Towards Autonomous Airborne Mapping of Urban Environments

Benjamin Adler, Junhao Xiao
University of Hamburg
Fachbereich Informatik,
Technische Aspekte Multimodaler Systeme (TAMS)
Hamburg, Germany
adler,xiao@informatik.uni-hamburg.de

***Abstract*— This work documents our progress on building an unmanned aerial vehicle capable of autonomously mapping urban environments. This includes localization and tracking of the vehicle's pose, fusion of sensor-data from onboard GNSS receivers, IMUs, laserscanners and cameras as well as realtime path-planning and collision-avoidance. Currently, we focus on a physics-based approach to computing waypoints, which are subsequently used to steer the platform in three-dimensional space. Generation of efficient sensor trajectories for maximized information gain operates directly on unorganized point clouds, creating a perfect fit for environment mapping with commonly used LIDAR sensors and time-of-flight cameras. We present the algorithm's application to real sensor-data and analyze its performance in a virtual outdoor scenario.**

I. Introduction

Automatic model building has always been one of the most important parts of autonomous robotics. Exploration and mapping - as a subclass of this problem - has been the topic of many publications in recent years. The SLAM problem was first solved for two-dimensional mapping scenarios and later-on expanded to three-dimensional environments.

This paper presents a novel approach on increasing the efficiency of such mapping procedures. While the implementations of SLAM have improved greatly in recent years, one aspect of mapping three-dimensional environments was given comparatively little attention: finding the next best view. Typically, SLAM algorithms have been researched and developed on mobile platforms moving on flat surfaces [10], [11], [5]. This setup does not necessitate high efficiency in exploration, as it does not inherently impose time constraints. The authors of this paper, however, are implementing exploration and mapping algorithms on a flying platform with a maximum flight time of less than 15 minutes. Aiming to explore as much of the environment as possible within this limited time motivates an efficient way of finding sensor poses (and in extension, sensor trajectories) that enable sensors to deliver information as quickly as possible. While repeatedly passing over the unknown environment in scan-line fashion results in very fast scanning, a better approach needs to be found to ensure scanning of surfaces that are concealed from those sensor trajectories.

This paper is organized as follows: the next section investigates other authors work of recent years in this and related fields. In the subsequent chapters, we first introduce the hardware platform used for testing our implementations and then explain the basic data structures our algorithm relies on. After our algorithm is explained and illustrated, we discuss its computational complexity and the results achieved so far. In the final chapter, we share our thoughts on its current limitations and future developments.

II. Related Work

Since the introduction of SLAM, much work has been done to refine and extend the algorithm for use in both two- and three-dimensional environments. Finding the next best view, though, has never been a part of the SLAM problem [8] and is thus rarely addressed in these publications.

Generating safely reachable waypoints from previously acquired sensor data is an extension of the next-best-view problem (which itself is an extension of the art-gallery problem [13]), as the former does not include the constraints of safe navigation and instead deals with the generation of isolated viewpoints[3]. As sensor poses in areas between known and unknown environment offer a good compromise between safe reachability and high information gain, frontier-based approaches are a popular method to compute new waypoints in two-dimensional space and are well documented in the literature [8], [9], [11], [12]. Unfortunately, information about exploration boundaries is hard to generate from three-dimensional point clouds.

As our robot is very maneuverable, the generated waypoints and trajectories can be created with less concern of the platform's ability to catenate them due to motion constraints that other tyes of vehicles (e.g. fixed-wing UAVs) previously entailed [1]. The common problem of wind, manifesting as both opportunity and hindrance for many UAVs, has spawned many contributions in the past [4], [2]. Even though wind is still a matter of concern for our platform, it does no longer impose constraints on path planning.

