroscopes are inferior compared to more expensive, higher-grade IMUs. With the IMU being very lightweight and mounted on a vibrating platform, it follows that the produced values are accordingly noisy: diagram 5.2(a) shows angular velocities during a manual 4-minute flight of the UAV as an example. This flight was conducted with the UAV configured as depicted in figure 3.4. While this overview doesn't lend itself well to detailed analysis of the flight, it demonstrates the expected increase in amplitude after startup and also shows spikes at the end, induced by landing and touchdown. It

can also be seen that rotations around roll and pitch axes are usually small in amplitude (less than 25°/s), but quite frequent. As it is common for manually controlled multirotor flights, rotations around the yaw axis are comparably rare, because they are mostly commanded to re-align the orientation of the UAV with that of the pilot on the ground. The graph demonstrates in surprising clarity how this maneuver is executed roughly every 20 to 30 seconds, when the orientation offset has grown to a point where the pilot is unable to transform roll and pitch commands issued on the remote control's coordinate system to the motions that will result untransformed on the UAV.

Graph 5.2(b) shows the angular velocities during different phases of the flight: clearly visible are the noisy readings while the motors are not moving and the UAV is parked (green section). As expected, the vibrations of motors and propellers spinning up are picked up by the gyroscopes, the result of which can be seen in the yellow section. During the time-window marked in white, the UAV's motors are idling at comparably low speed. Yet, the values of the roll-gyro already show a considerable drift towards a negative rotation. When the vehicle finally takes off in the red section, the amplitude of the vibrations appear to grow with the rotational velocity of the propellers.

Graph 5.2(c) presents all of the gyroscope's readings during one second (100 seconds into the flight). Since there are no comparative measurements made with an IMU supporting a higher measurement-bandwidth, it is not possible to determine whether the values represent the UAV's true motion, or to what extent they include image frequencies appearing due to undersampling of the original signal.

Finally, the data indicates that rotations during flight (and even touchdown, depicted in figure 5.2(d)) are well within the gyros' input ranges Xsens Technologies B.V. (2009).

Diagram 5.3(a) presents the accelerations sensed during the same flight. Since the IMU's Z axis is aligned with gravity, the values from the respective accelerometer are offset by $9.81 m/s^2$. Analogous to the visualization of the gyroscope data above, this overview exhibits the expected increase in noise after startup as well as a sudden spike at touchdown.

Interestingly, the graph over the complete flight shows much more variation (presumably vibrations) in accelerations on the Z axis. Generally, vibrations were expected to be more prevalent in the plane parallel to that of the propellers (i.e., X and Y axes), since their lack of balancing was assumed to be the main cause of unwanted oscillations. The observations are best explained after taking a closer look at the vibration dampening mechanism pictured in figure 3.4(a): the rubber shock mounts are installed vertically as standoff fasteners, connecting the sensor-module (composed of IMU, battery and both laser scanners) with the UAV's frame above it. This type of shock mount absorbs vibrations much better in lateral directions than it absorbs energy along its primary axis.

5.3(b) also yields interesting results: during motor-spinup, vibrations appear predomi-

nantly in the X/Y-plane of the vehicle, while vibrations on the Z axis remain negligible. Once the propellers rotate at their idling velocity, the amplitudes decrease. This indicates that the distribution of vibration energy to different directions highly depends on the rotational velocities of the motor/propeller setups.

The traditional approach to vibration damping is to analyze the unwanted oscillations in the frequency domain, then design decoupling mechanisms optimized for the dominant frequencies, the respective amplitudes and the weight of the object that is to be decoupled. This has worked rather well for e.g. single-rotor helicopters, which are commonly pitch-controlled and have their engine running at almost constant speeds. On multi-rotor helicopters however, attitude and thrust is controlled by varying the motor/propeller-velocities - as a result, the assumption of an almost-constant vibration pattern over time no longer holds, and the frequency-spectrum of vibrations experienced by the IMU widens. This is aggravated by resonances causing phenomena as described above, making mechanical decoupling both more important and more difficult at the same time.

In conclusion, the data implies that the dampening setup successfully decouples the sensor module from much of the vibrations passing through the vehicle's frame. On the other hand, it's analysis strongly recommends that further optimizations be applied to vibration dampening.

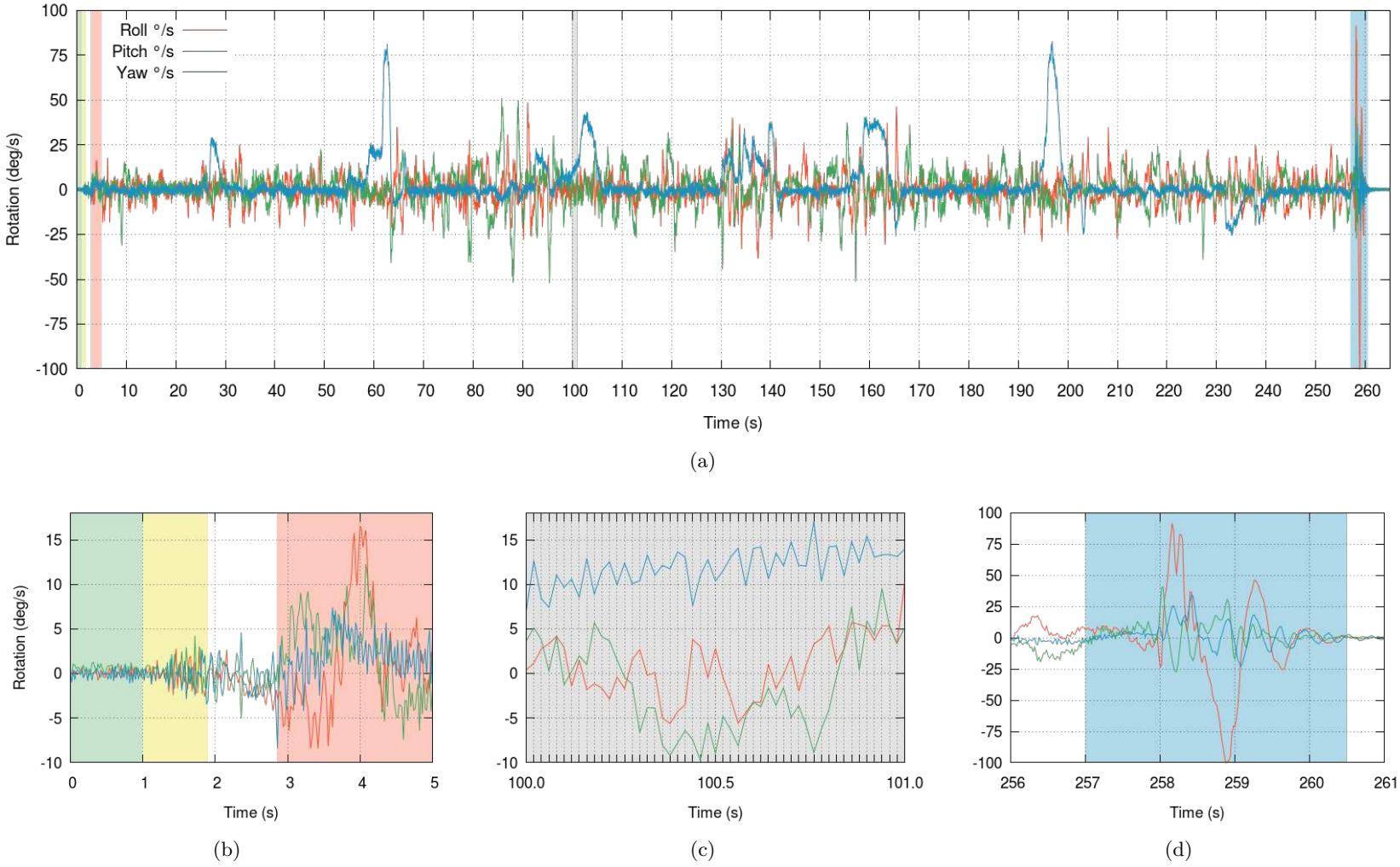


Figure 5.2: (a): Rotations experienced during a manual flight: (b) shows rotations experienced without motion (green), during motor-spinup (yellow), idling (white) and take-off (red). Subfigure (c) shows rotations in mid-flight in higher temporal resolution, (d) shows rotations while landing.

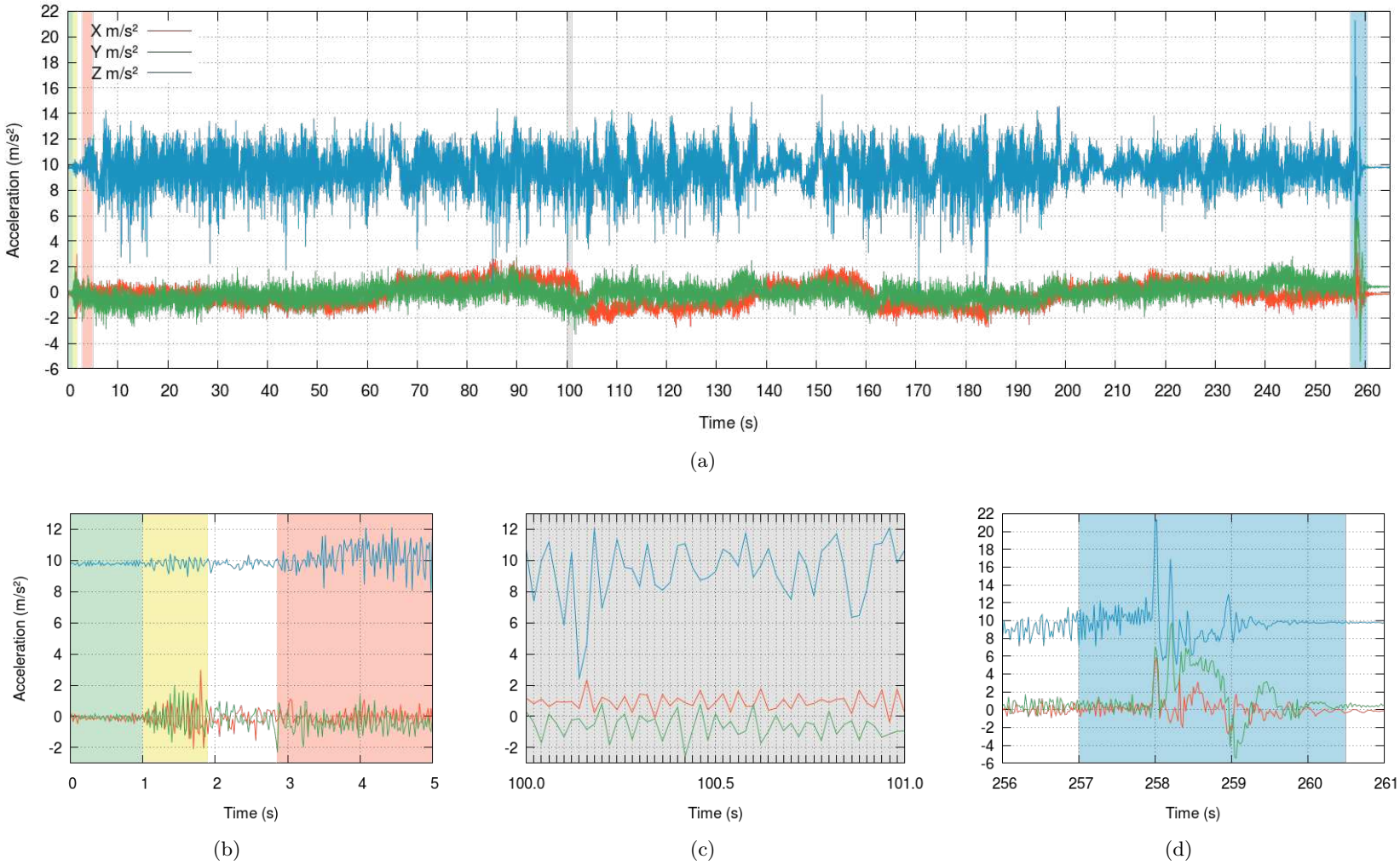


Figure 5.3: (a): Accelerations experienced during a manual flight: (b) shows accelerations experienced without motion (green), during motor-spinup (yellow), idling (white) and take-off (red). Subfigure (c) shows accelerations in mid-flight in higher temporal resolution, (d) shows accelerations while landing.

