Autonomous Exploration of Urban Environments using Unmanned Aerial Vehicles

Benjamin Adler[1], Junhao Xiao[2] and Jianwei Zhang[1]

[1]Institute of Technical Aspects of Multimodal Systems
University of Hamburg
22527 Hamburg, Germany
{adler,zhang}@informatik.uni-hamburg.de

[2]Department of Automatic Control
National University of Defense Technology
137 Yanwachi street
410073 Changsha, China
junhao.xiao@ieee.org

Abstract

This paper introduces a custom-built unmanned aerial vehicle, capable of autonomous exploration in urban environments. It consists of a multicopter, an inertial navigation system and two 2D laser range finders. In addition to a description of the hardware architecture and individual components being used, the authors also discuss challenges and problems that arose during its construction as well as optimizations and workarounds applied in the course of its development.

Also presented is the software architecture, with a focus on a novel algorithm capable of generating multiple next best views (NBVs), sorted by achievable information gain. Although being designed for application on airborne platforms in urban environments, it works directly on raw point clouds and thus can be used with any sensor generating spatial occupancy information (e.g. LIDAR, RGBD- or time-of-flight-cameras). To satisfy constraints introduced by real-time operation on UAVs, the algorithm is implemented on a highly parallel SIMD architecture and benchmarked using GPUs from multiple hardware generations, using data from real flights. It is also compared against the previous, CPU-based proof of concept.

As the underlying hardware imposes limitations with regards to memory access and concurrency, necessary data structures and further performance considerations are explained in detail.

Open-source code for this paper is available at http://www.github.com/benadler/octocopter/.

1 Introduction

While robots have become a very common sight in industrial environments ever since their first application for manufacturing tasks in the middle of the last century, their adoption in other fields of human labor has been excruciatingly slow. This is surprising at first, as machines have shown superior performance in comparison to humans in many disciplines: often, they are faster, more precise, more durable and even

cheaper in the long term. In stark contrast to that, most of today's robots still are incapable of performing even the most simple tasks in acceptable time - how else can it be explained that 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: [...] navigating a backyard, recognizing a face; seeing."

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: sense, remember and recognize their environment, manipulate objects and communicate. The further away from industrial manufacturing and other highly structured environments robots are to be of help, the more important these skills become.

For this reason, autonomous environment sensing and model learning is one of the most fundamental challenges in mobile robotics. Especially in field robotics, the availability of a 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 for progress is the ability of simultaneous localization and mapping (SLAM): little over a decade ago, first-generation SLAM approaches as presented in Thrun et al. (1998) have been limited to robots moving on planes and in highly structured indoor environments. During the last decade, the algorithms advanced, keeping pace with the physical development of robots: better locomotion has allowed robots to become far more maneuverable and escape indoor environments, better energy sources have allowed for longer operation, better computers allow for faster speed. Just two years ago, algorithms were presented, capable of localizing robots with six degrees of freedom, while mapping three- or even four-dimensional space in real-time (Newcombe et al., 2011).

Still, for many real-world applications in the field, mapping an outdoor environment by simply defining a region of interest in 3D space and leaving the details of the procedure to an autonomous robot would constitute a considerable improvement.

After successfully implementing localization and mapping on an unmanned aerial vehicle to enable generation of maps even in regions that are unnavigable for ground-based robots, our research focus has been on advancing our approach from mere mapping to exploration. A large part of the difference is the generation of goal configurations in order to maximize the system's information gain. Determining this sensor placement is a generalization of the NP-hard 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.

Because our 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, as this enables creation of trajectories that include all NBV-derived waypoints in an order optimized to allow catenation by the robot in minimal time.

The work described in this paper builds on our previously published result (Adler et al., 2013). It extends the paper by presenting the hardware architecture in more detail and adding descriptions of the software modules. The approach for generation of Next Best Views is explained more thoroughly. Furthermore, new algorithms for creating collision-free flight paths between computed waypoints were amended. Waypoints and flight paths of larger point clouds are presented in a more detailed results section.

This paper is organized as follows: The next section presents a short overview of work in this and related fields. In Section 3, the general architecture of the hard- and software setup is introduced. Section 4 starts with the idea behind the approach for generating waypoints providing high information gain and continues to detail the data structures and algorithms necessary to do so. The next Section discusses how the generated

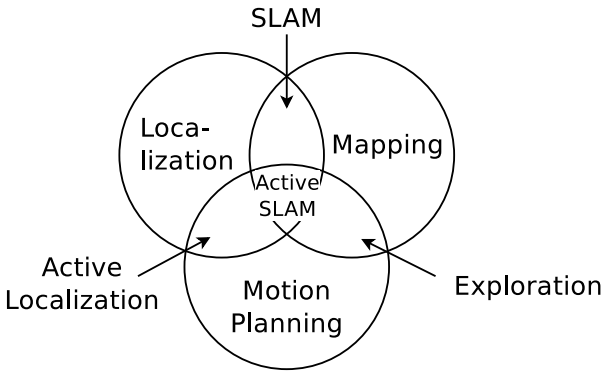


Figure 1: Subtasks of automatic model learning, from Fairfield (2009)

waypoints are used to create collision-free paths through the environment, followed by a description of the motion control systems required to let the UAV follow aforementioned paths. In Section 7, we show waypoints and paths generated for point clouds captured during real flights and analyze the real-time applicability of the algorithms. Finally, the paper is concluded in Section 8 by summarizing the results and discussing future work.

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

2 Related Work

Fairfield (2009) groups the tasks of localization, mapping and planning into their combinations SLAM, exploration, active localization and, unifying all three problems, active SLAM (see Figure 1). One one hand, this model does not allow unambiguous classification of all existing approaches due to the lack of clear definitions and leaves room for debate concerning e.g. whether localization and planning ("active localization") can exist without a map. Still, it serves as a helpful guidance. In the next paragraphs, we will focus on the state of the art in motion planning and its application to autonomous model learning.

The art gallery problem has been researched extensively by O'Rourke in 1987 and still serves as a base for many related approaches. Understandably, the presented algorithms mostly operated on polygons in two-dimensional space. Because the problem statement presumes the map to be known a-priori, these works do not present sufficient solutions for NBV planning in unknown environments.

There have been multiple publications surveying active perception planning for reconstruction and inspection, the most recent being Scott et al. (2003), which classifies methods as either surface-based, volume-based or global. 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.

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 (2011) yields helpful output so 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.

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".

Null and Sinzinger (2006) introduces 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. Surface normals are computed 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 for a grid of 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.

Because the amount of computational resources hasn't increased as fast as required by the integration of the third dimension into planning algorithms, 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 2D 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.

A frontier-based approach is commonly used in many NBV problems (Yamauchi, 1998; Shade and Newman, 2011), because it yields sensor-poses located between known and unknown regions. On 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).

For our platform, localization quality does not depend on classical map features in a way common to most SLAM algorithms, as we rely on GNSS-based localization. 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 precision of the computed position. When mapping very close to buildings or in-between high-rise architecture, obstructions of satellites becomes so important

that motion planning must consider both satellite orbits and local geometry.

Following the realization that the localization used on our vehicle is sufficiently precise in typical urban environments, we optimize the exploration strategy towards the more application-oriented metric of map completeness (not just coverage) as well as mapping duration instead of localizability.

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 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.

3 Experimental Platform

3.1 Hardware

(a) First configuration, in which laser scanner's FOV is oriented vertically and the inertial measurement unit is fixed to the scanner for precise alignment.

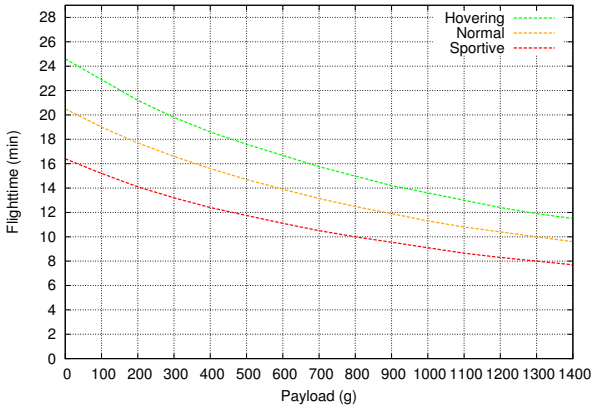
(b) Second configuration with the scanner's FOV oriented downwards and perpendicular to principal direction of travel. The inertial measurement until is mounted using vibration dampeners for improved INS performance.

Figure 2: Previous hardware configurations of the experimental flying platform with mounted GNSS-antenna and -receiver, laser scanners, IMU and processor-board. The red boom points forward.

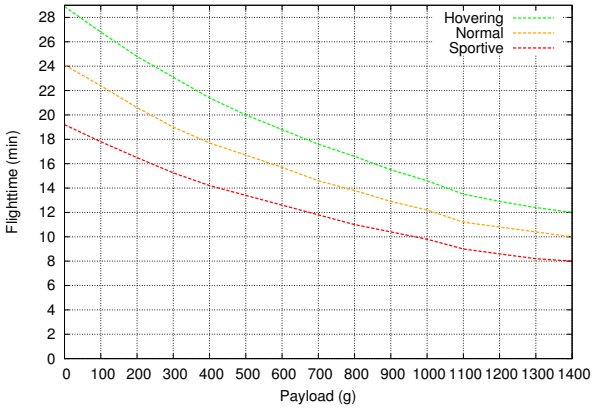
3.1.1 Unmanned Aerial Vehicle

The UAV is built upon an "Okto 2"-multirotor helicopter from the mikrokopter project (see Figure 3). 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 1 for a list of components and their weight, as well as Figure 4a 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 IMU 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 by this loss, a missing propeller was often detected

Figure 3: The current setup includes a second laser scanner for obstacle avoidance and mapping. Both laserscanners and the IMU are mounted together with the battery on a stiff and leightweight 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 INS solution, more precise sensor alignment and better crash survivability.



(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 4: Graph depicting the achievable flight times versus the payload carried by differently-sized multi-rotors, courtesy of the mikrokopter project. While these numbers are of theoretical nature, they match the experienced real-world flight-times very closely.