III. Our approach

A. Platform

To save time during the construction of our platform, we used an "Okto 2"-octocopter from the mikrokopter-project as a base for our vehicle (see Fig. 1). Its central FlightControl (FC) processor-board is connected to eight brushless-motor controllers via an I^2C bus. Employing its on-board gyros and accelerometers, the FC is programmed to

Fig. 1. The experimental flying platform with mounted GNSS-antenna/receiver, processor-board, laser scanner and IMU.

stabilize the platform by itself when no other motion-control-commands are received from either the connected remote-control-receiver or its serial port. We also fitted an Intel Atom processor-board to the platform to process incoming sensor data and send navigation-commands to the FlightControl board.

A Septentrio AsterX2i RTK-GNSS (Real Time Kinematic Global Navigation Satellite System) receiver has been installed and connected to an XSens MTi IMU, enabling measurement of the platform's pose at 20Hz. After fusing the GNSS trajectory and IMU sensor-data, the resulting position is accurate to within 5cm while the orientation shows a maximum error of $\sim 1^\circ$ for pitch, roll and yaw angles.

The laserscanner is mounted looking downwards, scanning stripes orthogonal to the platforms heading during flight. Gathering information on obstacles in the vehicle's path necessitates a yawing movement, allowing our algorithms to detect and avoid potential collisions.

After balancing the platform's extra loads, it exhibits very favorable flight dynamics and barely drifts when idling. Including payload and a 5000mAh 4s1p LiPo battery, the platform weighs 2250 grams and requires around 350W of power while hovering, leading to a flight-time of about 12 minutes.

B. Data structures

Occupancy grid maps - and in extension elevation maps - are often used to store sensor data in generalized form [12]. Although the data structure permits easy traversability computation and frontier detection, storing more complex geometry or overhangs remains difficult or even impossible, limiting its practical use to two-dimensional environments. Furthermore, its uniform grid size requires a global compromise between model quality and memory consumption. Multi-Level surface maps, introduced by Triebel et al. [14], eliminate many of these limitations, while still being constrained to a grid.