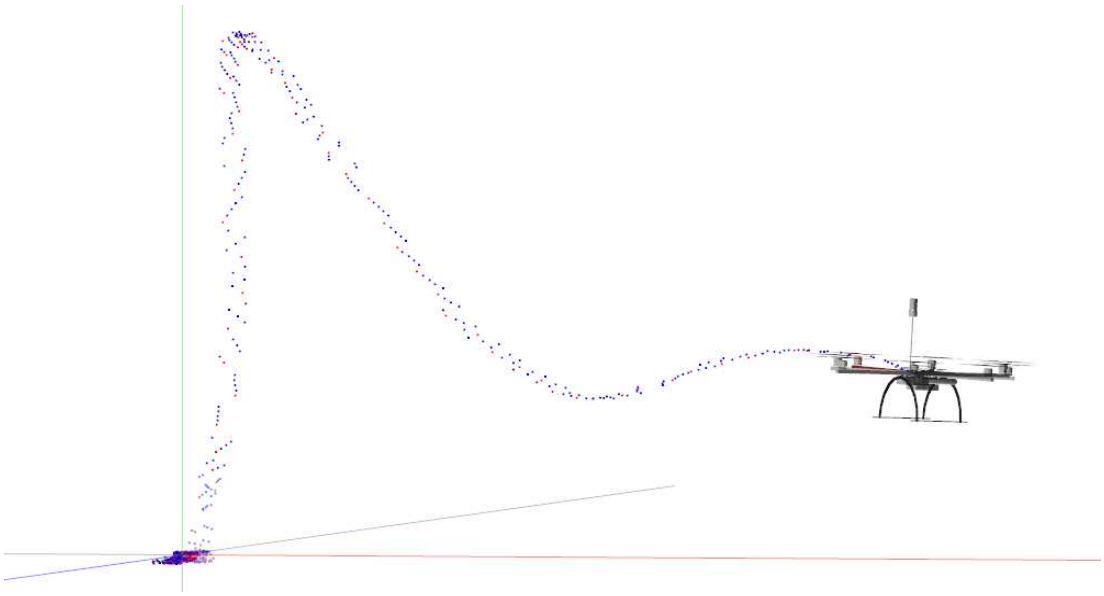


Figure 5.4: Visualization of the vehicle trajectory during lift-off, exhibiting considerable IMU drift and noise. Poses are computed at 50Hz. Poses in red are the result of fusion between GNSS data (10Hz) and IMU data (50Hz). They are interleaved by 4 consecutive poses generated by double-integrating the IMU readings (drawn in blue).

Because of the low-quality MEMS-accelerometers, vibrations and insufficient bias estimations, double-integration of accelerations between valid GNSS solutions (described in section 3.1.3 C) often leads to unusable integrated poses. This is shown in figure 5.4. As soon as a new GNSS-based position becomes available (100ms later), the drift introduced in the previous iterations can be corrected. However, the weighing of sensor information cannot be adjusted directly, which is unfortunate, as experience suggests that the filter regularly applies too much weight to IMU measurements. This behavior prevents an incoming GNSS-based position (which is precise to few centimeters) from completely correcting the drift that occurred in the preceding IMU-only iterations.

The accuracy of the point cloud produced has not been investigated thoroughly, but comparing distances measured in the point cloud with those measured manually have revealed errors of no more than 25 centimeters. While this may not be sufficiently accurate for many problems traditionally solved with ground-based surveying equipment, the data produced is precise enough to be used for motion planning, collision avoidance and some applications of surface reconstruction. Figure 5.1 gives an impression of the resulting cloud by showing the UAV scanning nearby persons in-flight. The aforementioned video shows the UAV scanning a street light twice within roughly a minute, where the lamp's mast shows up roughly 20cm away from the first scan in the point cloud. Depending on the strength of vibrations picked up by the IMU, kinematic alignment and even firmware revisions of the GNSS/Inertial navigation system, errors greater than 25cm have also been observed, caused especially by incorrect heading val-

ues. Since the GNSS/Inertial data fusion algorithms are proprietary and closed-source, no further investigations as to the cause of these problems could be made.

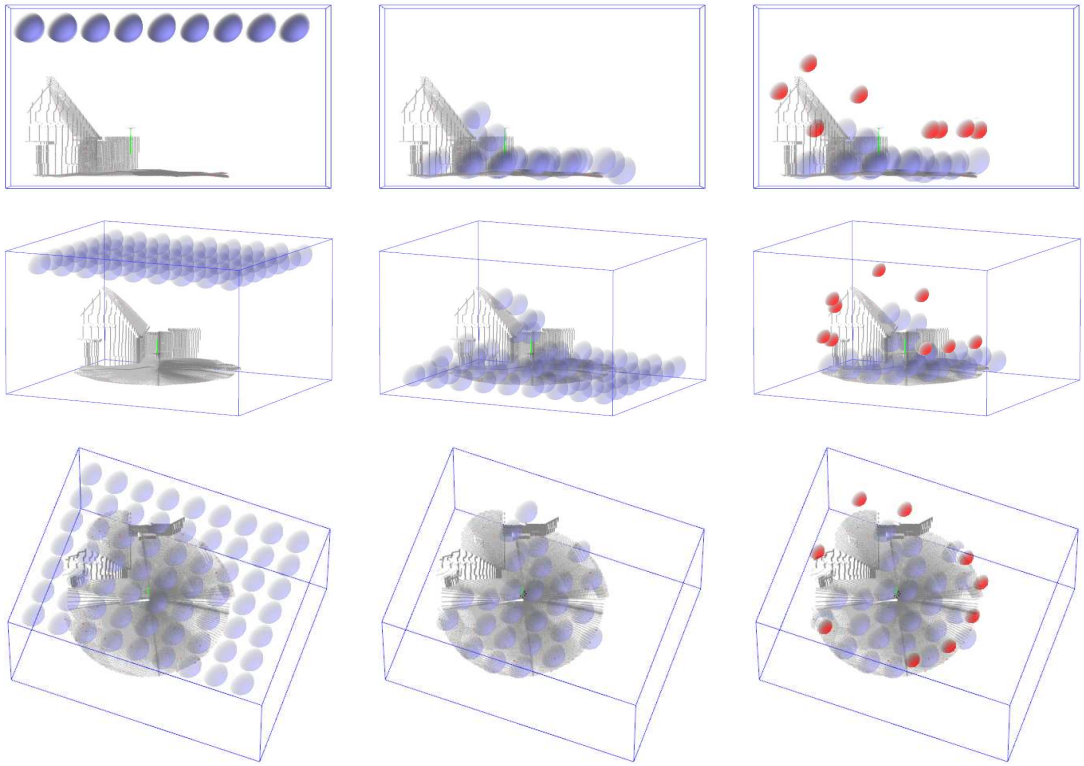
5.3 Generated waypoints and paths

To validate the idea of waypoint generation, the algorithm has been applied to various different sample point clouds. Before the system was capable of creating point clouds of outdoor environments, virtual environments were modeled in the simulator, and subsequent simulation yielded datasets that could be used for testing. Figure 5.5 shows waypoints being generated for an example point cloud: the data was captured by simply yawing the first hardware revision of the UAV by 180° in the simulator and is processed using the CPU-based approach described in Adler et al. (2012). Results pictured in figure 5.5, column c) show that *Next Best Views* are generated at safe locations and close to the frontier between known and unknown environment. The insufficient computational power of the CPU restricts real-time generation to few particles with large sizes, so the process is unlikely to detect gaps high in the house's wall. This reliance on particle count and size is why we denote the approach to be *resolution-complete*.

The complexity of the virtual environment was subsequently increased by adding more architecture (five houses) and vegetation (seven different kinds of trees) to form a small village. To introduce more variation into the simulated environment, the position and orientation of individual models was largely randomized on simulator startup, effectively creating many different test environments. Furthermore, each simulation did not generate waypoints from a single point cloud. Instead, waypoint generation was restarted with the current point cloud whenever all previously generated waypoints had been passed.

Figure 5.6 shows an example of a mapping mission executed in the simulator, still using the CPU-based process. The sequence clearly shows how the UAV follows each iteration's set of waypoints, growing the map inside the bounding box. Once no further gaps can be found, the particle size is decreased and the process restarted in figure 5.6 e). Reordering waypoints into the shortest path often yields ring-like trajectories (seen in Subfigures a) to c)), which can be followed by both multi-rotor or even fixed-wing aircraft in an energy-efficient manner.

To assess the path's mapping-efficiency, the number of points stored in the point cloud for a given flight-time was established as the primary metric. In the CPU-based version, an octree-implementation allows setting a maximum point-density. For the point cloud storing the surface-reconstruction data, the minimum distance between neighboring points was set to $0.1m$, so that measuring the number of points stored over time is equivalent to measuring the scanned surface area over time. Tests were executed in a simulator, with the platform's linear velocity limited to $2.8m/s$ during all trials. While the scanned points were streamed to the base station, waypoints were generated and



(a) (b) (c)

Figure 5.5: CPU-based Next Best View generation, processing data from simulation. The three columns show a static point cloud from three different viewpoints, being subjected to simulated particles in order to find gaps. a) depicts the initial setup: the bounding volume and sample geometry are drawn in blue, while the point cloud is drawn in grey. b) shows sampling geometry interacting with the point cloud. c) shows how the last collision positions of detection spheres are converted to waypoint candidates after having reached the bounding volume's bottom plane.

statistics about flight-time and point cloud sizes were logged.

Graph 5.7 shows results of using a simple, manually planned and collision-free scanline-based exploration with trajectories similar to those proposed in Xu et al. (2011) for optimal complete terrain coverage. The maps resulting from scanning along these trajectories can be seen in Fig. 5.8(a) and 5.8(b) for four and six equidistant passes, respectively. As expected, flying more scanlines in the same area results in more points scanned, albeit with a slightly slower rate.

In comparison, the graph's red line shows the efficiency of mapping when using waypoints generated using particle simulation: because the point cloud needs an initial population for our algorithm to start, the scanning rate stagnates shortly after launch during the first phase of waypoint generation, then reaches comparable speeds as the vehicle starts passing waypoints. In this phase of flight, the automatically generated trajectory is as efficient as a manually planned, collision-free optimal path, with the

obvious advantage of having been generated automatically. Compared to exploring in swath-based fashion, the true strength in this method lies in the fact that on-line re-planning occurs such that when no more waypoints can be generated, gradually smaller sampling geometry is used to improve on ever smaller deficiencies in the reconstructed surface.