Table 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**

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. Its source code is published and free to use for non-commercial purposes.

The platform also carries an onboard 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, with 1GB of RAM and 8GB of non-volatile flash memory on a micro-SD card. Equipped with seven USB, two serial ports and IEEE 802.11n WiFi, it connects all devices on board the vehicle and allows for fast communication with the base station, with reduced data rate even when the line-of-sight between both is obstructed.

3.1.3 Inertial Navigation System

For self-localization, the UAV is equipped with a "Septentrio AsterX2i" commercial INS system, featuring an "XSens MTi" MEMS IMU and a dual-frequency GNSS[1] RTK[2] receiver supporting GPS and GLONASS constellations. Altogether, the IMU, GNSS-receiver, -antenna, breakout board, cables and case weigh just 184 grams. As RTK-GNSS uses constant updates of satellite observations from a GNSS base to correct for errors induced in the transmission of the satellite vehicle's SiS[3], additional infrastructure is required to feed these differential corrections to the rover in-flight. The INS also retrieves the IMU's values at 50Hz and fuses them with 10Hz GNSS PVT[4] solutions. Because the MEMS-grade IMU exhibits a drift of more than 15 degrees per hour (earth's rotational velocity), it is unable to reliably measure earth's rotation and thus is unable to perform static alignment. To solve the system's heading after startup, trajectories of IMU and

[1] Global Navigation Satellite System
[2] Real Time Kinematic
[3] signal-in-space
[4] position, velocity, time

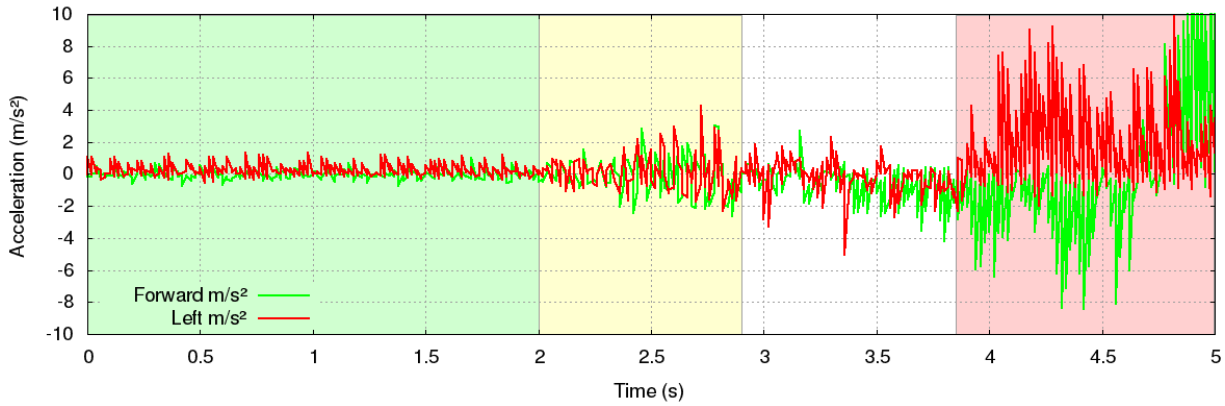


Figure 5: Raw accelerations experienced without motion (green), during motor-spinup (yellow), idling (white) and take-off (red).

GNSS have to be aligned while the vehicle moves forward (kinematic alignment). This happens during the first seconds of flight. In the next phase, covariances of the fused solution are minimized by flying curves, accelerating and stopping. The duration of this phase depends on the desired covariance thresholds and motions experienced by the INS, but usually takes less than 30 seconds.

Because the IMU is of MEMS-grade precision, is very lightweight and mounted on a vibrating platform, the values read by the INS are comparably noisy. Vibration damping - an eternal challenge for UAV-mounted sensors - becomes even more difficult because the frequency of vibration highly depends on the motor-speeds. Figure 5 shows the raw values received by the IMU at the beginning of a flight. Clearly visible are the noisy readings while parked (green section), vibrations experienced from the motors spinning up (yellow) and during flight (red). When requesting position and attitude from the INS at intervals shorter than the minimum GNSS-measurement-interval of 100ms, fused measurements (GNSS and IMU combined) will be interleaved with data that is produced by double-integrating IMU readings onto them. In case of strong vibrations, this leads to unusable integrated poses, as depicted in Figure 6. While using this data for sensor fusion with the scanned range data is somewhat detrimental to point cloud accuracy, it can have disastrous effects when used in a flight controller that contains a derivative component.

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 experiences and usually converges to a maximum error of less than 1.0° .

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 2a and connected to the aforementioned onboard computer. Its laser scanned a front-facing planar field of view of 270°, resulting in a 90°blind spot in its back, which 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 ahead of the UAV. While this idea worked satisfactorily in simulation, the actual INS was not capable of 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 IMU's values.

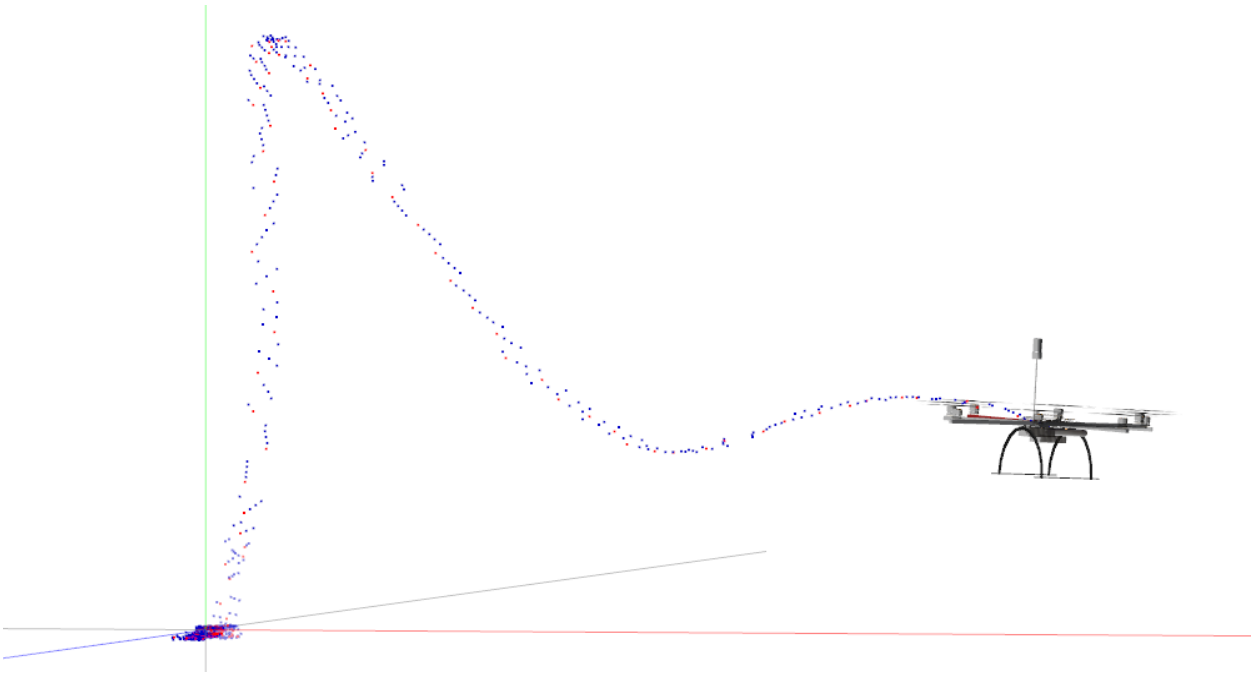


Figure 6: Visualization of the vehicle trajectory during lift-off, exhibiting considerable IMU drift and noise. Poses are sampled at 50Hz, each fused pose (red) is followed by 4 IMU-integrated poses (blue).

In a second version, 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 2b). 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 laserscanner from the IMU, resulting in a detrimental effect of unknown magnitude on the registration accuracy. Like the first revision, the scanner moved the system's center of gravity away from its geometrical center, which was partly compensated for by moving the battery.

The final setup (Figure 3) was created in order to accomodate a second, forward-looking laser scanner for better obstacle detection. Both scanner's points are fed into the same point cloud, also accelerating the scanning procedure. The comparatively lightweight onboard computer and INS were now installed above, while the heavy combination of two laser scanners, IMU and battery were placed below the center of gravity. By using carbon-fiber sandwich boards, the necessary board is extremely warp resistant and weighs just 41 grams. It directly couples the IMU with both LIDAR sensors and is less prone to vibrations as it also holds the battery. Still, it is installed using four vibration dampeners.

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, its value is generated using a local clock that starts at zero when the laserscanner 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 USB specifications. 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 enable better 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. By connecting this signal to one of the event-in pins of the GNSS receiver and configuring the latter to emit a timestamp on falling edges, each scan is timestamped with a precision of better than 10 nanoseconds, using the same GPS-derived clock as for the INS measurement's timestamps. Assuming a constant rotational velocity of the LIDAR's mirror, each single ray can be temporally associated to the received poses with microsecond-precision. 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.

The disadvantage of this mechanism is that generation of 40 timestamp-packets per second 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 align successfully. To alleviate this problem, a 74HC4040 binary counter was installed between the laser scanner and 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). This setup enables registration of the scanner's rotational phase while saving the limited computational resources of the INS' embedded system.

As soon as the INS solves time, position and attitude precisely, the onboard computer fuses data from all sensors in real-time, creating a point cloud that is streamed to the ground station during flight.

3.1.5 Electromagnetic interference

Flight controller, laser scanner and especially onboard computer emit electromagnetic interference on L1 (1575.42 MHz) and especially L2 (1227.6 MHz) bands, which is received 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 long periods of time, introducing considerable errors into the solutions of position and attitude. To rectify this problem without access to expensive equipment like spectrum analyzers, all sources of radiation were wrapped in aluminum foil. Wires to and from the onboard computer were found to act as antennas, emitting further inteference, which was fixed by running the affected cables through ferrite rings. Small parts of the interferences remain, as large parts of the laser scanner's surface must remained unshielded for obvious reasons. Moving the GNSS antenna upwards and away from the scanner has reduced the remaining effect of this problem so much that GNSS reception has become reliable. On the downside, the added shielding has imposed the need to monitor temperatures of onboard computer and motor controllers during flight.