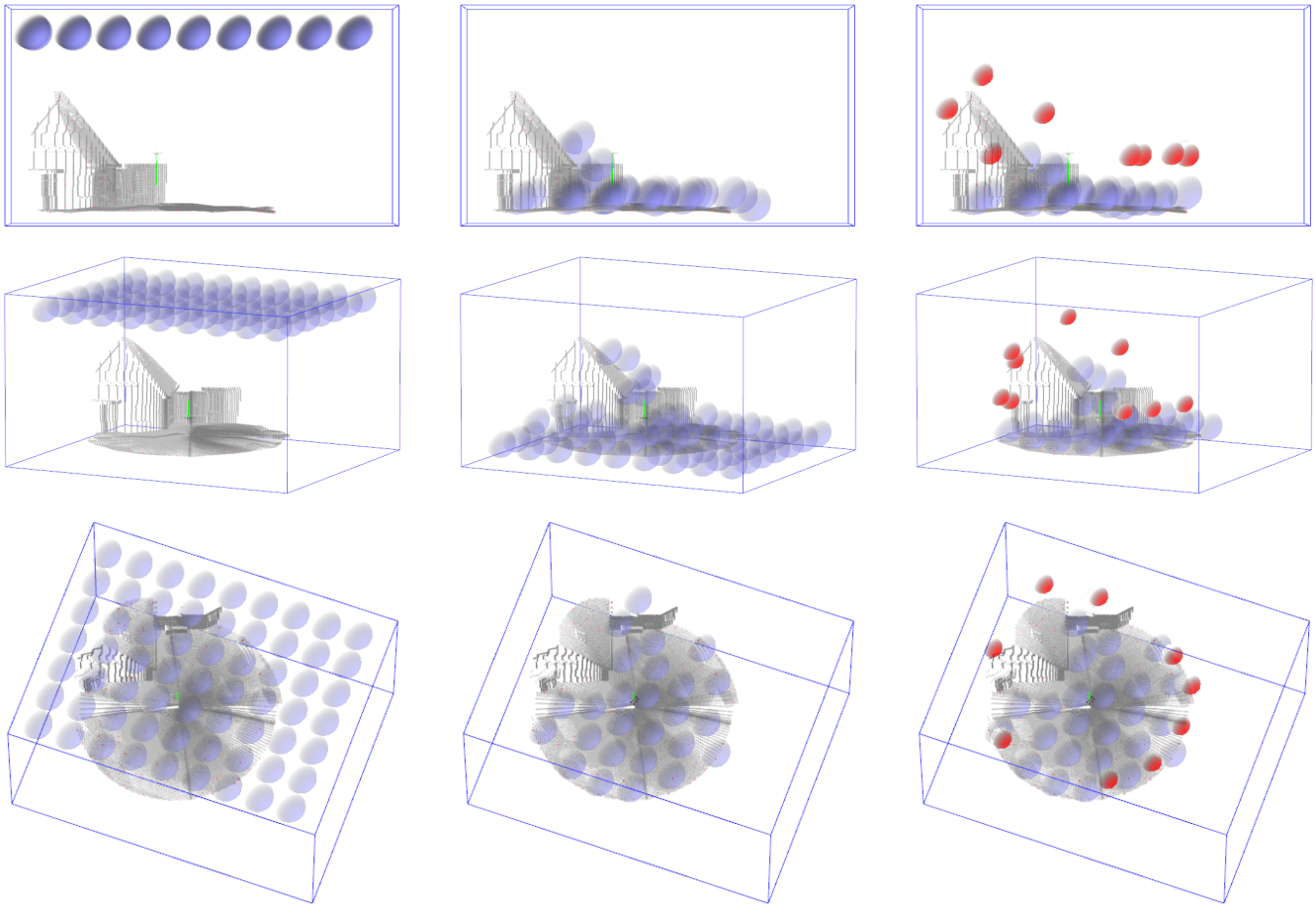
In order to capture sensor data representing arbitrarily complex geometry and detail, we decided to implement our algorithm on an octree-based data-structure. Octrees are well suited for storing information in non-uniform resolution and the process of storing and retrieving data can easily be executed in parallel, allowing for optimization of parts of our algorithms. Each point in the octree also stores a vector pointing from that point back to the sensor's position at the time of its capture. This vector's length will be used lateron to assign points recorded from further distance (and thus suffering a greater impact of platform-orientation errors) a lesser weight in the following surface reconstruction.

C. Algorithm

Three-dimensional environment mapping is often implemented using laser scanners, so unorganized point clouds are a very common type of sensor data. Any sensor generating spatial occupancy information can be used with our algorithm, as the point cloud is its only sensory input. Finding the next best view in such data can be very hard, as it does not supply any information about geometric structures such as corners, edges, surfaces and normals. In contrast to many other algorithms ([5], [10]), ours does not require such information.

Prior to the mapping process, the user creates a bounding volume as a representation of the region-of-interest around the vehicle's starting position, thereby defining the environment that is to be mapped. All generated viewpoints will be constrained to this volume and its vicinity, ensuring that the vehicle does not leave the scene. During initialization, a physics engine is set up so that objects colliding with the bounding volume's bottom plane trigger an event handler described below. Each new point in the point cloud is automatically registered as a static collision object in the physics world. After the octree is populated with an initial set of points (i.e. right after lift-off), the waypoint generation algorithm (Algorithm 1) is executed as depicted in Fig. 2:

1) Initialize sample geometry (line 5): spheres of radius sr_c are spawned evenly along the bounding volume's top plane. Increased sample geometry size will create fewer instances, less computational burden (see section III-D) and less waypoints, while smaller sample geometry will find smaller gaps in the sampled surface.
2) Find gaps: execute the physics simulation with small timesteps Δt, allowing the sampling geometry to fall (l 25) and collide with other sampling geometry (l 30) and the static collision objects making up the scanned point cloud (l 36). Whenever a sample sphere collides with the point cloud, that event is saved into map_c, associating that sphere to the position of only its most recent collision with the point cloud (l 38). Because the data structure is implemented as a map, repeated collisions of the same sphere with the pointcloud will replace its previously saved position of collision. Since the last contact indicates a gap in the point cloud big enough to let pass a detection sphere or the border of the point cloud, this position is regarded as a waypoint candidate.
3) Whenever a sample geometry instance collides with the bounding volume's bottom plane, that instance is removed from this run of the simulation (l 27) and



(a) (b) (c)

Fig. 2. Waypoint generation, seen from three different viewpoints; perspective projection. a) depicts the initial setup: the bounding volume and sample geometry are drawn in blue, while the point cloud is drawn in grey. b) shows sampling geometry interacting with the point cloud. In c), the last collision positions of detection spheres are converted to waypoint candidates (yellow) after having reached the bounding volume's bottom plane. Please see the attached video for an animation of the process.

the associated waypoint candidate - if present - is processed further in step 4).

4) Waypoint candidates are converted to waypoints if there are no other waypoints in close vicinity (omitted from pseudo-code). During conversion, their height needs to be adjusted, as waypoint candidates tend to spawn on the scenery's ground. The height h, by which waypoint candidates are elevated before becoming waypoints, is calculated by $h = Range_{Lidar} * \frac{sr_c}{sr_i}$ (l 10). By making the distance of generated waypoints to the detected gaps in the surface depend on the current size of the sampling geometry, a natural behavior emerges: The vehicle will first scan larger voids in the surface from further away. When the sampling geometry's size is reduced in later iterations of the waypoint generation, it will fly closer passes to fill smaller gaps that couldn't be observed from previous sensor poses.
5) Each waypoint generation run stops when all sampling geometry has either been deleted after hitting the bounding box's bottom plane or come to rest on the pointcloud (to ignore objects that move slowly, but indefinitely due to numerical instabilities in the solver employed by the physics engine, we implemented a threshold-based termination criterion, as seen in line 40). After all generated waypoints have been passed by the UAV or if no waypoints could be generated, the sample geometry radius sr_c is decreased and the algorithm is restarted at 1). This can be repeated as long as the sample geometry's cross section remains larger than the gaps between points of well-scanned surfaces in the pointcloud (see $MinDistNeighbor$ in section III-D).

Depending on size, number and shape of sampling geometry, our approach simulates exposing the scanned surface to a fluid pushing through the model. The idea of generating watertight models from previously generated datasets has been documented before [17], [16], so the novelties here are:

- testing the currently collected data for watertightness in-flight and using the result as a termination criterion for exploration.
- testing for watertightness using a physical simulation

Algorithm 1 Non-parallel version of waypoint generation

Input:
Gravity G, LidarRange LR, Pointcloud PCD,
Bounding Box BB,
Sphere-radius: (initial sr_i / current sr_c)

function FINDWAYPOINTS(G, LR, PCD, BB, sr_i)
  $map_s(sphere \mapsto (pos \in \mathbb{R}^3, vel \in \mathbb{R}^3)) \leftarrow 0$,
  $sr_c \leftarrow sr_i$
  repeat
    $map_s \leftarrow$ RESETSPHERES(sr_c)
    $gaps \leftarrow$ FINDGAPS(PCD, G, BB, map_s, LR)
    if $gaps.size() > 0$ **then**
      for all $gap \in gaps$ **do**
        $// \ raise \ waypoints$
        $gap.height \mathrel{+}= LR * \frac{sr_c}{sr_i}$
        FLYTO(gap)
      end for
    else
      $// \ no \ gaps \ found, \ decrease \ sphere \ size$
      $sr_c \leftarrow \frac{sr_c}{2}$
    end if
  until $sr_c <= \frac{1}{2} * PCD.MinDistNeighbor$
end function