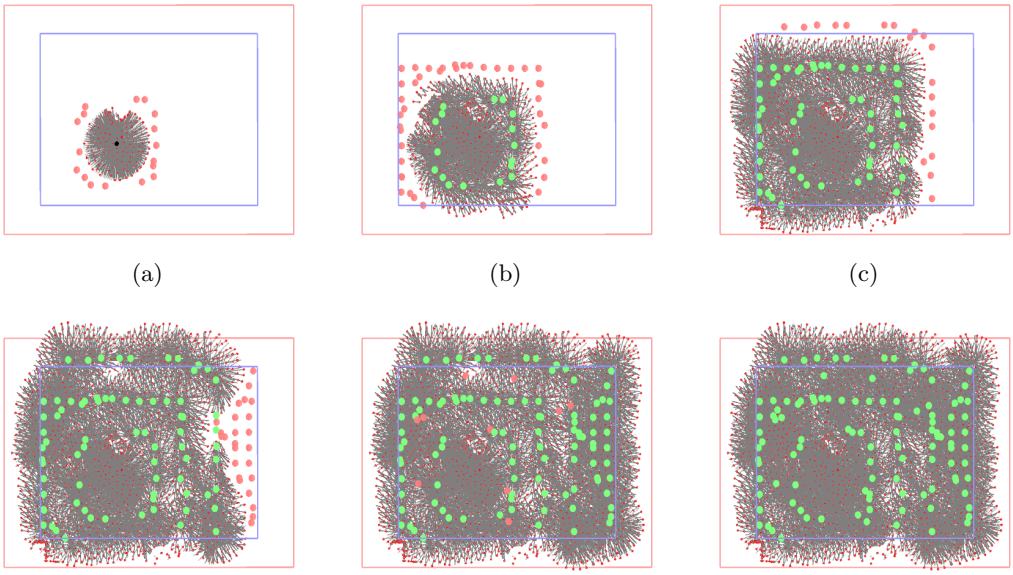


Figure 5.6: A simulated mapping mission seen from the top, showing how multiple iterations of waypoint generation lead to continuous scanning at the border between known and unknown environment. Red waypoints are enqueued for scanning, green waypoints have been passed. In figures a) to d), sample spheres with $r = 3m$ have been used to quickly explore the landscape. In figure e), sampling geometry of reduced size ($r = 1.5m$) has allowed finding smaller holes in previously scanned surfaces. Figure f) shows all visited waypoints and the resulting coverage.

Due to constraints imposed by German regulations and insurance conditions, real-world flights were confined to the campus of the Department of Computer Science at the University of Hamburg. Nevertheless, the campus features a wide variety of architecture (building sizes range from sheds to six-story buildings) and vegetation (plain fields, low vegetation and high trees). The robustness of motion control and localization was tested during more than 50 flights. The suitability of generated waypoints was tested extensively in simulation, then confirmed during more than 10 flights. The successful navigation of generated waypoints is also presented in both aforementioned videos.

Further results are presented using an example data set that was generated during a real flight over the campus. Both the campus and the resulting point cloud (colored according to height) are displayed in figure 5.10. After about 100 seconds of flight within a bounding box of 64*64*32 meters, motion planning is executed: Figure 5.11(a)

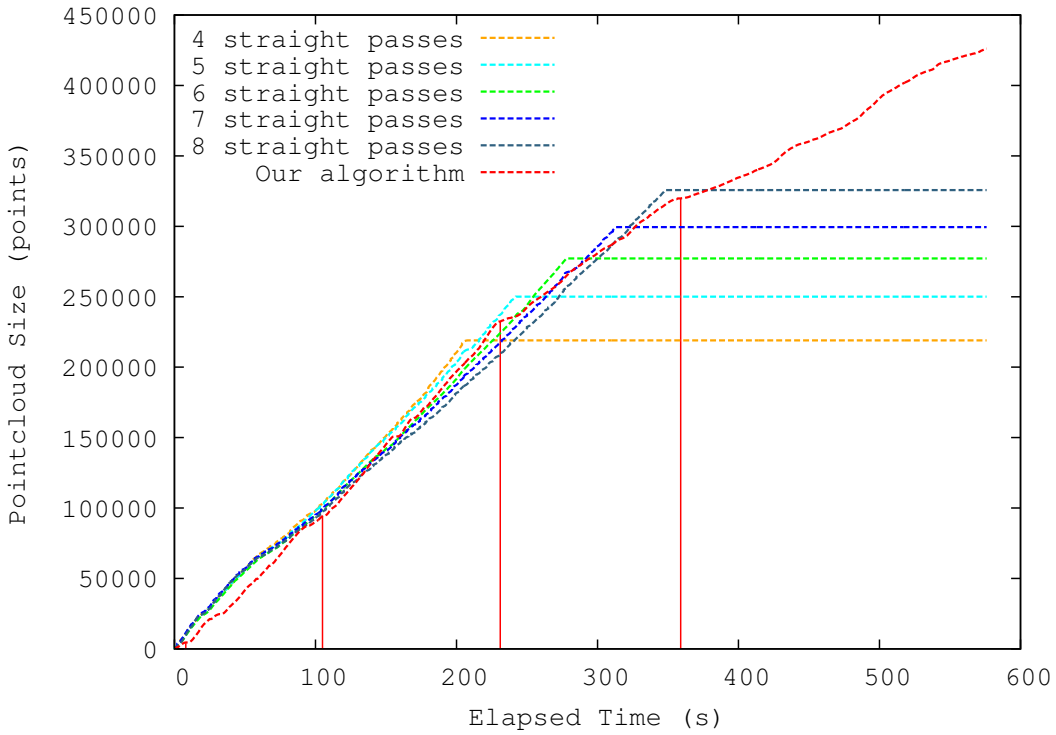
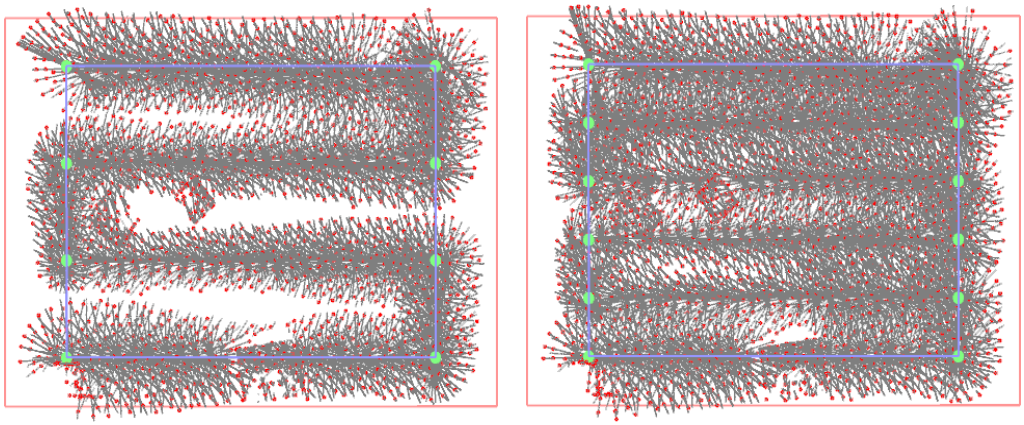


Figure 5.7: Point cloud size vs. flight-time using several waypoint generators. Vertical lines are drawn on every round of waypoint generation using our algorithm, executed after all previously generated waypoints have been passed.

shows 32k particles of 0.5m radius being poured over 36k colliders of the downsampled point cloud. After five seconds, the particle simulation is stopped and the resulting values in the grid of information gain are presented in figure 5.11(b). One hundred cells containing the highest information gain are converted into waypoint candidates with 3D world-positions. Candidates closer than 5.0m are merged and passed to the path planner.

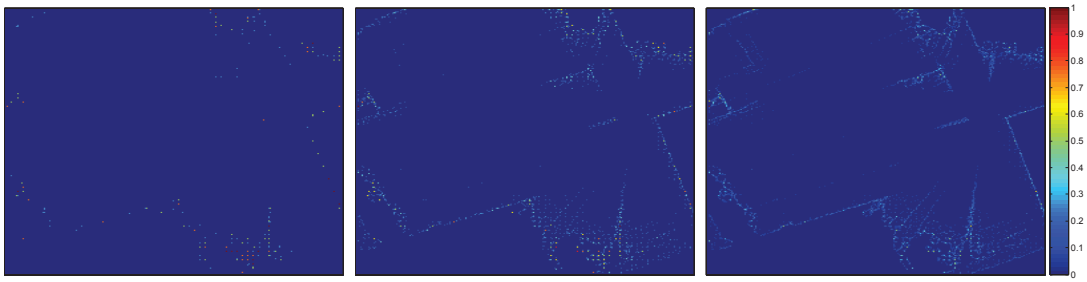
Next, the path planner builds a 3D occupancy grid and dilates it as described in section 4.3 and depicted in figure 5.12(a). A collision-free path from start to goal cell is found and passed to the rover program in the form of waypoints, (shown in figure 5.12(b)). Using an NVIDIA GeForce GTX 670, processing the grid of information gain to a set of ordered waypoints takes between 500ms and 600ms (using an occupancy grid of 32^3 cells). While using a finer grid is possible, the UAV's diameter of more than 1m (including propellers) make a cell-size of 2^3m seem sensible.

The path planner is based on a simple grid-based search-algorithm, as the underlying parallel architecture makes approaches like best-first search (like e.g. A*) have less impact on performance, especially when the number of grid cells to search in each iteration does not exceed the number of available multiprocessors. As most geometry-based algorithms, the planner is resolution-complete, meaning that if a path exists, it will be



(a) 4 passes (b) 6 passes

Figure 5.8: Point clouds resulting from straight scanline passes. Note the unscanned surface in the bottom center shadowed by a roof.

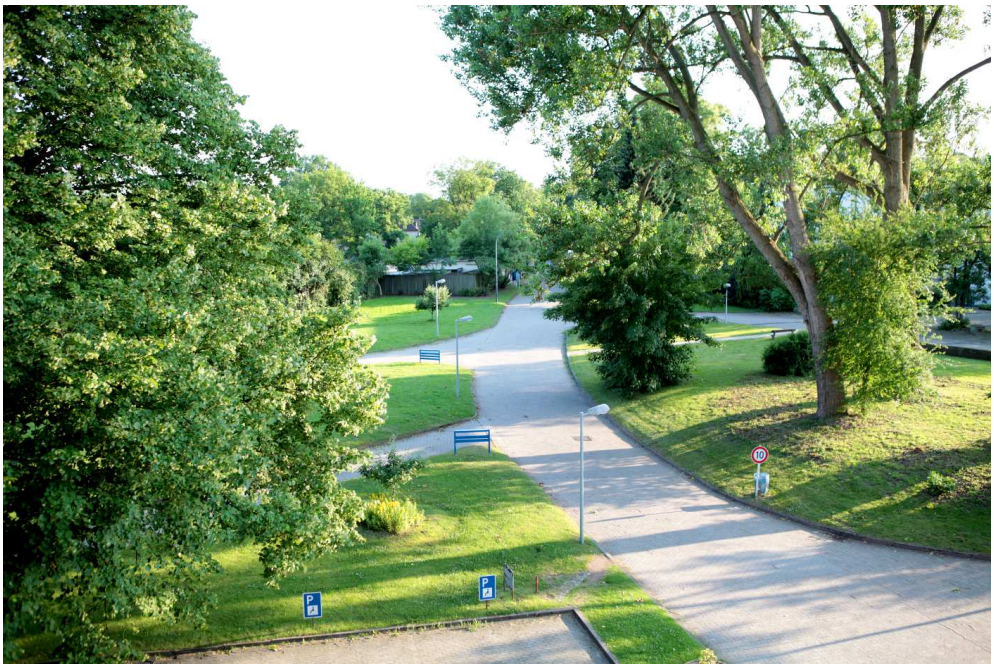


(a) (b) (c)