For a complete diagram of the hardware used in this paper, please see Figure 7.

3.2 Software

During development, a custom software stack was created, consisting of three main modules: 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 very constrained hardware, all software is implemented using C++. A simple TCP/IP-based 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 2 for a complete listing.

Table 2: 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

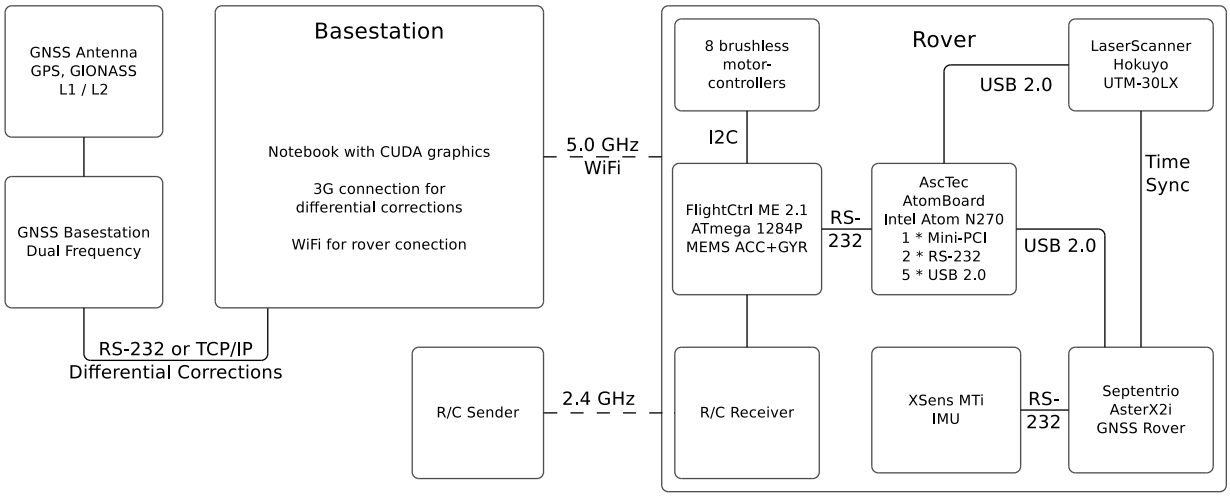


Figure 7: Diagram of hardware and interconnections used for both base and rover.

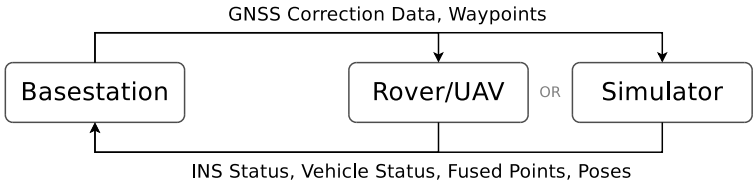


Figure 8: Communication between the main software modules. The base station connects to either the rover (UAV) or a simulator, permitting tests of the employed algorithms in simulation before they are tested on the UAV.

The simulator (see Figure 9) was created first in order to allow tests of how different scanner setups and flight controller implementations would affect speed and quality of point cloud generation. Much attention was paid to achieve realistic simulation: the physics engine (Bullet) computes forces based on precise weights of the components, while the lift-forces are determined based on experimental data relating the propeller's rotational velocity and thrust. To simulate motion induced by wind and gusts, data captured by an ultrasonic anemometer from a local weather-station was also incorporated into simulation.

The base station program (depicted in Figure 10) hosts almost all of the algorithms presented in this paper, as it runs on hardware offering far more computational resources than what are available onboard the UAV. After generating flight paths based on the received point cloud, the corresponding waypoints are sent back to the rover.

The rover program was created last: it runs on the UAV and interfaces with its FlightControl-board, the INS and laser scanners. It also implements the same communication protocol as the simulator, so the base station can connect to either the simulator or the rover and operate on simulated or real data (see Figure 8). Additionally to the communication performed between base station and simulator, the rover also sends vehicle- and INS-status to the base. The base station retrieves low-datarate RTK differential corrections (from about 0.5 to 3 kb/s, depending on the number of GNSS constellations supported and thus, satellites observed) from the GNSS reference station using a separate 2G, 3G or serial connection and forwards them to the rover. Also, a high-level motion-controller is implemented, processing incoming waypoints into control output for the UAV's low-level controller running in the FlightControl-board (see Section 6 for details). During flight, all sensor data and generated outputs from the high-level motion-controller are logged to disk. The base station program can open these files for later replay, greatly improving post-flight visualization, validation and debugging.

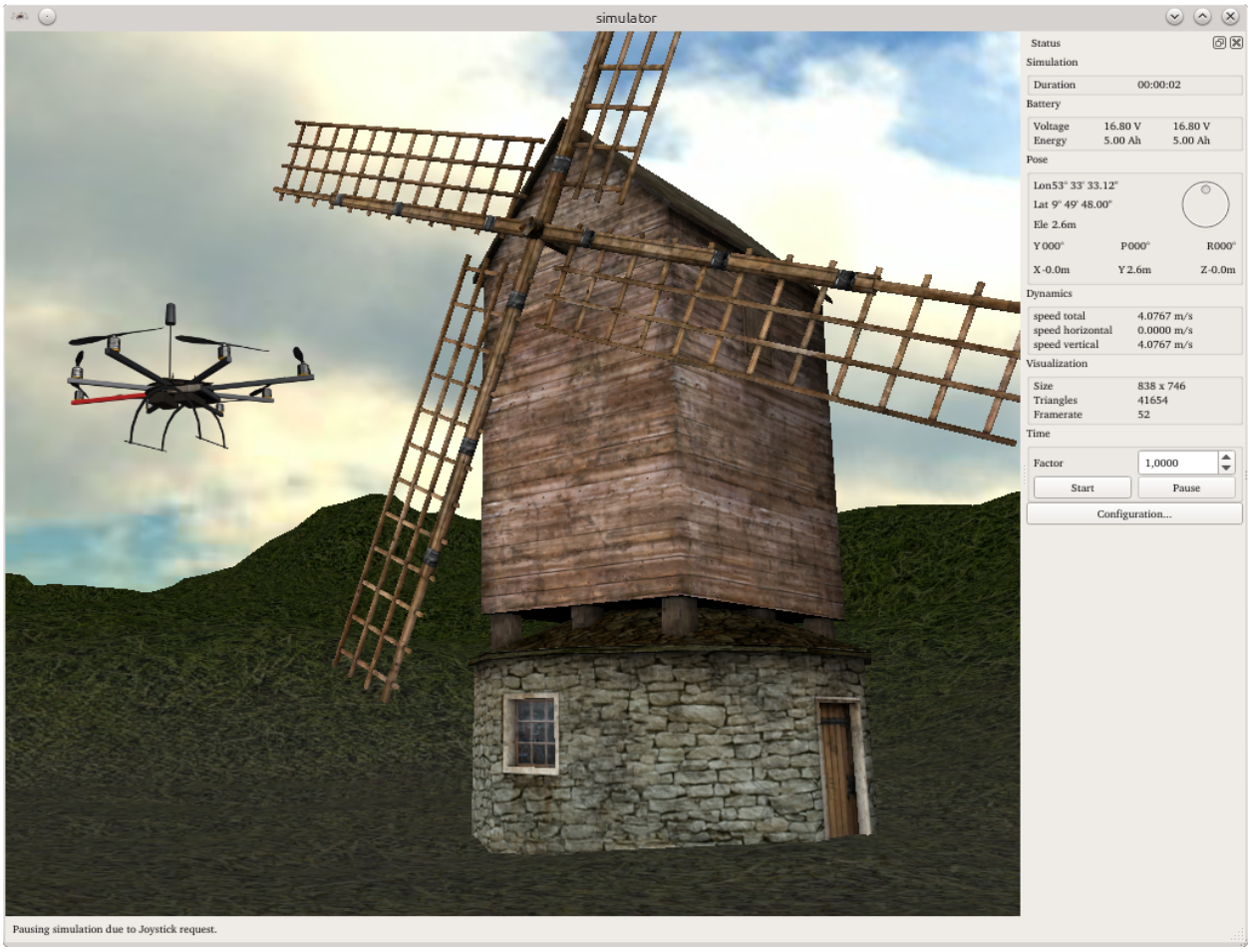


Figure 9: Screenshot of the simulator. The UAV can be controlled manually via joystick as well as autonomously using a custom high-level flight-controller that approaches generated waypoints.