function FINDGAPS(PCD, G, BB, map_s, LR)
  $map_c(sphere \mapsto CollisionPos \in \mathbb{R}^3) \leftarrow 0$
  repeat
    $maxVelocity \leftarrow 0 \ // \ to \ ensure \ termination$
    for all $s \in map_s.keys()$ **do**
      $// \ update \ position/velocity$
      $s.integrateMotion(G, \Delta t)$;
      if $s.collidesWith(BB.bottomPlane)$ **then**
        $map_s.remove(s)$
      else
        $// \ collide \ s \ with \ other \ particles$
        for all $t \in map_s.keys()$ **do**
          if $s.collidesWith(t) \wedge s \neq t$ **then**
            $s.vel \leftarrow s.collide(t)$
          end if
        end for
        $// \ collide \ s \ with \ point \ cloud$
        for all $p \in PCD$ **do**
          if $s.collidesWith(p)$ **then**
            $map_c.insert(s, s.pos)$
            $s.vel \leftarrow s.collide(p)$
          end if
        end for
      end if
      $maxVelocity = max(s.vel, maxVelocity)$
    end for
  until $maxVelocity < threshold$
  return $map_c.values()$
end function

and its real-time implementation directly on sensor-data without the need for additional pre-processing.
- re-weighing the compromise between exploration time and quality of the resulting model by changing sampling geometry size quickly and in-flight.
- the algorithm's gracefully degraded behavior when size and number of sampling spheres cannot be scaled to achieve simulation with near-fluid behavior due to constrained computational resources.

Although any geometry can be used to detect voids in the surface, using spheres yields three important advantages:

- Spheres can be represented by only radius and position, reducing the overall memory requirement (especially when all spheres share a common radius).
- Collision detection between points and spheres is a process of low computational complexity, rendering many tasks in collision detection's broad-, mid- and narrowphase redundant. To detect a collision, it is sufficient to check whether the distance between the sphere's center and the point is smaller than the sphere's radius. The sphere's orientation is of no concern during collision detection, reducing computational complexity even further.
- Most importantly, due to their shape, spheres are ideally suited to slip through smaller voids in the pointcloud, facilitating detection of regions requiring another scan-pass.

D. Computational complexity

The algorithms complexity derives from the computational effort of the collision detection phase in physics simulation and thus from the number of collision objects. Collision detection between n objects in general requires $n(n-1)/2$ collision checks to be performed, leading to a complexity of O(n^2). Optimized algorithms like sort-and-sweep [7] or implementations relying on spatial subdivison may reduce complexity to O(n log n) in favorable cases [6]. Nevertheless, this initially appears to be a major obstacle to deployment of the algorithm outside of simulation, as the pointcloud may grow to several million points during flight. After the following optimizations, however, the algorithm has proven to be sufficiently fast for real-time application on current-generation CPUs.

- Given a pointcloud consisting of n points and a set of m sample geometries, the algorithm does not execute collision tests between all $n+m$ objects. As the n points making up the accumulated scan data are static, the number of required tests is reduced to $\frac{m(m-1)}{2} + n * m$. Because $n \gg m$, this optimization yields a considerable loss in computational effort.
- When using the sensor data solely for waypoint generation, the density of the pointcloud can be reduced to allow gaps almost the size of the sampling geometries' cross section. That is, if sample spheres are created with a radius of r, the octree storing that pointcloud rejects insertion of points if there are neighbors within