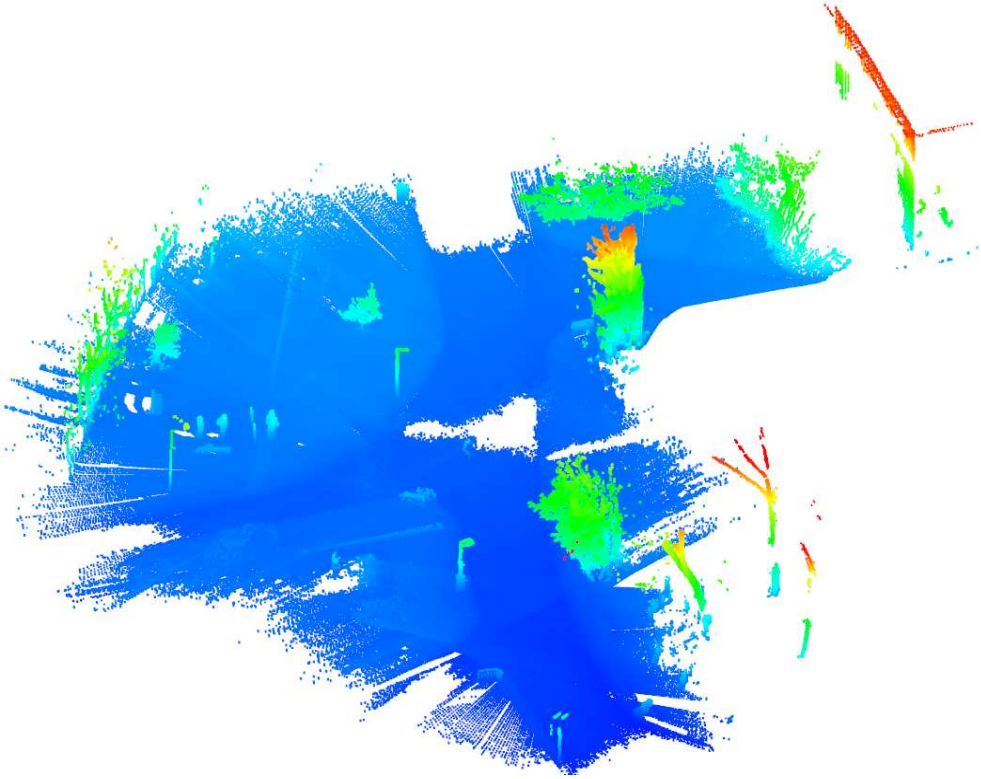
Figure 5.9: Grid of information gain with $256 * 32 * 256$ cells, projected as in figure 4.9(f). Maximum information gain is shown normalized using a heat map visualization a) after 400, b) after 700 and c) after 1000 simulation steps.

found given the grid's resolution is sufficiently fine. Because path-planning exploits the grid-cell's size to introduce a safety margin, false negatives are theoretically possible, but practically result from the compromise between reachability and safety.

Because the particle simulation causes the information gain in G_{IG}'s cells to steadily increase during simulation, termination criteria are non-apparent: a trade-off must be found between short runtimes for rapid generation of results and longer runtimes that allow a better sampling of the point cloud's gaps. Figure 5.9 presents the *normalized* information gain values of the point cloud depicted in figure 4.9(f) at different times into the particle simulation. While a visible difference exists between the information gain values at 400 and 700 steps into the simulation (a) and b)), the normalized information gain remains almost constant during the following simulation steps (between b) and c)). Thus, for the targeted application on UAVs, using a bounding box b with a size of $64^3 m$, 16k particles ($p \in P$) with a radius of $0.25m$ and 64k colliders ($c \in C$) allows generation of multiple NBVs in less than $3s$ using a NVIDIA GTX 670 graphics card.

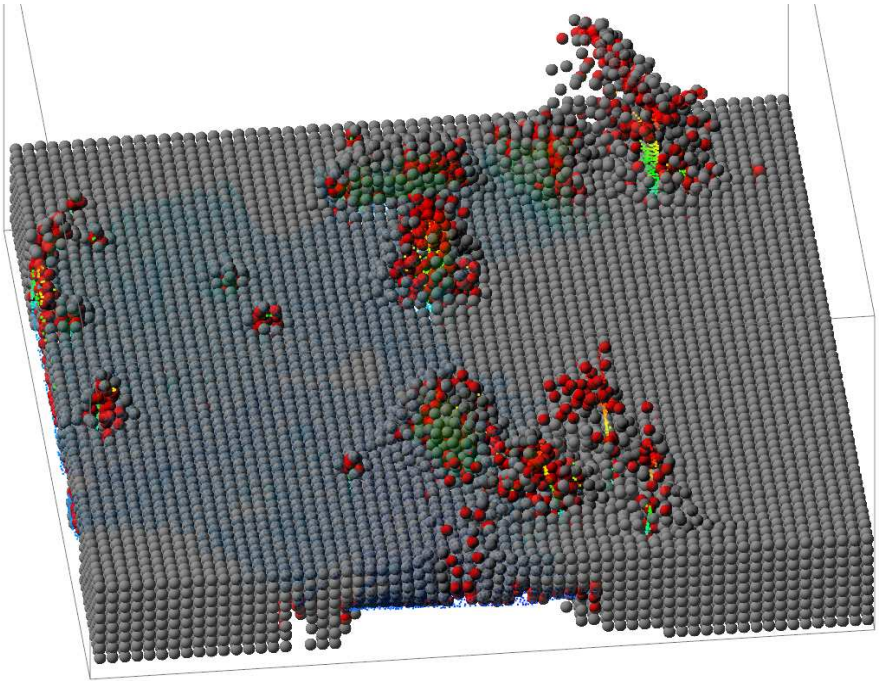


(a)

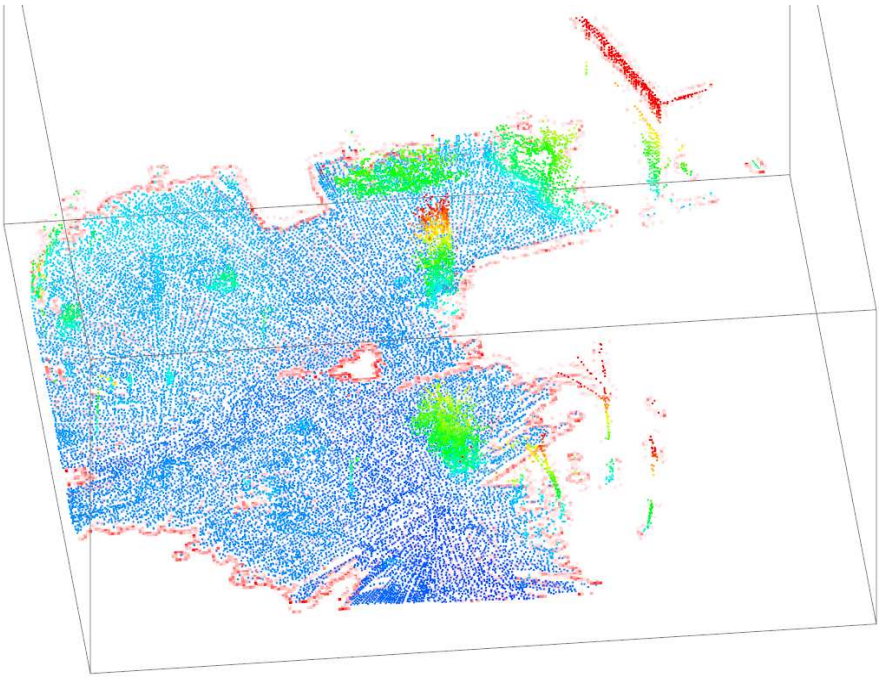


(b)

Figure 5.10: (a): Part of the campus of the Department of Computer Science at the University of Hamburg. (b) Point cloud resulting from about 100 seconds of flight on campus, aligned with the viewpoint of above photograph.

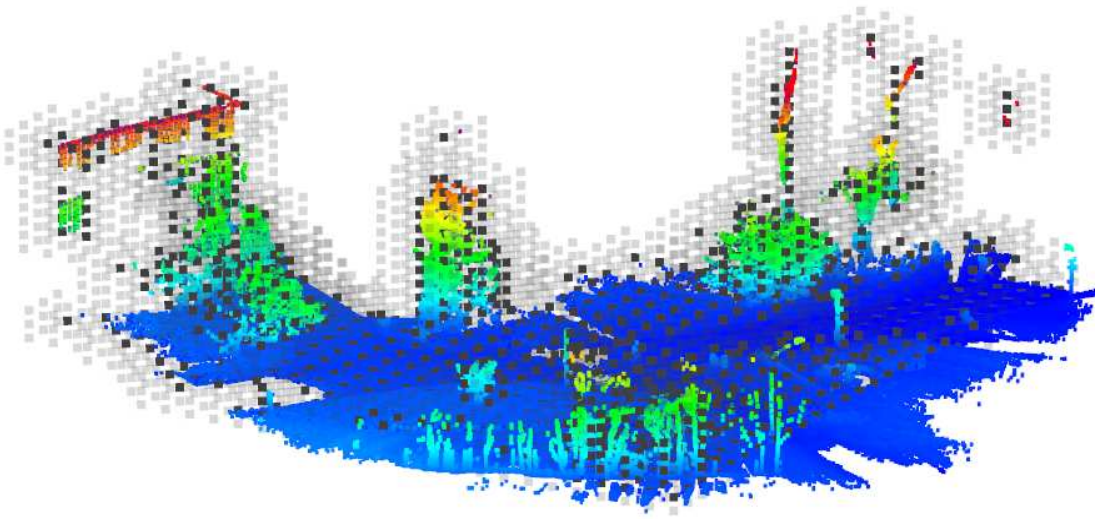


(a)

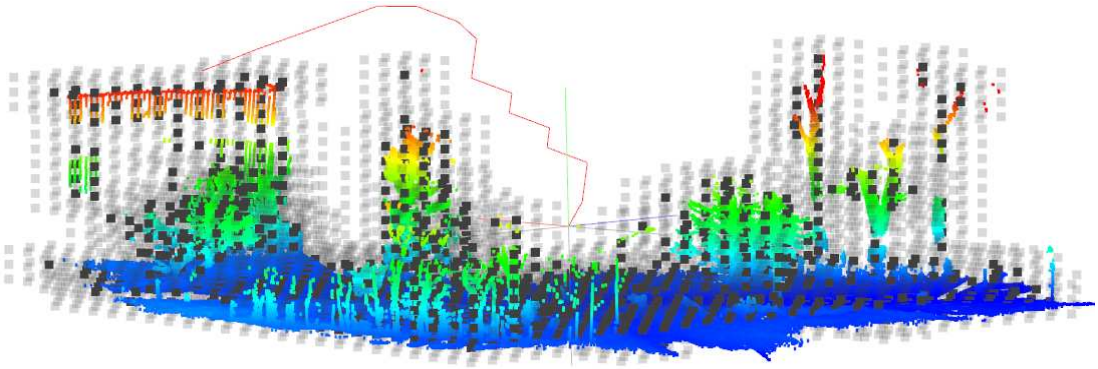


(b)

Figure 5.11: (a): GPU-based particle simulation in progress, colliding particles with sparse collider cloud to test the latter for watertightness. (b): Visualization of the grid of information gain. More saturated red indicates cells with higher information gain, which are more likely to be processed into waypoints.