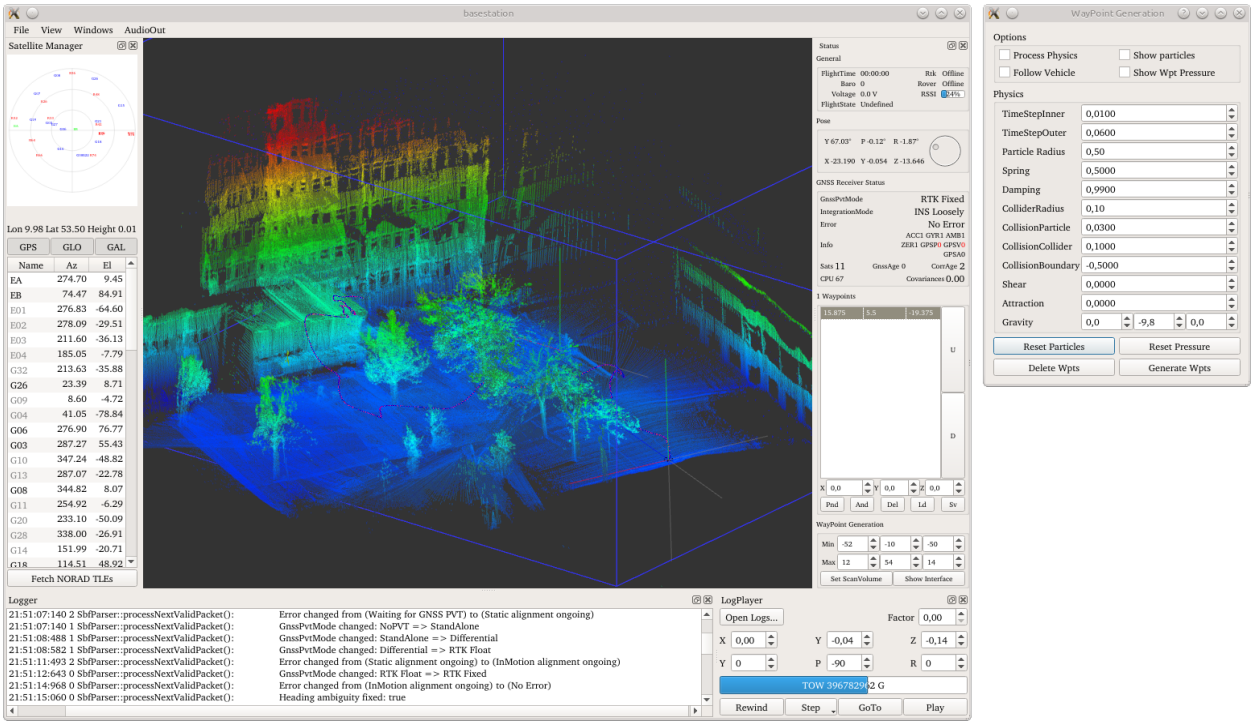


Figure 10: Screenshot of the basestation program, running on a notebook computer in the field. Depicted are the point cloud, streamed in real-time from the vehicle, as well as GNSS status indicators and flight planning controls.

4 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 for finding 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) 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 ($p \in P$) over C (see Figure 11). We can postulate that whenever a $p \in P$ first collides with a $c \in C$ 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.

This algorithm has previously been implemented in a similar form using CPU implementations in the bullet physics library (Coumans, 2013): the point cloud was managed using a fast dynamic bounding volume tree based on axis aligned bounding boxes for the broadphase collision detection. While the implementation proved to be a working concept, it turned out to be far too slow to handle the enormous number of collision-

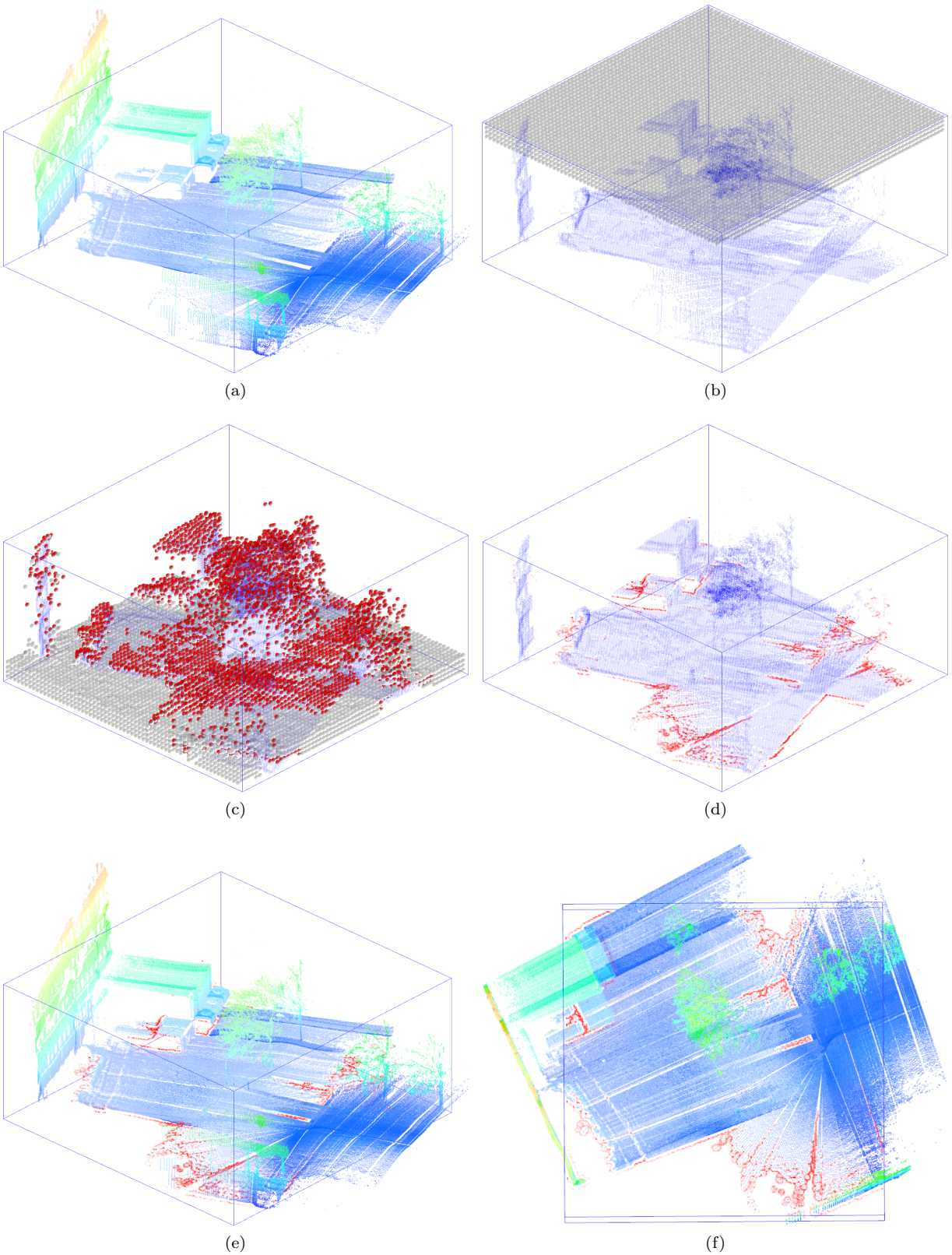


Figure 11: Overview of the process: a) shows predefined bounding box b and the initial point cloud from the onboard laser scanner. b) shows 16k particles $p \in P$ in gray being poured over downsampled cloud of colliders $c \in C$ in blue. c) depicts one timestep of simulation, with falling particles $p \in P$ 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.

tests necessary for rapid detection of small gaps. Thus, the algorithm was ported to CUDA running on a graphics card, and optimized for real-time applicability.

4.1 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 Data Reduction on the GPU

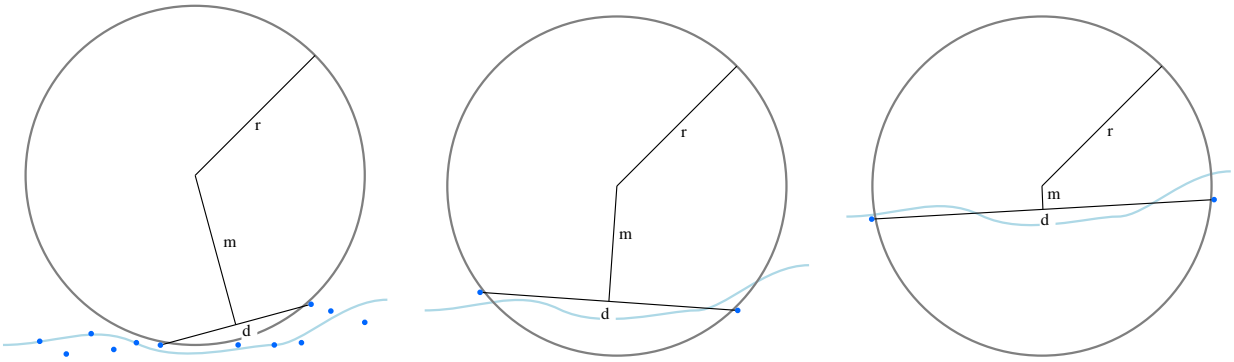
As shown in section 7, the particle simulation's bottlenecks are collision-detection and -processing. In every iteration of the simulation, this process is first executed between the simulated particles themselves, then executed between the particles and the collider cloud. Thus, optimizing it becomes key to detecting gaps and generate 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 12a) 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 12b). 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:

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

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,



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

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 12c). Thus,

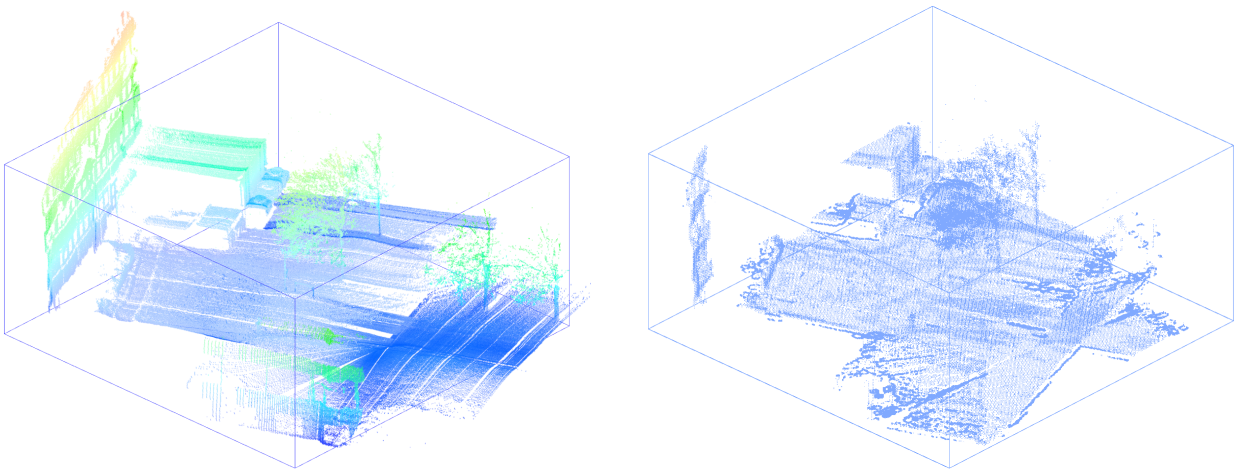
$$p_{\vec{v}} * \Delta t < \sqrt{r^2 - (\frac{d}{2})^2}$$

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. 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 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 13. An overview of this process is given in Figure 14: 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 15a), 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 $N_{G_{SHT}} = G_{SHT_x} * G_{SHT_y} * G_{SHT_z}$ cells is created to enable efficient access to neighboring colliders.