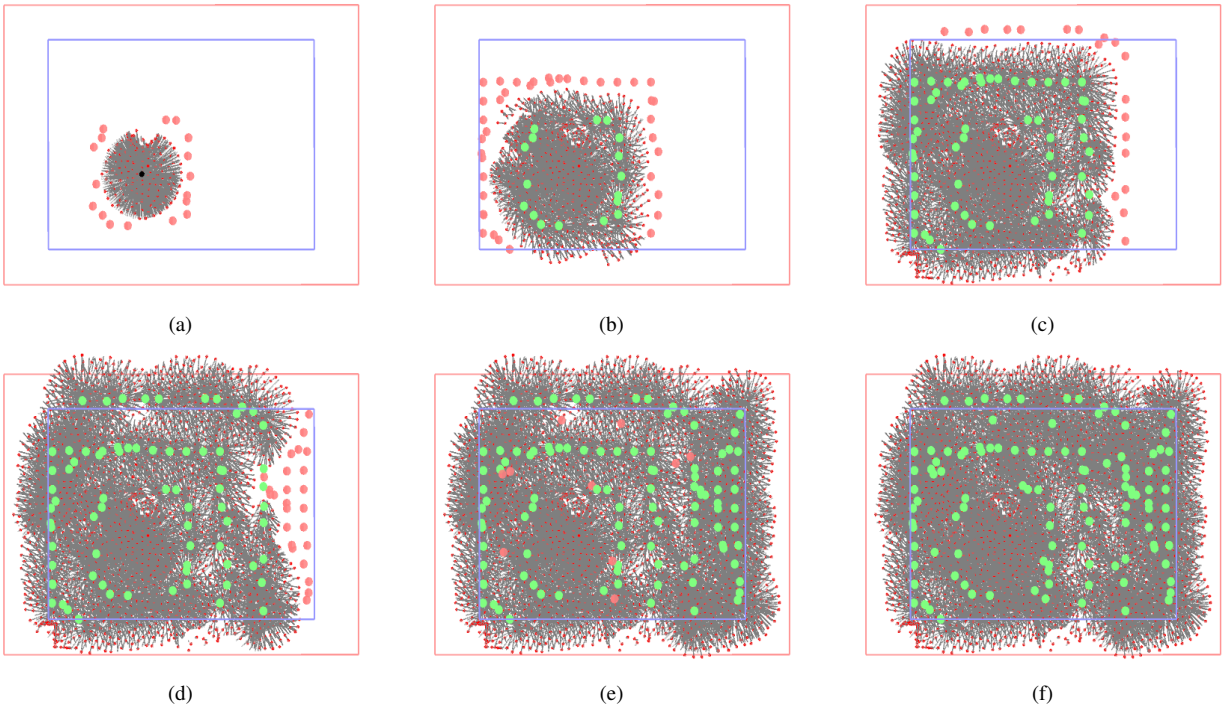


Fig. 3. As seen from the top, multiple iterations of waypoint generation lead to continuous scanning at the border between known and unknown environment. Red waypoints are enqueued for scanning, green waypoints have been passed. In figures a) to d), sample spheres with $r = 3m$ have been used to quickly explore the landscape, while in figure e), sampling geometry of reduced size ($r = 1.5m$) has allowed finding smaller holes in previously scanned surfaces.

a distance of $d << 2r$ ($MinDistNeighbor$ in the pseudo-code). This is implemented using neighbor-queries during insertion of candidate points and - while causing a higher computational complexity in that phase - yields a reduction of n by about two dimensions in practice, dramatically reducing the number of collision pairs that have to be checked in every step of the physics simulation. For this reason, we use two pointclouds in our work; one is used for surface reconstruction and is stored in an octree which allows for close neighbors and high density (depicted by grey points in the figures) and another pointcloud used for waypoint generation and collision avoidance, depicted by red points in the figures.

IV. Results

To assess our algorithms performance, we established the number of points stored in the pointcloud for a given flight-time as the primary metric. As explained in chapter III-D, our octree-implementation allows setting a maximum point-density during construction. For the pointcloud storing the surface-reconstruction data, we defined the minimum distance between neighboring points to be $0.1m$, so that measuring the number of points stored over time is equivalent to measuring the scanned surface area over time. All tests were executed in a simulator, with the platform's linear velocity limited to $2.8m/s$ during all trials. While the scanned points were streamed to the basestation, waypoints were generated and statistics about flight-time and pointcloud sizes were logged.