(a)



(b)

Figure 5.12: (a): Shows the occupancy grid from the opposite side, with 32^3 cells superimposed over the dense point cloud. Occupied cells are dark gray, dilated cells in light gray. (b): Grid cells containing high information gain values were converted to waypoint candidates. Afterwards, close candidates were merged and a collision-free trajectory was generated from the vehicle's position (in the center) to the best candidate.

5.4 Success and Failure Analysis of Waypoint Generation, Path Planning and Motion Control

Testing point clouds for watertightness requires correctly tuned parameters. The ratio between point cloud density and particle size must be considered, which is discussed in section 4.2.3 and presented in figure 4.10. Since these parameters' interdependency can be expressed by a simple, closed-loop equation, they can be set appropriately in software after determining bounding-box b of the environment to be mapped, as well as the graphics card's memory capacity, memory bandwidth and GPU clock speed. The process also relies on sensible initial particle placement, meaning that the bounding box b must completely contain the point cloud to be tested when particles are placed at the top. Interior NBV problems have not been tested, but would require particles to be spawned within the point cloud, not above.

Difficult scenarios like gaps in high walls or below carports can remain undetected when a low number of particles prevents the simulation from filling the point cloud to a sufficient height or when the particle-size makes reaching gaps impossible. This can be rectified by initializing the simulation with large particles and restarting it with decreasing size when no waypoints can be generated. This is a common workflow for resolution-complete, probabilistic techniques.

Path-planning is performed using a classic occupancy grid. The UAV cannot be commanded to collide with geometry in occupied cells because the cell's sides are longer than the UAVs diameter, and occupied cells are dilated before path-planning is executed. However, collisions can occur in highly dynamic environments: since the occupancy grid is created from the pointcloud and checked for collisions every two seconds, fast-moving objects could block the UAVs path and cause an impact before collision avoidance has detected the condition. The combined field-of-view of the platform's laser range finders is depicted in figure 3.4(b). As can be seen, only parts of the environment can be sensed. Even though this disadvantage is largely compensated by the platform's own motion, small geometry can remain undetected. This problem was detected early in the design of the platform, and an attempt was made to improve the field-of-view using a single laserscanner (see figure 3.2(b)). Leveraging the UAVs unique maneuverability, a constant yawing motion during flight would have resulted in a FoV covering the full sphere. Unfortunately, the navigation system's Kalman filter turned out unable to produce accurate localization when exposed to this flight pattern. In conclusion, the UAV's collision avoidance must work with incomplete data, and thus, cannot guarantee a collision free flight.

The combination of the high-level motion controller's low update frequency (10Hz) and a communication latency of up to 40ms to the low-level flight-controller, navigating in tight spaces can become risky when flying at high velocities or strong wind gusts. For this reason, the maximum translational velocity was capped to 2m/s during experiments, which proved to be a good fit for the resulting point density at higher altitudes.

Optimizing towards a faster scanning procedure would thus require more aggressive maneuvering (see figure 3.5 for the consequences in energy consumption) and faster laser scanners.

5.5 Scalability of Waypoint Generation

This section analyses the runtime behavior of the presented waypoint generation approach in relation to the problem size. As explained in section 4.2.4, the computations used to create waypoints are performed entirely on the GPU. Thus, memory requirements on the graphics card are one upper bound on the problem size. Today, discrete graphics cards with less than 1GB of memory are hard to find, and while 2GB is the most common capacity, up to 12GB are available on high-end cards.

The amount of memory required on the graphics card is determined by

- the number of points in the collider cloud (N_C)
- the number of particles (N_P)
- the number of cells ($N_{G_{SHT}}$) in both spatial hash tables
- the number of cells for the global grid of information gain ($N_{G_{IG}}$)

Table 5.2 lists the respective data structures, data types and their size in more detail. Memory requirements for particles and colliders grow linearly with their number. However, increasing the resolution of the grid structures for particle/collider hash tables and information gain requires $O(r^3)$ space.

As an example, when exploring a bounding box with a volume of $64m^3$ and generating waypoints for a point cloud with 64k points using 128k particles, two spatial hash tables are used for particles and colliders. Each table hosts $N_{G_{SHT}} = 128 * 64 * 128$ cells, while the computed information gain is stored in a grid with $N_{G_{IG}} = 256 * 32 * 256$ cells. This setup results in a total memory requirement of about 26.5 Mb.

The previous section shows that memory requirements are almost negligible for modern hardware.

In practice, the problem size is limited by the algorithms runtimes, and this condition is aggravated by real-time requirements. Figure 5.13 shows the time required for *one* iteration of particle simulation in relation to problem size and underlying hardware.

Generation of waypoints from the grid of information gain, updating the occupancy grid for collision avoidance and using it to plan paths is only done once every couple seconds and takes less than one second. Particle simulation, however, is executed many thousand times per flight, so the following analysis of runtime behavior shall focus on this phase exclusively.

Table 5.2: Memory allocated in GPU memory space

Data	Type	Size	Count
P_{vel}	float4	16 bytes	N_P
P_{col}	float4	16 bytes	N_P
P_{pos}	float4	16 bytes	N_P
P_{index}	uint	4 bytes	N_P
$P_{cellhash}$	uint	4 bytes	N_P
$CellFirst_P$	uint	4 bytes	$N_{G_{SHT}}$
$CellLast_P$	uint	4 bytes	$N_{G_{SHT}}$
C_{pos}	float4	16 bytes	N_C
C_{index}	uint	4 bytes	N_C
$C_{cellhash}$	uint	4 bytes	N_C
$CellFirst_C$	uint	4 bytes	$N_{G_{SHT}}$
$CellLast_C$	uint	4 bytes	$N_{G_{SHT}}$
G_{IG}	char	1 byte	$N_{G_{IG}}$

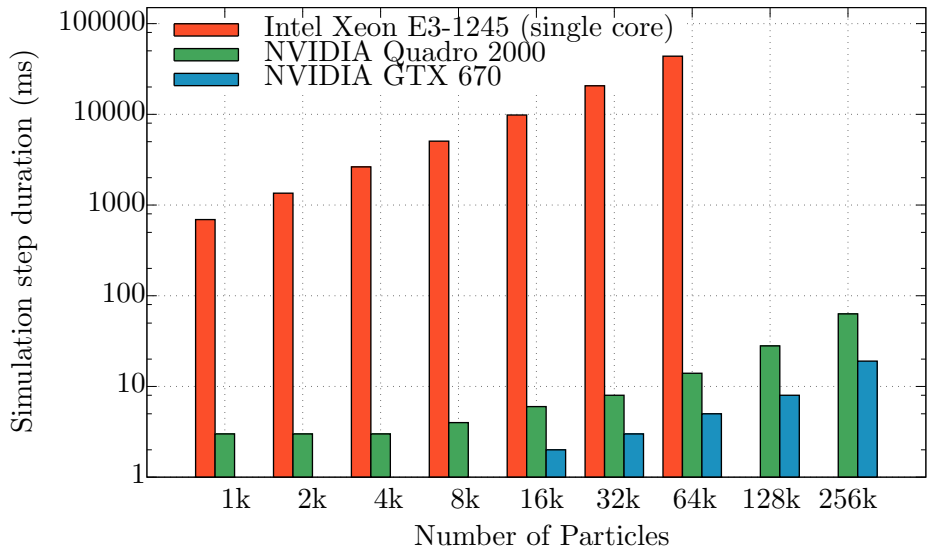


Figure 5.13: This graph shows the *maximum* time required for a single time step of the simulation containing 16k colliders against different numbers of particles. Visualization was disabled in all tests, note the logarithmic scale of the ordinate axis. The collision stage took 89.1% of the shader-processor's time, particle-system integration only 0.6%. The runtime of the proof-of-concept implementation for the CPU is up to three orders of magnitude longer than those of the GPU-based implementations.

Adler et al. (2012) describes the CPU-based implementation for particle simulation and also provides quantitative results by comparing the scanned surface over time using the algorithm against results obtained using scan-line passes. The CPU-based implementation was tested on an Intel Xeon E3-1245 CPU clocked at 3.30 GHz. It is an unoptimized, single-threaded version which delegates collision detection and handling to the Bullet Physics library. Afterwards, it queries for collisions that occurred during processing in order to manage a structure similar to P_{col}. Activating visualization causes further slowdowns, as usage of OpenGL immediate mode requires that all geometry is re-uploaded to the device for every frame. As shown in figure 5.13, testing a point cloud with 10k points for watertightness using 1k to 64k particles (without visualization) required between 0.7 and 44 seconds for *each* simulation step.

The GPU implementation was tested on a NVIDIA Quadro 2000 graphics card with 192 CUDA fermi-cores clocked at 625 MHz as well as a NVIDIA GTX 670 card, providing 1344 CUDA Kepler-cores running at a clock speed of 980 MHz. It is up to three orders of magnitude faster, taking between 2 and 12 ms. This is because integration of motion, collision detection and handling as well as point cloud visualization using OpenGL core profile are very suitable for processing on Single Instruction Multiple Data (SIMD) architectures.

Figure 5.14 presents the runtime of the particle simulation algorithm described in section 4.2.4, as captured by NVIDIA's Visual Profiler.