(a) (b)

Figure 13: (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.

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 15b 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, using a fast radix sort on the GPU (similar to Satish et al. (2009)).

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 15c. 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).

In step 7, the values in VBO_C can finally be compacted, in effect removing all zero-points. This free space

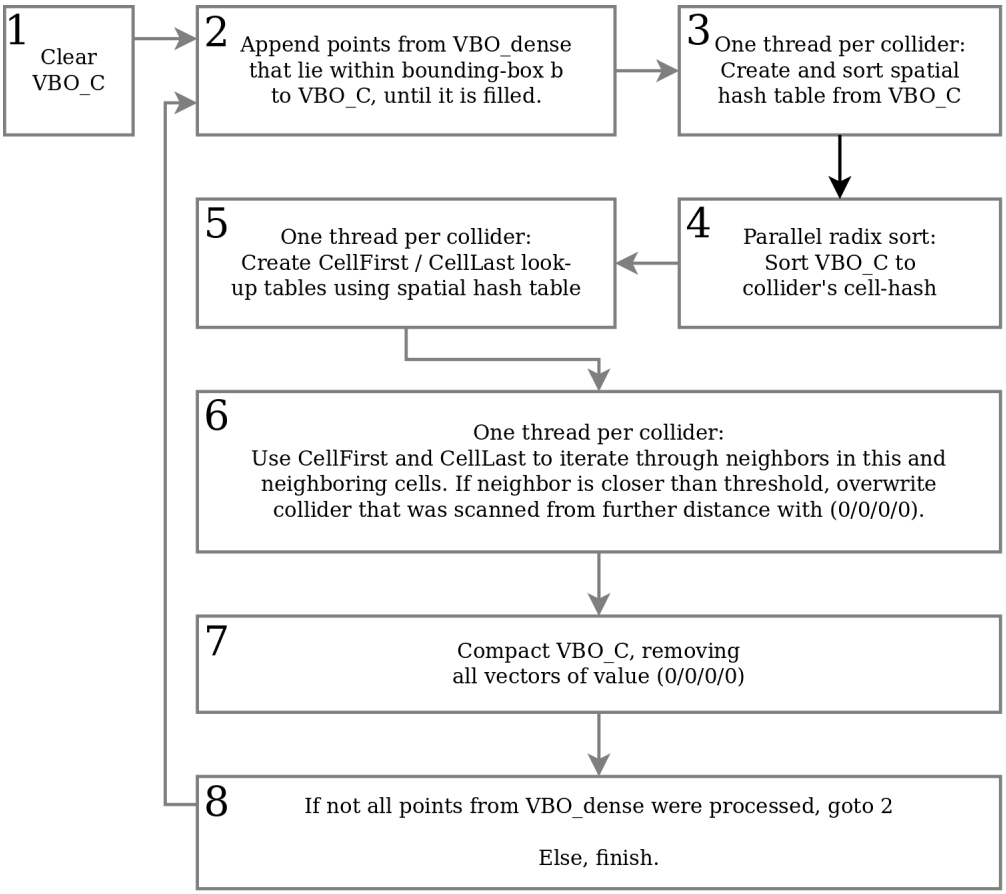


Figure 14: A flow-chart describing the process of downsampling the point cloud in VBO_{dense} to a sparse version C stored in VBO_C.

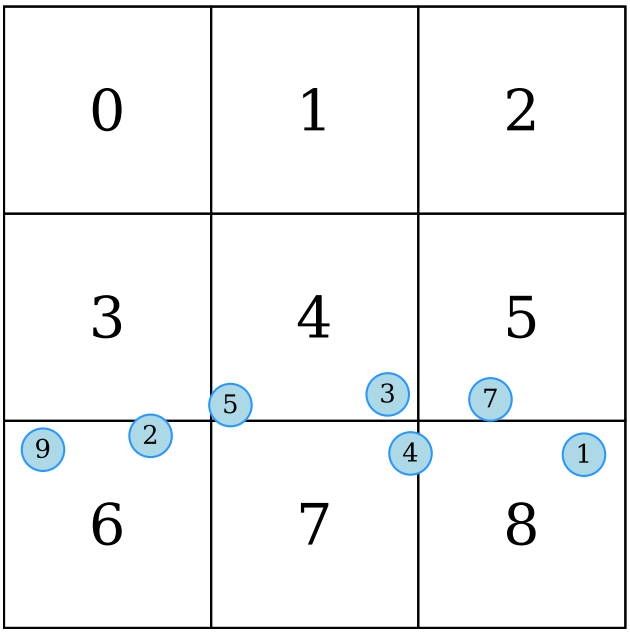
allows more points from the dense point cloud to be appended and reduced again, as decided in step 8.

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 7.

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 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.

4.3 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



(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 15: 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 16: 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.

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 17. 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 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, requiring further data structures to be allocated in the graphics card's memory, as shown in Figure 16.

Using these structures, a single iteration of the particle simulation is processed as described in Algorithm 1:

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 in lines 9-19: 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 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.

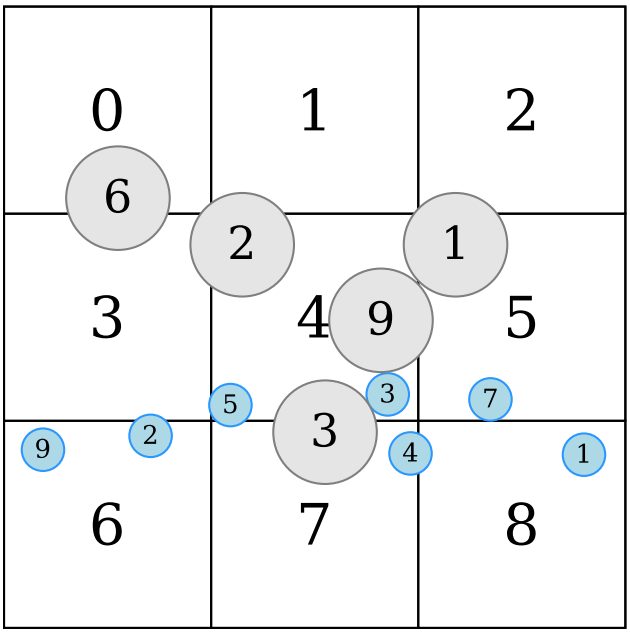
To detect and process collisions, N_P threads fetch $P_{pos}[threadId]$ and compute the hash value of G_{SHT}'s respective cell. They then iterate through its own and all $3^3 - 1 = 26$ neighboring cells in the grids of the SHTs for *both* other particles and colliders (lines 25-44), re-using the vectors built in section 4.2. 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 35), 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 null, used to increment the information gain of G_{IG}'s cell containing it in line 54.

After multiple iterations, particle simulation is terminated (see section 7 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.