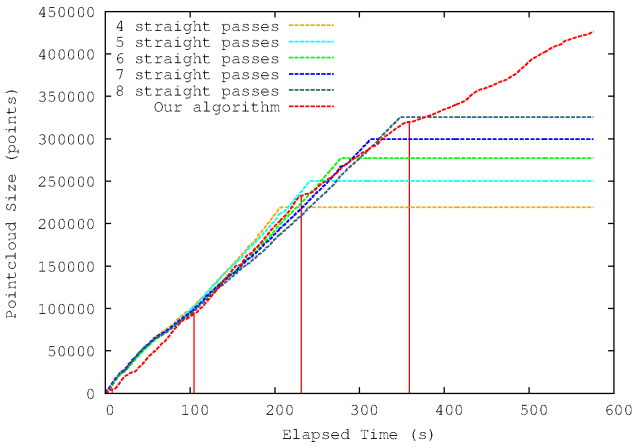


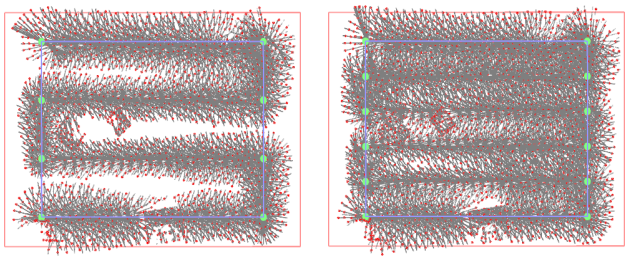
Fig. 4. Pointcloud size vs. flight-time using several waypoint generators. Vertical lines are drawn on every round of waypoint generation using our algorithm, executed after all previously generated waypoints have been passed.

The graph in Fig. 4 shows results of using a simple, manually planned and collision-free scanline based exploration with trajectories similar to those proposed in [15] for optimal complete terrain coverage. The results of scanning along these trajectories can be seen in Fig. 5(a) and 5(b) for four and six equidistant passes, respectively. As expected, flying more scanlines in the same area results in more points

scanned, albeit with a slightly slower rate.

Additionally, the graph's red line shows our algorithms' performance: because the pointcloud needs an initial population for our algorithm to start, the scanning rate stagnates shortly after launch during the first phase of waypoint generation, then reaches comparable speeds as the vehicle starts passing waypoints. In this phase of flight, our algorithm is as efficient as a manually planned collision free optimal path, with the obvious advantage of having been generated automatically. Compared to exploring in scanline-fashion, the true strength in our method lies in the fact that on-line replanning occurs such that when no more waypoints can be generated, gradually smaller sampling geometry is used to improve on ever smaller deficiencies in the reconstructed surface. This yields a truly dynamic and efficient scanning procedure that adapts well to many different kinds of outdoor scenarios.

During 160 seconds of flight on our university campus, our vehicle registered 815k points in an area of 80 by 60 meters. Of these points, 3121 were used in the sparse octree for collisions with sample geometry. After spawning 475 sample spheres with a radius of 1.5m, our algorithm took 7784 milliseconds to compute 71 waypoints on a single core of an Intel Xeon CPU running at 3.30GHz. Since waypoints can be computed much faster than the platform is able to reach them, our approach can be used for dynamic, in-flight path-planning.



(a) 4 passes (b) 6 passes

Fig. 5. Pointclouds resulting from straight scanline passes. Note the unscanned surface in the bottom center shadowed by a roof.

V. Outlook

With the algorithm performing collisions between large numbers of static and dynamic objects, it is currently limited by the speed of collision detection. Physics-simulation in general and collision detection especially lend themselves well to optimization using massively-parallel implementations on GPUs. The authors are currently porting the CPU-based implementation of the algorithm to CUDA in order to leverage the immense performance gains made possible by highly-parallel execution. As described by Le Grand [6], execution of the simulation's broadphase on the GPU can speed up collision detection by more than an order of magnitude compared to calculation on the CPU.