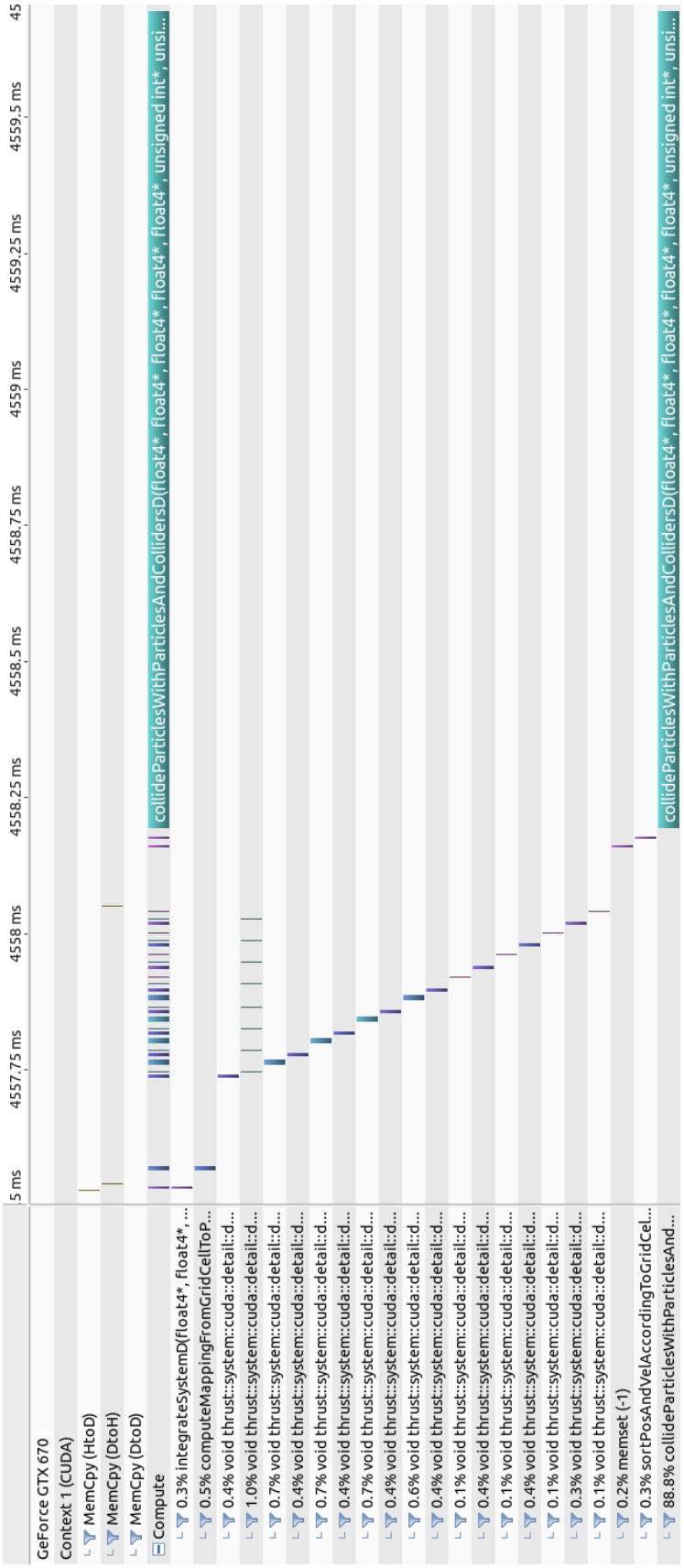


Figure 5.14: A timeline showing a single iteration of the particle simulation on a NVIDIA GeForce 670 GPU, generated by NVIDIA Visual Profiler. The kernels are ordered from top to bottom in order of execution and can be mapped intuitively to the steps of the algorithm presented in section 4.2.4. Data was captured using an unoptimized debug build, so absolute runtimes are not comparable to those stated in figure 5.13. Instead, the timeline shall give an overview of the relative kernel runtimes. The graph strongly suggests that future optimization focus on collision-detection and -handling as well as sorting.

Below the "Compute"-entry, every row of the timeline represents a specific kernel running on the GPU. Gaps between kernel launches are caused by host-initiated GPU management operations: mostly, these are memory (un)allocations on the device or querying the device's capabilities for optimum algorithm- and gridsize-selection.

The first kernel, *integrateSystem*, is described in section 4.2.4, Algorithm 2, lines 27-40. Processing of every particle yields a simple O(n) time complexity. Memory load and store operations always refer to the particle's indices and are executed synchronously by all running kernels. Thus, memory access is coalesced, allowing the kernel launch to finish in far less than one percent of the whole iteration's runtime. Scattered load/stores are only caused when particles hit the bounding box's floor, which is comparably rare.

Kernel *computeMappingFromGridcellToParticle* (defined in Algorithm 1, lines 2-5) translates to a coalesced reading of particles and two coalesced writes, explaining the time-efficiency of O(n) in similar manner.

Radix sort (invoked in line 6) is delegated to the thrust library's implementation, which splits up the workload into multiple kernel launches, accounting for all rows labeled *thrust::system::cuda::detail::...*. Adding up the single radix kernel's runtimes does not indicate long execution times. However, the preceding setup-time (shown as a long gap before the first launch of a thrust-kernel) is spent querying the GPUs capabilities and allocating memory. After sorting, almost the same amount of time is spent freeing the previously allocated memory. Performing these steps in every iteration of the simulation is not conceptually required and incurs a rather large runtime cost. As is visible in the "Compute"-row, memory allocations are also interleaved between sorting-kernel launches, so that sorting effectively does require a bulk amount of time and is the second best candidate for optimization.

The task handled by the kernel carrying the unwieldy name *sortParticlePosAndVelAccordingToGridCellAndFillCellStartAndEndArrays* implements Algorithm 1, lines 32-42. It uses shared memory infrastructure to reduce originally scattered into coalesced memory accesses. Doing so, the kernel finishes in negligible O(n) time.

The last kernel in the iteration is responsible for detecting and handling collisions. Although collision-detection between particles/particles and particles/colliders can become $O(n^2)$ in the worst case, this phase scales far better than that in practice. At first, all threads perform a quick, coalesced read of the respective particle's position. Next, threads must query 26 cells in neighboring 3D-space, which leads to as many reads of the previously generated lookup tables. Here, optimized memory locality is of absolute importance, because scattered accesses often cause cache misses, incurring high latencies from DRAM read operations. Worse, if particles are found in those cells, their position must be read, translating to further potential cache misses. Because of limitations in the multi-processors' thread-schedulers, the smallest logical group of threads (called a *warp*) always finishes when its last thread has finished. While this

did not present a bottleneck in other kernels that feature a fixed control flow, collision detection forces threads to iterate over a variable number of neighbors; some thread's particles will have many neighboring particles and colliders, while others will have none at all. Thus, a warp of threads must always wait for the thread that must handle the most particle-neighbors and thus, takes the longest amount of time to complete. This symptom is called *warp divergence* and could only be solved by pre-sorting the particles according to the number of collisions. However, this would counteract the pre-sorting of particles implemented in order to optimize memory locality. Global memory load latency is documented to lie between 400 and 600 cycles (tested in Wong et al. (2010)), so memory locality is a far more important optimization target than warp divergence.

In conclusion, the profiling results in figure 5.14 impressively demonstrates that the particle/particle and particle/collider collisions are by far the most computationally demanding stages, requiring 90% of the time spent on the streaming multiprocessors; Waypoint generation is limited by the speed of collision detection.

To improve scalability, collision-detection, waypoint generation, path planning and obstacle avoidance have been designed to work within the constraints of a bounding box b. When environments become too large to be processed on the GPU, b can be reduced, effectively causing the UAV to map only a part of that environment. When done, b is shifted to yet unexplored terrain once no more gaps can be found. Currently, the user can either shift b manually in the base station's user-interface, or instruct the path planner to automatically center b at UAV's current position when it approaches its boundaries.

Chapter 6

Conclusions and Outlook

This dissertation has researched autonomous airborne mapping and presents a solution that actually works in real, unstructured outdoor environments.

Extensive testing in chapter 5 has shown that the system features a wide environmental envelope and is capable of producing georeferenced maps in real-time.

In conclusion, the research questions listed in section 1.2 (page 8) can be answered as follows:

- A light-weight and maneuverable UAV was designed and constructed. A wide range of sensors was mounted, electrically connected and successfully integrated into the software architecture (presented in chapter 3).
- A novel algorithm was devised, finding multiple next-best-views for unorganized point clouds. A proof-of-concept showed the algorithm's applicability, and experiments with a GPU-based implementation (see section 4.2) proved it to be useful (experimental results are shown in chapter 5).
- Collision avoidance allows the platform to safely travel through the air, updating the planned path in case of upcoming collisions (see section 4.3).
- The problem of high-level motion control was solved, enabling safe flights even in the presence of moderate winds, and adhering to local safety regulations (see section 4.4).

Finding these answers has allowed the experimental platform to fulfill all five goals set on page 6.

This thesis has contributed to the state of the art by presenting

- the first unmanned aerial vehicle that streams an accurate point cloud of an outdoor environment to a ground station in real time for further processing

- an approach for waypoint generation through particle simulation, which has enabled efficient airborne exploration
- a system that tightly integrates a vehicle with a wide range of sensors and a custom software stack, successfully performing autonomous, truly three-dimensional mapping of outdoor environments

In the last five years, the amount of UAV-related robotics research has increased up to the point that the most important robotics conferences (e.g. IROS, ICRA) have added additional tracks for presentations of aerial robotics research. Most of this research focuses on motion control and self-localization and mapping (SLAM) in indoor environments, and has brought to bear great advances in these scenarios. By combining a small-scale UAV with LIDAR and navigation sensors to autonomously map *outdoor* environments, this thesis researched an application that was far less popular.
However, this popularity has starting catching up in the more recent past; indeed, many major manufacturers of geodetic equipment (e.g. Leica, Trimble, Javad) are currently working on their own LIDAR-based UAVs for mapping of outdoor environments.

6.1 Limitations and open questions

Even though the results presented in the previous chapters are very encouraging, a few issues remain for the system to become more robust, and thus, a viable option for airborne mapping in commercial contexts:

- Localization depends entirely on GNSS/Inertial navigation, constraining applicability to open-sky environments. As hinted in chapter 2, SLAM-based techniques are evolving rapidly, so that this limitation can be expected to fall within the next years. Although precise georeferencing would have to be sacrificed in SLAM-only systems, losing the cost and complexity of the GNSS infrastructure (including reference station) would greatly simplify the system.
- The accuracy of the generated point cloud can not be guaranteed. Depending on initialization of the system and vibrations from the UAV platform, heading inaccuracies of up to 20° have been observed. Fortunately, new GNSS/Inertial navigation systems with significantly higher accuracy (more than an order of magnitude) have become available in the meantime – and cost even less than the presented device.
- Laws and regulations in most countries still prevent commercial large-scale, airborne robotics from becoming wide-spread, even more so for autonomous systems. In the end, this is an issue of reliability: several reports from both civil and military sources document that the incidence of catastrophic crashes ("Class-A mishap rate" in military parlance) of UAVs is still between 5 and 50 times that of classic piloted aircraft. As long as this status remains, the future of commercial

and personal UAV use looks dim.

- Oftentimes, a point cloud is only an intermediate product. Surface reconstruction, which converts a point cloud back into (textured) surfaces, is a complex endeavor in itself, especially when it has to be performed in real-time. Although this problem might also be a good candidate for a GPU implementation, it was excluded from this thesis.

6.2 Future research directions

The algorithm for generation of waypoints can be further improved in terms of efficiency. Because most parts of the execution are memory-bound, the impact of using half-floats (i.e. binary16 in IEEE 754 parlance) for at least the collider positions can be researched, as it allows doubling both capacity and perceived memory bandwidth. This data format has been supported by OpenGL since version 3.0. CUDA supports half floats merely as a storage format, but conversion to single-precision floats requires only a single instruction. Especially when applied to outdoor scenarios, a precision in the centimeter range for the colliders is more than sufficient for gap detection. Particles' collision positions can also be converted to half-floats, but this is expected to have less effect, since updates to these values are comparatively rare. Whether particle positions and velocities can also be stored with lower precision needs to be investigated, as slight changes in the particle system's parameters often translate to large changes in the particles' behavior.

Another option for even faster execution is the possibility of using multiple graphics cards. While embarking on this path strongly constrains the choice of mobile hardware accommodating this setup, a separation of the subtasks performed seems possible. A promising separation would be to execute particle/particle collisions on one graphics card and particle/collider positions on the other. After the collision-induced changes of the particle's velocities are computed on each card, they would simply have to be added to the other card's particle-velocities after each iteration. Because the time spent visualizing the dense point cloud, particles, colliders and the grid of information gain is non-negligible (but not included in the results above), another option is to separate data visualization from data processing: during flight, the dense point cloud could be rendered (and newly appended points could be reduced) on one graphics card, with the result being sent directly to the second card, using direct memory transfers to completely bypass the CPU.

To further optimize memory access patterns, other cell-hashing functions that are expected to provide better locality than the currently used simple serial hashing function can be considered. Tests with Hilbert- and Z-Order curves (Morton code) could be performed, as suggested by Green (2012).

We have successfully developed a planar surface based outdoor mapping system in our

previous work Xiao et al. (2013), which is fast, accurate and robust compared to state-of-the-art algorithms, but not fully autonomous, because a human operator is required for viewpoint planning. It is therefore interesting to embed the NBV planning algorithm presented in this thesis into the system for autonomous exploration tasks.