Algorithm 1 Massively parallel test for watertightness of C

```
1: // Build spatial hash table for particles
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: // Build cell lookup table for particles
9: allocate sharedHash[N_P]
10: for each core i < N_P do in parallel
11:     sharedHash[i + 1] ← P_index[i]
12:     synchronizeThreads( )
13:     if sharedHash[i] ≠ sharedHash[i − 1] then
14:         P_cellFirst[P_cellhash[i]] ← i
15:         if i > 0 then
16:             P_cellLast[sharedHash[i + 0]] ← i
17:         end if
18:     end if
19: end for
20:
21: // Repeat lines 2–19 for colliders SHT and cell
22: // lookup table if collider cloud C changed
23:
24: // collide particles
25: for each core i < N_P do in parallel
26:     force ← (0, 0, 0)
27:     cellHashes ← getNeighborHashes(P_pos[i])
28:     for each h ∈ cellHashes do
29:         // collide p against colliders
30:         for j ← C_cellFirst[h], C_cellLast[h] do
31:             force += collideDEM(P_pos[i], C_pos[j])
32:         end for
33:         // save particle pos in case of collider-collision
34:         if force ≠ (0, 0, 0) then
35:             P_col[i] ← P_pos[i]
36:         end if
37:
38:         // collide p against other particles
39:         for j ← P_cellFirst[h], P_cellLast[h] do
40:             force += collideDEM(P_pos[i], P_pos[j])
41:         end for
42:         P_vel[i] ← (force + P_vel[i])
43:     end for
44: end for
45:
46: // Integrate motion
47: for each core i < N_P do in parallel
48:     P_vel[i] ← damping ∗ (P_vel[i] + (g ∗ Δt));
49:     P_vel[i] ← collideWithBoundingBox(P_pos[i])
50:     P_pos[i] ← P_pos[i] + (P_vel[i] ∗ Δt)
51:     if P_pos[i].y < b.min.y then
52:         P_pos[i].y ← b.max.y
53:         if Pcol[i] ≠ (0, 0, 0) then
54:             G_IG[getCellHash(Pcol[i])]+=1
55:             Pcol[i] ← (0, 0, 0)
56:         end if
57:     end if
58: 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 17: Additional vectors in GPU memory, required for particle simulation. Their structure is completely analogous to that described in Figure 15.

5 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 18, left), in effect creating a three-dimensional occupancy grid (Figure 18, 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 18, 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 were 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: 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 repeats 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 18, 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.

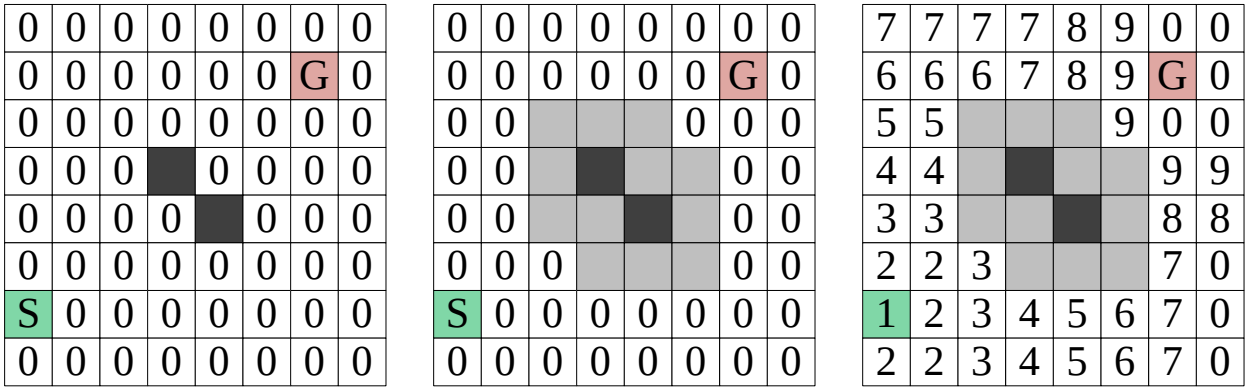


Figure 18: Parallelized path planning using a uniform occupancy grid. Left: Initialized occupancy grid with start and goal cells marked and cells containing colliders being marked as occupied. Center: After dilation, searching for traversable cells in the diagonal neighborhood is safe. Right: Populated 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.

6 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 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 3. States transition automatically from *ApproachWaypoint* to *Hover* when there are no more waypoints and back, when new waypoints arrive.

Table 3: 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 ("ExternControl")**	**High-Level controller state (waypoints absent)**	**High-Level controller state (waypoints present)**
UserControl	-107	Disabled	UserControl	UserControl
Hover	20	Enabled	Hover	Hover
ApproachWaypoint	128	Enabled	Hover	ApproachWaypoint

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 *UserControl* 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 forwards. To avoid slow and clumsy cylces of "orient towards target", "fly forwards" 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, 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 behaviour 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 6). 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, this rate is still sufficient for flight control.

Proportional and derivatve 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.

7 Results

7.1 Localization and environmental conditions

Currently, the system relies on the INS providing accurate position and attitude solutions, requiring reception of signals on different frequencies from a sufficient number of satellites. During INS 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, which works reliably

in all but the most occluded local horizons. Once the carrier phase ambiguities have been solved, *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 INS is unable to *start* centimeter-precision positioning using signals from mixed constellations, e.g. 3 GPS and 3 GLONASS satellites. Once the PVT is supported by both GPS and GLONASS, we experienced very robust positioning even very 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 inteference 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, the base station notifies the pilot in this case, allowing manual control.

While the precision of the point cloud produced has not been investigated thoroughly, 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 traditional surveying applications, the data produced is precise enough to be used for motion planning, collision avoidance and some applications of surface reconstruction. Figure 19 gives an impression of the resulting cloud by showing the UAV scanning nearby persons in-flight. The 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.