Architecture featuring horizontal planes that shield the ground below (e.g. carport) has proven problematic with the current approach, because the sampling geometry's size and number did not always allow for traversal through the remaining gaps. Porting our algorithm to the GPU should allow completely filling the user-defined bounding box with smaller sample geometries while still adhering to real-time compatible time constraints. We hope to show that the more fluid-like behavior of the sample geometries resulting from this change will render gap detection below difficult geometries possible.

Finally, we plan to move our research's focus from waypoint generation to the problems of generating safely navigatable paths between them, dynamically replanning paths when necessary and discarding waypoints that cannot be reached safely.

References

[1] Sikha Hota and Debasish Ghose, A Modified Dubins Method for Optimal Path Planning of a Miniature Air Vehicle Converging to a Straight Line Path, American Control Conference, St. Louis, MO, USA, 2009

[2] Rachelle L. McNeely and Ram V. Iyer and Phillip R. Chandler, Tour Planning for an Unmanned Air Vehicle under Wind Conditions, Journal of Guidance, Control, and Dynamics, Vol. 30, No. 5, September-October 2007

[3] Shengyong Chen and Y. F. Li and Jianwei Zhang and Wanliang Wang, Active Sensor Planning for Multiview Vision Tasks, Springer Verlag Berlin Heidelberg, 2008

[4] Timothy G. McGee and Stephen Spry and J. Karl Hedrick, Optimal path planning in a constant wind with a bounded turning rate, Proc. of the AIAA Guidance, Navigation and Control Conference and Exhibit, San Francisco, CA, August 2005.

[5] Dominik Joho and Cyrill Stachniss and Patrick Pfaff and Wolfram Burgard, Autonomous Exploration for 3D Map Learning, *Autonome Mobile Systeme (AMS)*, Springer, 2007, pp. 22-28

[6] Scott Le Grand, Broad-Phase Collision Detection with CUDA, *GPU Gems 3,* (2007)

[7] David Baraff, Dynamic Simulation of Non-Penetrating Rigid Bodies, Cornell University, pp. 52 (1992)

[8] Héctor H. González-Baños and Jean-Claude Latombe, Navigation Strategies for Exploring Indoor Environments, *I. J. Robotic Res.* 21(10-11), 829-848 (2002)

[9] Marcus Strand and Ruediger Dillmann, Grid based next best view planning for an autonomous robot in indoor environments

[10] Axel Walthelm and Amir Madany Mamlouk, Multisensoric Active Spatial Environment Exploration and Modeling (2011)

[11] Brian Yamauchi, Frontier-Based Exploration Using Multiple Robots, Proceedings of the Second International Conference on Autonomous Agents, Minneanapolis, ACM Press, 1998

[12] Robbie Shade and Paul Newman, Choosing Where To Go: Complete 3D Exploration With Stereo, IEEE International Conference on Robotics and Automation, 2011

[13] Joseph O'Rourke, Art Gallery Theorems and Algorithms, Oxford University Press, Oxford, 1987

[14] Rudolph Triebel et al., Multi-Level Surface Maps for Outdoor Terrain Mapping and Loop Closing, Proceedings of the 2006 IEEE/RSJ IROS 2006

[15] Anqi Xu and Chatavut Viriyasuthee and Ioannis Rekleitis, Optimal Complete Terrain Coverage Using an Unmanned Aerial Vehicle, IEEE International Conference on Robotics and Automation, 2011

[16] Mario Sormann and Christopher Zach and Joachim Bauer and Konrad Karner and Horst Bishof, Watertight Multi-view Reconstruction Based on Volumetric Graph-Cuts

[17] Alexander Hornung and Leif Kobbelt, Robust Reconstruction of Watertight 3D Models from Non-uniformly Sampled Point Clouds Without Normal Information