References

Adler, B., Xiao, J., and Zhang, J. (2012). Towards Autonomous Airborne Mapping of Urban Environments. In *2012 IEEE Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 77–82.

Adler, B., Xiao, J., and Zhang, J. (2013). Finding Next Best Views for Autonomous UAV Mapping through GPU-Accelerated Particle Simulation. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1056–1061.

Adler, B., Xiao, J., and Zhang, J. (2014). Autonomous Exploration of Urban Environments using Unmanned Aerial Vehicles. *Journal of Field Robotics*, 31(6):912–939.

Adrados, C., Girard, I., Gendner, J.-P., and Janeau, G. (2002). Global Positioning System (GPS) Location Accuracy Improvement due to Selective Availability Removal. *Comptes Rendus Biologies*, 325(2):165–170.

Andreasson, H. and Lilienthal, A. J. (2010a). 6d scan registration using depth-interpolated local image features. *Robotics and Autonomous Systems*, 58(2):157–165.

Andreasson, H. and Lilienthal, A. J. (2010b). 6D Scan Registration using Depth-Interpolated Local Image Features. *Robotics and Autonomous Systems*, 58(2):157–165.

Angelino, C. V., Baraniello, V. R., and Cicala, L. (2012). UAV Position and Attitude Estimation using IMU, GNSS and Camera. In *2012 International Conference on Information Fusion (FUSION)*, pages 735–742.

Anonymous (2010). Kinect Hacking 105: Full Resolution, Public Domain Images of the Speckle Pattern. `http://www.futurepicture.org/?p=129`. [Online; accessed 14-April-2014].

AscTec GmbH (2010). *CoreExpress Carrierboard Manual v1.0*. Ascending Technologies GmbH.

ASPRS Board (2003). *ASPRS LIDAR Data Exchange Format Standard*. The American Society for Photogrammetry and Remote Sensing.

Besl, P. and McKay, N. D. (1992). A Method for Registration of 3D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256.

Blaer, P. and Allen, P. K. (2007). Data Acquisition and View Planning for 3D Modeling Tasks. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 417–422.

Blodow, N., Goron, L. C., Marton, Z.-C., Pangercic, D., Ruhr, T., Tenorth, M., and Beetz, M. (2011). Autonomous Semantic Mapping for Robots Performing Everyday Manipulation Tasks in Kitchen Environments. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4263–4270.

Borenstein, J. and Koren, Y. (1995). Error Eliminating Rapid Ultrasonic Firing for Mobile Robot Obstacle Avoidance. *IEEE Transactions on Robotics and Automation*, 11(1):132–138.

Borrmann, D., Elseberg, J., Lingemann, K., Nüchter, A., and Hertzberg, J. (2008). Globally Consistent 3D Mapping with Scan Matching. *Robotics and Autonomous Systems*, 56(2):130–142.

Bosse, M. and Zlot, R. (2008). Map Matching and Data Association for Large-Scale Two-Dimensional Laser Scan-based SLAM. *The International Journal of Robotics Research*, 27(6):667–691.

Bosse, M. and Zlot, R. (2009). Continuous 3D Scan-Matching with a spinning 2D Laser. In *2009 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4312–4319.

Bosse, M., Zlot, R., and Flick, P. (2012). Zebedee: Design of a Spring-Mounted 3-D Range Sensor with Application to Mobile Mapping. *IEEE Transactions on Robotics*, 28(5):1104–1119.

Bryson, M. and Sukkarieh, S. (2006). Active Airborne Localisation and Exploration in Unknown Environments using Inertial SLAM. In *IEEE Aerospace Conference*, page 13 pp.

Bryson, M. and Sukkarieh, S. (2008). Observability Analysis and Active Control for Airborne SLAM. *IEEE Transactions on Aerospace and Electronic Systems*, 44(1):261–280.

Buss, H. and Busker, I. (2012). Mikrokopter Serial Protocol. `http://www.mikrokopter.de/ucwiki/en/SerialProtocol/`. [Online; accessed 12-October-2012].

Chen, S., Li, Y., Zhang, J., and Wang, W. (2008). *Active Sensor Planning for Multiview Vision Tasks*. Springer.

Chen, Y. and Medioni, G. (1991). Object Modeling by Registration of Multiple Range Images. In *1991 IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2724–2729.

Choset, H. and Nagatani, K. (2001). Topological Simultaneous Localization and Mapping (SLAM): Toward Exact Localization Without Explicit Localization. *IEEE Transactions on Robotics and Automation*, 17(2):125–137.

Cohen, C. (1996). Attitude Determination. Global Positioning System, Theory and Applications. In Parkinson, B. W. and Spilker, J. J., editors, *Progress in Astronautics and Aeronautics*, volume 164, pages 519–538. AIAA, Washington DC.

Connolly, C. (1985). The Determination of Next Best Views. In *1985 IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 432–435.

Coumans, E. (2008). Physics Parallelization. In *Game Developers Conference 2008*, San Francisco.

Coumans, E. (2013). Bullet Physics Library. `http://bulletphysics.org/`. [Online; accessed 19-February-2013].

Demski, P., Mikulski, M., and Koteras, R. (2013). Characterization of Hokuyo UTM-30LX Laser Range Finder for an Autonomous Mobile Robot. In Nawrat, A., Simek, K., and Świerniak, A., editors, *Advanced Technologies for Intelligent Systems of National Border Security*, volume 440 of *Studies in Computational Intelligence*, pages 143–153. Springer Berlin Heidelberg.

Dissanayake, M. W. M. G., Newman, P., Clark, S., Durrant-Whyte, H., and Csorba, M. (2001). A Solution to the Simultaneous Localization and Map Building (SLAM) Problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241.

Einhorn, E., Langner, T., Stricker, R., Martin, C., and Gross, H.-M. (2012). MIRA – Middleware for Robotic Applications. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 2591–2598, Vilamoura, Portugal.

Eissfeller, B., Ameres, G., Kropp, V., and Sanroma, D. (2007). Performance of GPS, GLONASS and Galileo. In *2007 Photogrammetric Week*, pages 185–199, Stuttgart.

Elfes, A. (1989). Using Occupancy Grids for Mobile Robot Perception and Navigation. *Computer*, 22(6):46–57.

Eling, C., Zeimetz, P., and Kuhlmann, H. (2013). Development of an Instantaneous GNSS/MEMS Attitude Determination System. *GPS Solutions*, 17(1):129–138.

Elseberg, J., Borrmann, D., and Nuchter, A. (2012). 6DOF semi-rigid SLAM for Mobile Scanning. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1865–1870.

Fairfield, N. (2009). *Localization, Mapping, and Planning in 3D Environments*. PhD thesis, The Robotics Institute, Carnegie Mellon University.

Frei, E. and Beutler, G. (1990). Rapid Static OPositioning based on the Fast Ambiguity Resolution Approach 'FARA': Theory and First Results. *Manuscripta Geodaetica*, 15(4):325–356.

Frueh, C. and Zakhor, A. (2001). 3D Model Generation for Cities using Aerial Photographs and Ground Level Laser Scans. In *2001 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages II 31–II 38.

Galindo, C., Saffiotti, A., Coradeschi, S., Buschka, P., Fernandez-Madrigal, J., and Gonzalez, J. (2005). Multi-hierarchical Semantic Maps for Mobile Robotics. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2278–2283.

Georgiev, A. and Allen, P. K. (2004). Localization Methods for a Mobile Robot in Urban Environments. *2004 IEEE Transactions on Robotics*, 20(5):851–864.

Ghosh, S. K. (2007). *Visibility Algorithms in the Plane*. Cambridge University Press.

Green, S. (2012). *Particle Simulation using CUDA*. NVIDIA, Santa Clara, CA, USA.

Hager, J. W., Behensky, J. F., and Drew, B. W. (1989). *The Universal Grids: Universal Transverse Mercator (UTM) and Universal Polar Stereographic (UPS)*. Fairfax, Virginia.

Harada, T. (2007). Real-Time Rigid Body Simulation on GPUs. In Nguyen, H., editor, *GPU Gems 3*. Addison Wesley, Boston.

Heikkinen, M. (1982). Geschlossene Formeln zur Berechnung Räumlicher Geodätischer Koordinaten aus Rechtwinkligen Koordinaten. *Zeitschrift für Vermessungswesen*, 107:207–211.

Holenstein, C., Zlot, R., and Bosse, M. (2011). Watertight Surface Reconstruction of Caves from 3D Laser Data. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3830–3837, San Francisco, CA, USA.

Horn, B. K. P. (1984). Extended gaussian images. *Proceedings of the IEEE*, 72(12):1671–1686.

Hwang, D.-H., Oh, S. H., Lee, S. J., Park, C., and Rizos, C. (2005). Design of a Low-Cost Attitude Determination GPS/INS Integrated Navigation System. *GPS Solutions*, 9(4):294–311.

Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., et al. (2011). KinectFusion: Real-Time 3D Reconstruction and Interaction using a Moving Depth Camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568. ACM.

Johnson, A. (1997). *Spin-Images: A Representation for 3-D Surface Matching*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Joho, D., Stachniss, C., Pfaff, P., and Burgard, W. (2007). Autonomous Exploration for 3D Map Learning. In *Autonome Mobile Systeme 2007*, pages 22–28. Springer.

Karney, C. F. (2011). Geodesics on an Ellipsoid of Revolution. *arXiv preprint arXiv:1102.1215*.

Karney, C. F. F. (2012). GeographicLib Software Library.

Khopkar, C. D. (1997). Solving the Art Gallery Problem via Genetic Programming. In Koza, J. R., editor, *1997 Genetic Algorithms and Genetic Programming at Stanford*, pages 110–119. Stanford Bookstore, Stanford, California, 94305-3079 USA.

Kim, D. and Langley, R. B. (2000). GPS Ambiguity Resolution and Validation: Methodologies, Trends and Issues. In *7th GNSS Workshop, International Symposium on GPS/GNSS*.

Kramer, J., Burrus, N., Herrera, D., Echtler, F., and Parker, M. (2012). *Hacking the Kinect*. Apress, New York, NY.

Kumar, S. and Moore, K. B. (2002). The Evolution of Global Positioning System (GPS) Technology. *Journal of Science Education and Technology*, 11(1):59–80.

Kümmerle, R., Triebel, R., Pfaff, P., and Burgard, W. (2008). Monte Carlo Localization in Outdoor Terrains using Multilevel Surface Maps. *Journal of Field Robotics*, 25(6-7):346–359.

Lange, I. and Brümmer, B. (2010). Hochfrequente 3D-Windmessdaten vom Wettermast Hamburg. personal message/email. KlimaCampus Hamburg.

Lemoine, F. G., Kenyon, S. C., Factor, J. K., Trimmer, R. G., Pavlis, N. K., Chinn, D. S., Cox, C. M., Klosko, S. M., Luthcke, S. B., Torrence, M. H., Wang, Y. M., Williamson, R. G., Pavlis, E. C., Rapp, R. H., and Olson, T. R. (1998a). The EGM96 Geoid Undulation with Respect to the WGS84 Ellipsoid. In Lemoine et al. (1998b).