The high-level motion controller is able to navigate the UAV reliably through open terrain and accomodates the inertial navigation system's special requirements with regard to flight dynamics. The UAV has been tested at outside temperatures down to -10°C and the motion controller is known to handle wind gusts of up to 10.8 m/s (the video shows a flight at October 1st, 2013, with the temperature at 12°C and an average wind of 4.2m/s with gusts up to 10.8m/s, according to the weather station of the nearby hamburg airport).

7.2 Generated waypoints and paths

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. This 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 21, 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 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. As expected, profiling shows 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.

Further results are presented based on data gathered during a short flight over the campus of the department of computer science of the University of Hamburg. Both the campus and the resulting point cloud (colored according to height) are displayed in Figure 22. After about 100 seconds of flight within a bounding box of 64*64*32 meters, motion planning is executed: Figure 23a 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

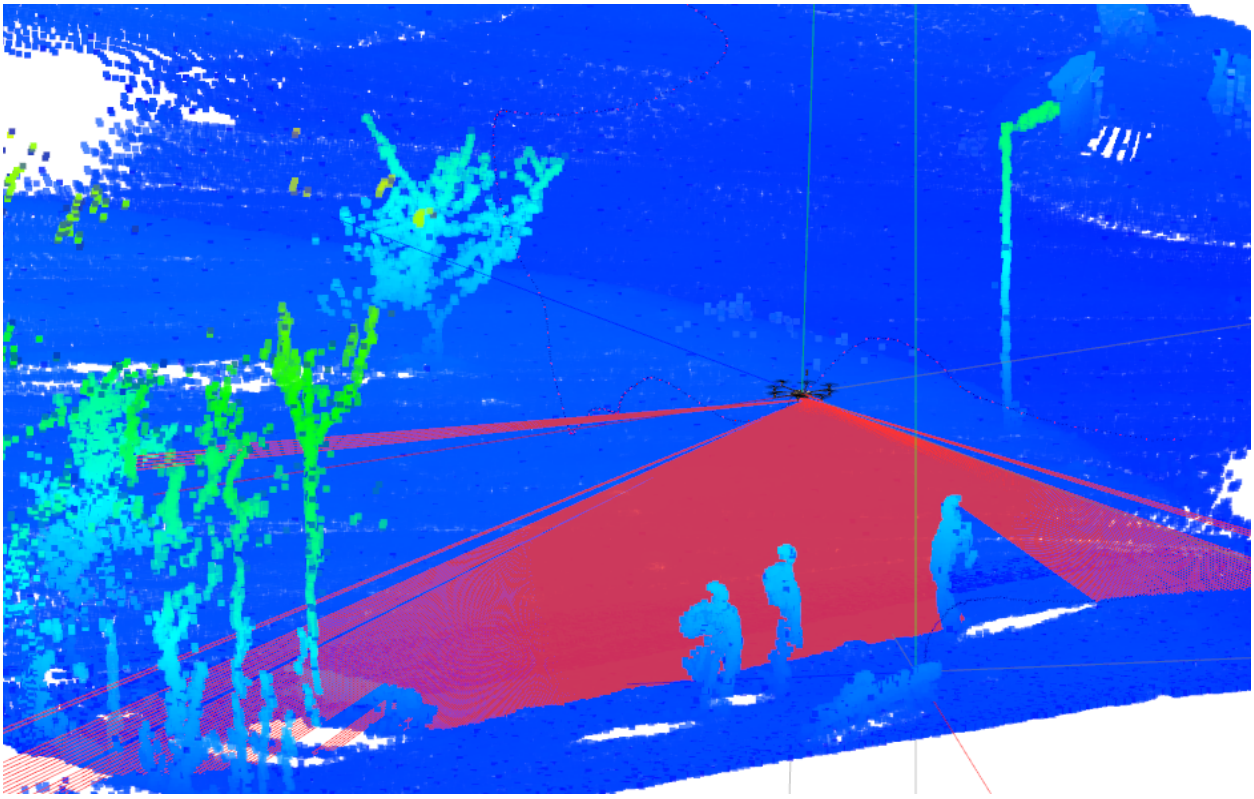
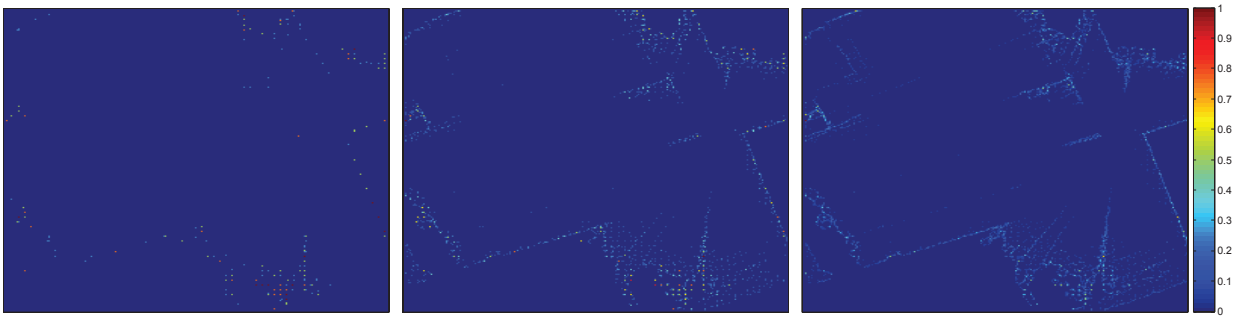


Figure 19: Close-up of the UAV scanning nearby persons on the campus. A single scan is shown, visualizing every ray that received a reflection in red.



(a) (b) (c)

Figure 20: Grid of information gain with 256∗32∗256 cells, projected as in Figure 11f. Maximum information gain is shown normalized using a heat map visualization a) after 400, b) after 700 and c) after 1000 simulation steps.

the resulting values in the grid of information gain are presented in Figure 23b. 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 5 and depicted in Figure 24a. A collision-free path from start to goal cell is found and passed to the rover program in the form of waypoints, as seen in Figure 24b. Using a NVIDIA GeForce GTX 670, the complete process of motion planning 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 found given the grid's resolution is sufficiently large. Because our approach 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 20 presents the *normalized* information gain values of the point cloud depicted in Figure 11f at different steps 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 64^3m, 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.

7.3 Success and Failure Analysis

Testing point clouds for watertightness requires correctly tuned parameters - especially the ratio between point cloud density and particle size must be considered. 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.

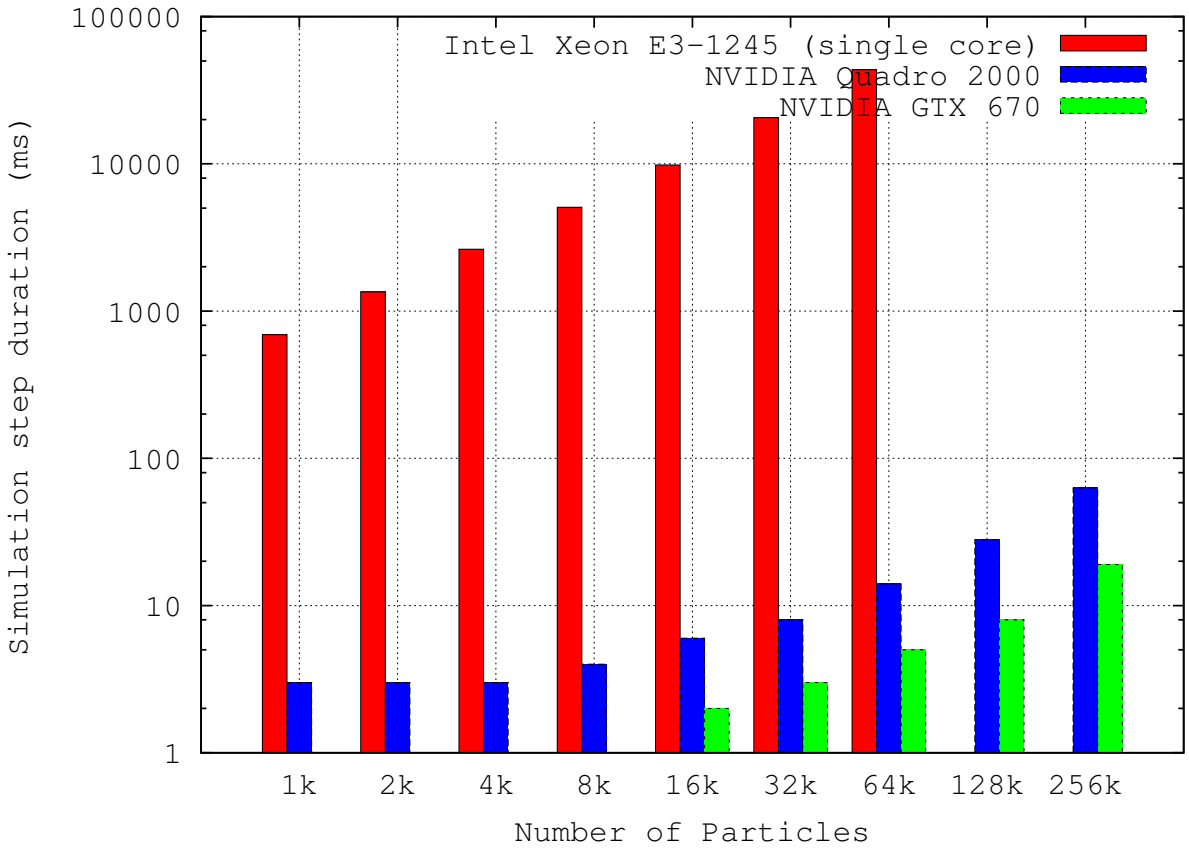


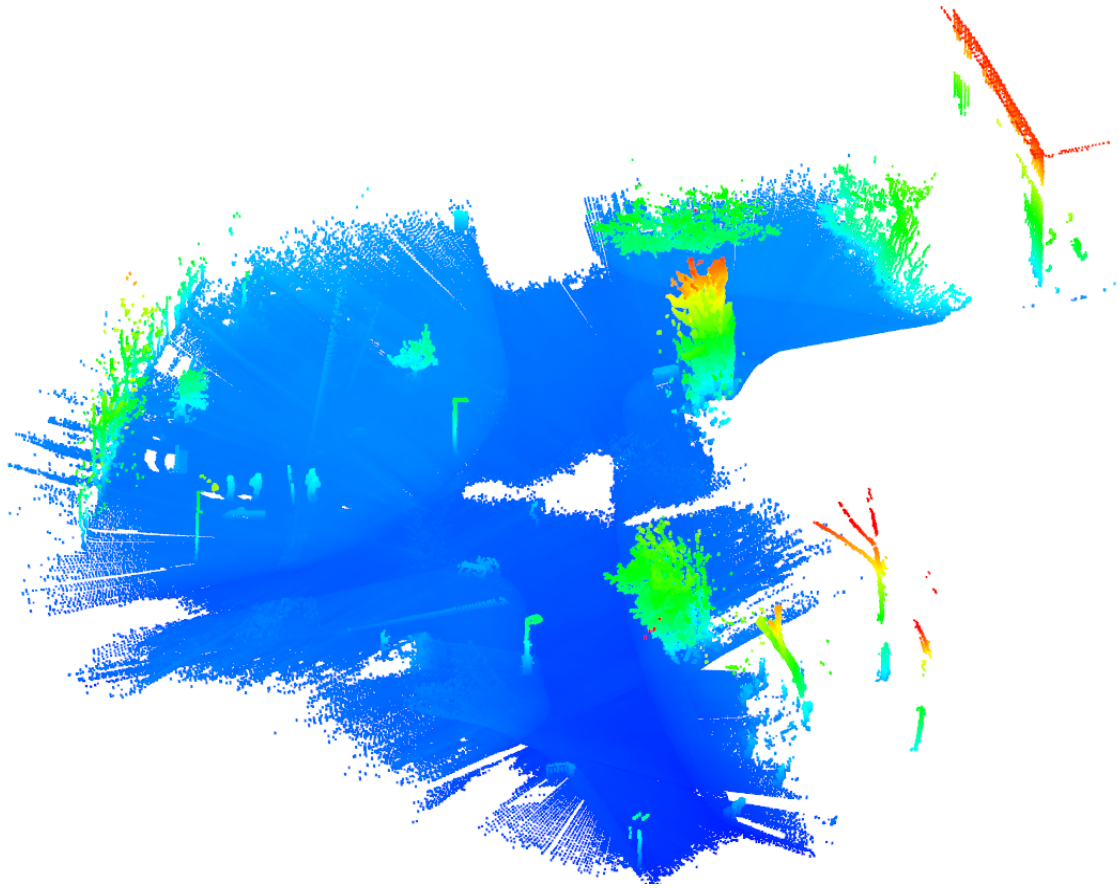
Figure 21: This graph shows the *maximum* time required for a single time step of the simulation containing 16k colliders against different numbers of particles. Visualization was disabled in all tests, note the logarithmic scale of the ordinate axis. The collision stage took 89.1% of the shader-processor's 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.

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. For this reason, we consider the approach probabilistically and resolution-complete.

7.4 Scalability

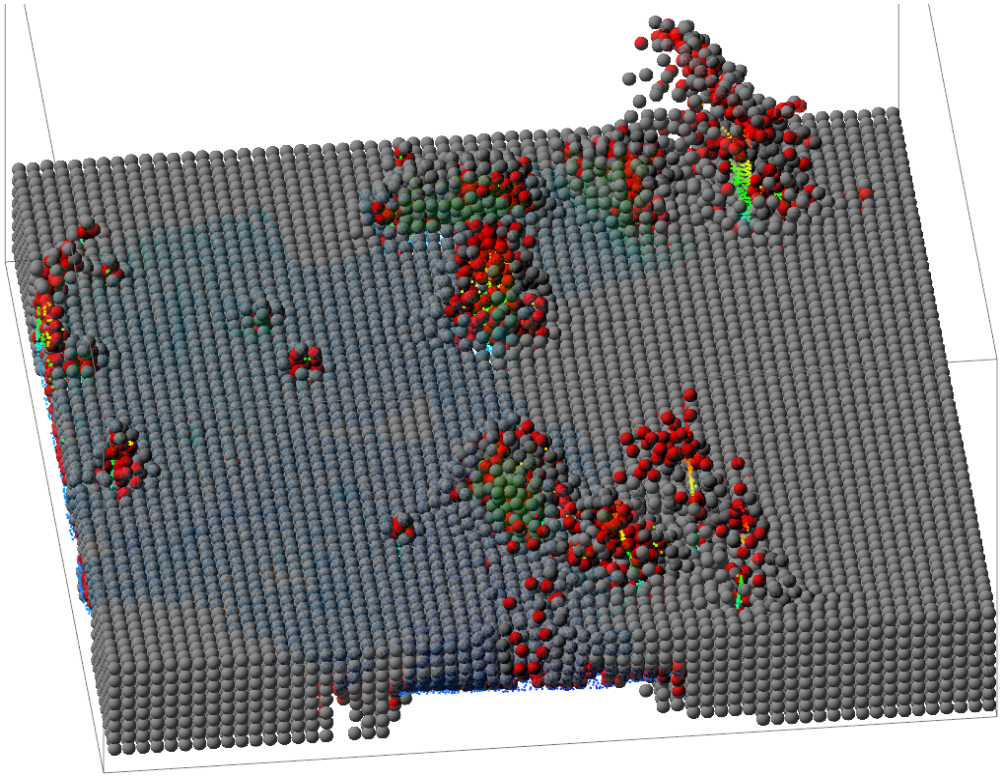
Within the particle simulation, most routines like radix sort, construction of cell lookup tables and particle integration run in linear time with respect to the number of particles, while collision-detection can become $O(n^2)$ in worst case. Processing n points from the sparse point cloud into an occupancy grid of resolution r takes O(n) time and O(r^3) space, while path planning through the grid takes O(r^3) time and space. To improve scalability, waypoint generation, path planning and obstacle avoidance have been designed to work within the constraints of a bounding box b. Independent of how the number of particles and colliders might be able to scale with future GPU generations, the approach is intrinsically scalable to larger terrains by simply shifting b to yet unexplored spaces 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.

(a)

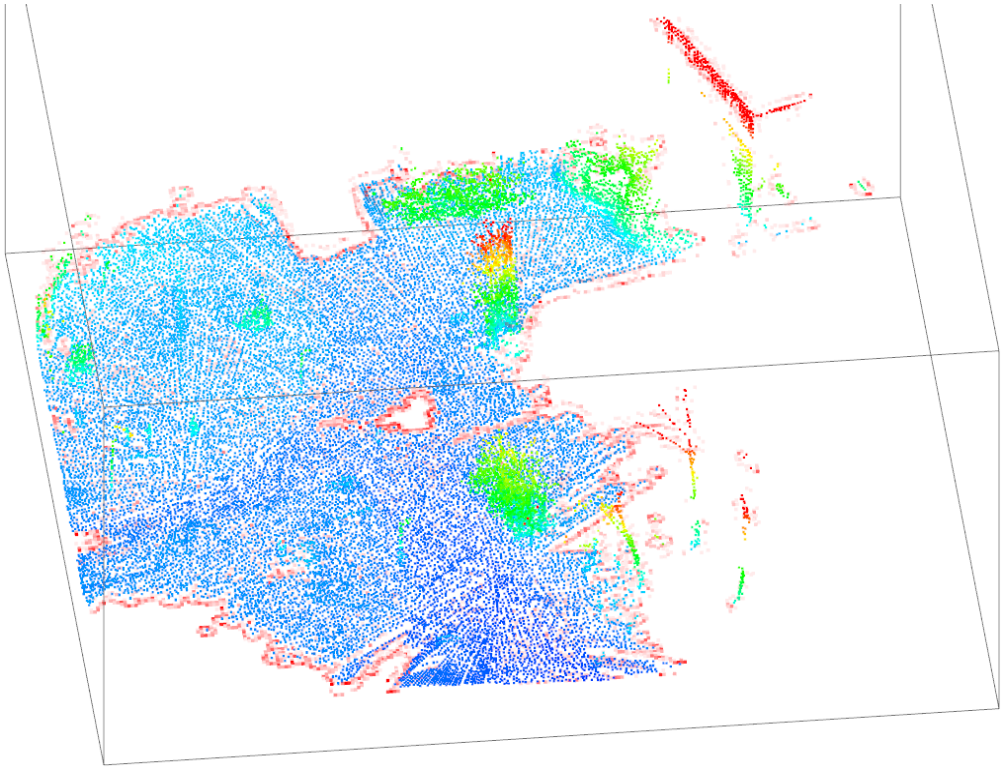


(b)

Figure 22: (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.

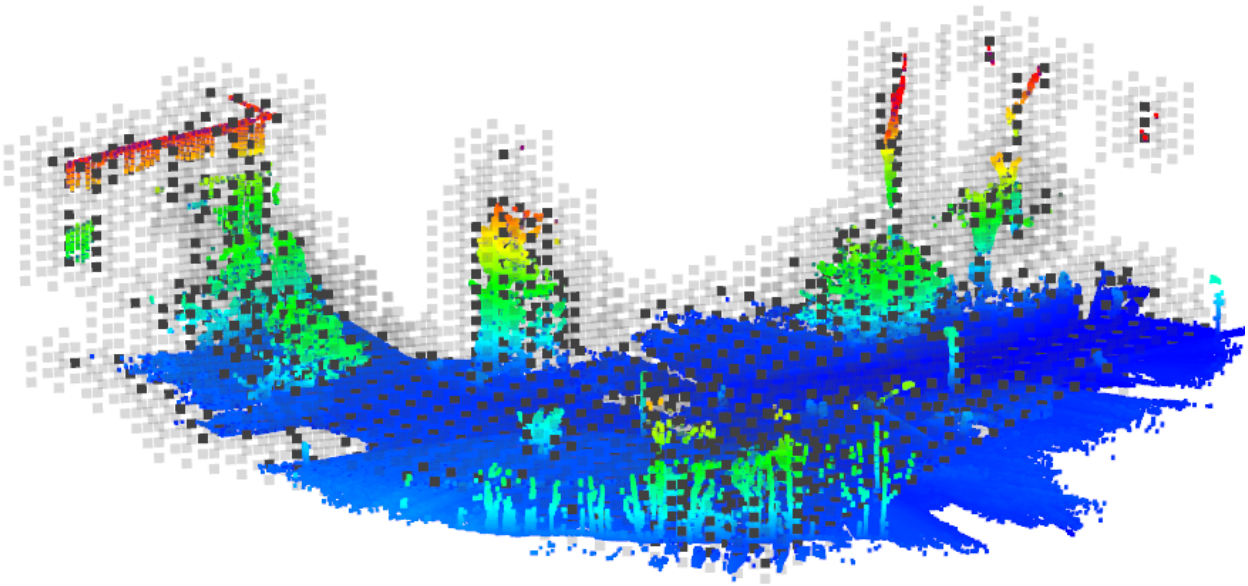


(a)

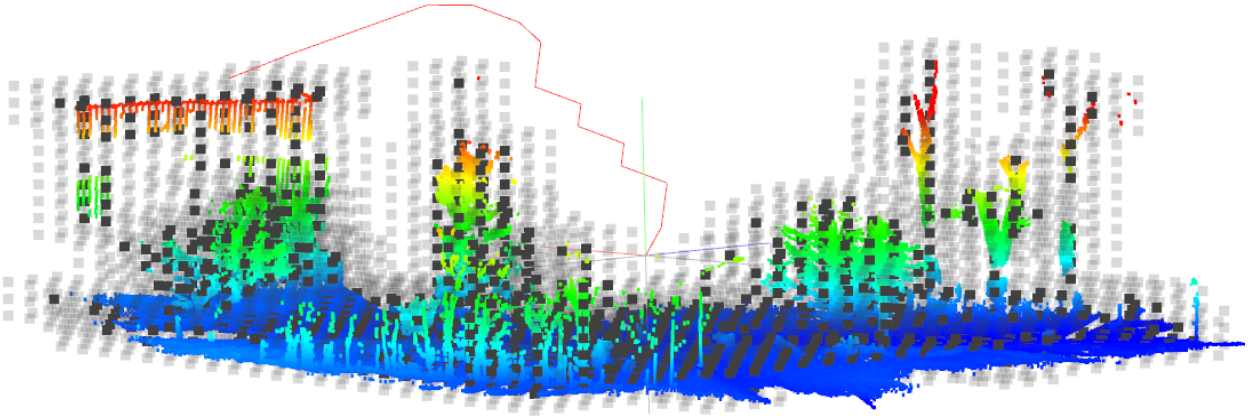


(b)

Figure 23: (a) 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 cells indicate higher information gain and are more likely to be processed into waypoints.



(a)



(b)

Figure 24: (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.

Table 4: 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	$N_{G_{IG}}$

7.5 Memory requirements

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 as well as the number of cells for the global grid of information gain $N_{G_{IG}}$. The respective data structures and their size are detailed in Table 4. As an example, for generating waypoints for a point cloud with 64k points using 128k particles, we use two spatial hash tables for particles and colliders with $N_{G_{SHT}} = 128 * 64 * 128$ cells each, while the computed information gain is stored in a grid with $N_{G_{IG}} = 256 * 32 * 256$ cells. This results in a total memory requirement of only 26.5 Mb.