Lemoine, F. G., Kenyon, S. C., Factor, J. K., Trimmer, R. G., Pavlis, N. K., Chinn, D. S., Cox, C. M., Klosko, S. M., Luthcke, S. B., Torrence, M. H., Wang, Y. M., Williamson, R. G., Pavlis, E. C., Rapp, R. H., and Olson, T. R. (1998b). *The Development of the Joint NASA GSFC and the National Imagery and Mapping Agency (NIMA) Geopotential Model EGM96*. Number 206861 in TP-1998. National Aeronautics and Space Administration / Goddard Space Flight Center.

Li, Y., Efatmaneshnik, M., and Dempster, A. (2012). Attitude Determination by Integration of MEMS Inertial Sensors and GPS for Autonomous Agriculture Applications. *GPS Solutions*, 16(1):41–52.

Li, Y. and Murata, M. (2002). New Approach to Attitude Determination using Global Positioning System Carrier Phase Measurements. *Journal of guidance, control, and dynamics*, 25(1):130–136.

Lumpkin, B. (1997). *Geometry Activities from many Cultures*. J. Weston Walch.

Machmudah, A., Parman, S., and Zainuddin, A. (2010). UAV Bezier Curve Maneuver Planning using Genetic Algorithm. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, GECCO '10, pages 2019–2022. ACM.

Makarenko, A., Williams, S., Bourgault, F., and Durrant-Whyte, H. (2002). An Experiment in Integrated Exploration. In *2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 534–539.

Merrill, D. and Grimshaw, A. (2011). High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters*, 21(02):245–272.

Mertz, C., Koppal, S. J., Sia, S., and Narasimhan, S. G. (2012). A low-power Structured Light Sensor for Outdoor Scene Reconstruction and Dominant Material Identification. In *Proceedings of the IEEE Workshop on Projector-Camera Systems (PROCAMS)*.

Mikrokopter, P. (2013). ROXXY2827-35 Current/Thrust Dataset. `http://www.mikrokopter.de/ucwiki/ROXXY2827-35/`. [Online; accessed 2013-10-23].

Mobarhani, A., Nazari, S., Tamjidi, A. H., and Taghirad, H. (2011). Histogram Based Frontier Exploration. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1128–1133, San Francisco, CA, USA.

Montemerlo, M., Thrun, S., Koller, D., Wegbreit, B., et al. (2002). FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In *2002 AAAI National Conference on Artificial Intelligence*, Edmonton, Canada.

Moravec, H. P. and Elfes, A. (1985). High Resolution Maps from Wide Angle Sonar. In *1985 IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 116–121.

Newcombe, R. A., Davison, A. J., Izadi, S., Kohli, P., Hilliges, O., Shotton, J., Molyneaux, D., Hodges, S., Kim, D., and Fitzgibbon, A. (2011). KinectFusion: Real-time dense Surface Mapping and Tracking. In *2011 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 127–136, Basel, Switzerland.

Nørretranders, T. (1999). *The User Illusion: Cutting Consciousness Down to Size*. Penguin Press Science.

Nüchter, A. and Hertzberg, J. (2008). Towards Semantic Maps for Mobile Robots. *Robotics and Autonomous Systems*, 56(11):915–926.

Nüchter, A., Lingemann, K., and Hertzberg, J. (2006). Extracting Drivable Surfaces in Outdoor 6D SLAM. In *The 37nd International Symposium on Robotics (ISR'06)*.

Nüchter, A., Lingemann, K., and Hertzberg, J. (2007a). Cached k-d Tree Search for ICP Algorithms. In *2007 International Conference on 3-D Digital Imaging and Modeling (3DIM)*, pages 419–426.

Nüchter, A., Lingemann, K., Hertzberg, J., and Surmann, H. (2007b). 6D SLAM – 3D Mapping Outdoor Environments. *Journal of Field Robotics*, 24(8-9):699–722.

Null, B. D. and Sinzinger, E. D. (2006). Next Best View Algorithms for Interior and Exterior Model Acquisition. In *2006 International Conference on Advances in Visual Computing*, ISVC'06, pages 668–677, Berlin, Heidelberg. Springer-Verlag.

Olson, C. (2000). Probabilistic self-localization for mobile robots. *IEEE Transactions on Robotics and Automation*, 16(1):55–66.

O'Rourke, J. (1987). *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, NY.

Pangercic, D., Pitzer, B., Tenorth, M., and Beetz, M. (2012). Semantic Object Maps for Robotic Housework-Representation, Acquisition and Use. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4644–4651.

Pathak, K., Borrmann, D., Elseberg, J., Vaskevicius, N., Birk, A., and Nuchter, A. (2010). Evaluation of the robustness of planar-patches based 3d-registration using marker-based ground-truth in an outdoor urban scenario. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 5725–5730. IEEE.

Pfister, S. T., Roumeliotis, S. I., and Burdick, J. W. (2003). Weighted Line Fitting Algorithms for Mobile Robot Map Building and Efficient Data Representation. In *2003 IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 1304–1311.

Potthast, C. and Sukhatme, G. S. (2014). A Probabilistic Framework for Next Best View Estimation in a Cluttered Environment. *Journal of Visual Communication and Image Representation*, 25(1):148–164.

Remolina, E. and Kuipers, B. (2004). Towards a General Theory of Topological Maps. *Artificial Intelligence*, 152(1):47–104.

Rizos, C. and Han, S. (2003). Reference Station Network Based RTK Systems – Concepts and Progress. *Wuhan University Journal of Natural Sciences*, 8(2):566–574.

Roberts, J. M., Corke, P. I., and Buskey, G. (2002). Low-Cost Flight Control System for a Small Autonomous Helicopter. In *2002 Australasian Conference on Robotics and Automation*, volume 1, pages 71–76. Australian Robotics Automation Association.

Rusu, R., Marton, Z., Blodow, N., Holzbach, A., and Beetz, M. (2009a). Model-Based and Learned Semantic Object Labeling in 3D Point Cloud Maps of Kitchen Environments. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3601–3608.

Rusu, R. B., Blodow, N., and Beetz, M. (2009b). Fast point feature histograms (fpfh) for 3d registration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3212–3217. IEEE.

Sa, I. and Corke, P. (2011). Estimation and Control for an Open-Source Quadcopter. In *2011 Australasian Conference on Robotics and Automation*.

Salvi, J., Fernandez, S., Pribanic, T., and Llado, X. (2010). A State of the Art in Structured Light Patterns for Surface Profilometry. *Pattern Recognition*, 43(8):2666–2680.

Schuh, H. and Kutterer, H. (2013). The Global Geodetic Observing System (GGOS) of the International Association of Geodesy (IAG). In *GGOS - RAS/ROSKOSMOS Meeting*, Vienna.

Schwager, M., Julian, B. J., and Rus, D. (2009). Optimal Coverage for Multiple Hovering Robots with Downward Facing Cameras. In *2009 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3515–3522.

Scott, W. R., Roth, G., and Rivest, J.-F. (2003). View Planning for Automated Three-dimensional Object Reconstruction and Inspection. *ACM Comput. Surv.*, 35(1):64–96.

Segal, A., Haehnel, D., and Thrun, S. (2009). Generalized-ICP. In *Proceedings of Robotics: Science and Systems*, Seattle, USA.

Sengupta, S., Sturgess, P., Ladicky, L., and Torr, P. H. (2012). Automatic Dense Visual Semantic Mapping from Street-Level Imagery. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 857–862.

Septentrio, S. N. (2012). *SBF Reference Guide*. Septentrio Satellite Navigation NV/SA, Greenhill Campus, Interleuvenlaan 15G, 3001 Leuven, Belgium, 1.14.0 edition.

Shade, R. and Newman, P. (2011). Choosing Where To Go: Complete 3D Exploration with Stereo. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2806–2811, Shanghai, China.

Singh, L. and Fuller, J. (2001). Trajectory Generation for a UAV in Urban Terrain, using Nonlinear MPC. In *2001 American Control Conference*, volume 3, pages 2301–2308.

Standage, T. (2002). *The Turk: The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine*. Berkley Trade.

Strand, M. and Dillmann, R. (2010). Grid Based Next Best View Planning for an Autonomous Robot in Indoor Environments. In *2010 International Workshop on Robotics for Risky Interventions and Environmental Surveillance-Maintenance*, RISE'2010, Sheffield, UK.

Thrun, S., Burgard, W., and Fox, D. (2000). A real-time Algorithm for Mobile Robot Mapping with Applications to Multi-Robot and 3D Mapping. In *2000 IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 321–328.

Thrun, S., Burgard, W., Fox, D., Hexmoor, H., and Mataric, M. (1998). A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots. In *Machine Learning*, pages 29–53.

Thrun, S. and Montemerlo, M. (2006). The Graph SLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures. *The International Journal of Robotics Research*, 25(5-6):403–429.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., et al. (2006). Stanley: The Robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692.

Triebel, R., Pfaff, P., and Burgard, W. (2006). Multi-Level Surface Maps for Outdoor Terrain Mapping and Loop Closing. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2276–2282.

van der Hoeven, A., Hanssen, R. F., and Ambrosius, B. (2002). Tropospheric Delay Estimation and Analysis using GPS and SAR Interferometry. *Physics and Chemistry of the Earth, Parts A/B/C*, 27(4):385–390.

Van Kaick, O., Zhang, H., Hamarneh, G., and Cohen-Or, D. (2011). A survey on shape correspondence. In *Computer Graphics Forum*, volume 30, pages 1681–1707. Wiley Online Library.

Velodyne Lidar (2013). Velodyne Presents Successful Implementation of HDL-32E LiDAR on UAV at sUSB Expo in San Francisco. `http://www.prweb.com/releases/2013/7/prweb10963826.htm`. [Online; accessed 28-July-2013].

Vermeille, H. (2002). Direct Transformation from Geocentric Coordinates to Geodetic Coordinates. *Journal of Geodesy*, 76(8):451–454.

Whaite, P. and Ferrie, F. (1997). Autonomous Exploration: Driven by Uncertainty. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(3):193–205.

Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. (2010). Demystifying GPU Microarchitecture through Microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 235–246.

Wurm, K. M., Hornung, A., Bennewitz, M., Stachniss, C., and Burgard, W. (2010). OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems. In *2010 International Conference on Robotics and Automation (ICRA), Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*, Anchorage, Alaska.

Xiao, J., Adler, B., Zhang, H., and Zhang, J. (2013). Planar Segments Based 3D Point Cloud Registration in Outdoor Environments. *Journal of Field Robotics*, 30(4):552–582.

Xsens Technologies B.V. (2009). *MTi and MTx User Manual and Technical Documentation*. Xsens Technologies B.V.

Xu, A., Viriyasuthee, C., and Rekleitis, I. (2011). Optimal Complete Terrain Coverage using an Unmanned Aerial Vehicle. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2513–2519.

Yamauchi, B. (1998). Frontier-Based Exploration using Multiple Robots. In *Agents*, pages 47–53.

Yang, Y. and Farrell, J. A. (2001). Fast Ambiguity Resolution for GPS/IMU Attitude Determination. In *Proceedings of the 14th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS 2001)*, pages 2990–2997.

Zhang, L., Curless, B., and Seitz, S. M. (2002). Rapid Shape Acquisition using Color Structured Light and Multi-Pass Dynamic Programming. In *2002 International Symposium on 3D Data Processing Visualization and Transmission*, pages 24–36.

Zhao, H. and Shibasaki, R. (2003). Reconstructing a Textured CAD Model of an Urban Environment using Vehicle-borne Laser Range Scanners and Line Cameras. *Machine Vision and Applications*, 14(1):35–41.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 19. November 2014

Unterschrift