8 Conclusion and Outlook

This paper has presented the following contributions:

- a light-weight and fast airborne platform, capable of quickly scanning urban environments by producing geo-referenced point clouds of configurable density.
- a novel algorithm finding multiple next best views, working directly on unorganized point clouds.
- an implementation of said algorithm that both exploits the enormous computational power available on todays massively parallel hardware and scales well with the number of available processors.
- a fast and simple GPU-based algorithm for processing the generated NBVs into collision-free sequences of waypoints

The authors plan to further improve the presented approach in terms of efficiency. Because most parts of the implementation are memory-bound, the impact of using half-floats (i.e. binary16 in IEEE 754 parlance) for at least the collider positions will 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 deemed sufficient for gap detection. The particle's collision positions will 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. The most promising separation of tasks 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 can 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, it is planned to compare performance to other cell-hashing functions that are expected to provide better locality than the currently used simple serial hashing function. Tests are currently being done with Hilbert- and Z-Order curves (Morton code), 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 this NBV planning algorithm in the system for autonomous exploration tasks. Especially for application on UAVs, extensions allowing anticipation of collisions between the robot and the fused point cloud are currently being researched. Whenever C is updated, one thread per cell can be employed to check potentially colliding points in every grid cell that is traversed by the planned trajectory within milliseconds. If at least one thread detects a potential collision, the path needs to be replanned.

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 *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1056–1061, Tokyo, Japan.

Blaer, P. and Allen, P. K. (2007). Data acquisition and view planning for 3-D modeling tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 417–422, Marina, San Diego, California, USA.

Bryson, M. and Sukkarieh, S. (2006). Active airborne localisation and exploration in unknown environments using inertial SLAM. In *IEEE Aerospace Conference*, page 13 pp.

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

Connolly, C. (1985). The determination of next best views. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 432–435.

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

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

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

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

Holenstein, C., Zlot, R., and Bosse, M. (2011). Watertight surface reconstruction of caves from 3D laser data. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3830–3837, San Francisco, CA, USA.

Makarenko, A., Williams, S., Bourgault, F., and Durrant-Whyte, H. (2002). An experiment in integrated exploration. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 534–539, Lausanne, Switzerland.

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

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 *10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 127–136, Basel, Switzerland.

Null, B. D. and Sinzinger, E. D. (2006). Next best view algorithms for interior and exterior model acquisition. In *Proceedings of the Second international conference on Advances in Visual Computing - Volume Part II*, ISVC'06, pages 668–677, Berlin, Heidelberg. Springer-Verlag.

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

Potthast, C. and Sukhatme, G. S. (2011). A probabilistic Framework for Next Best View Estimation in a Cluttered Environment. Available at: http://robotics.usc.edu/ potthast/vua2012.pdf, retrieved on January 13th, 2013.

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

Satish, N., Harris, M., and Garland, M. (2009). Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, Rome, Italy.

Scott, W. R., Roth, G., and Rivest, J.-F. (2003). View planning for automated three-dimensional object reconstruction and inspection. *ACM Computing Surveys*, 35(1):64–96.

Shade, R. and Newman, P. (2011). Choosing where to go: Complete 3D exploration with stereo. In *IEEE International Conference on Robotics and Automation*, pages 2806–2811, Shanghai, China.

Strand, M. and Dillmann, R. (2010). Grid based next best view planning for an autonomous robot in indoor environments. In *Proceedings of the Fourth International Workshop on Robotics for risky interventions and Environmental Surveillance-Maintenance*, RISE'2010, Sheffield, UK.

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.

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.

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