Finding Next Best Views for Autonomous UAV Mapping through GPU-Accelerated Particle Simulation

Benjamin Adler, Junhao Xiao and Jianwei Zhang[1]

Abstract— **This paper presents a novel algorithm capable of generating multiple next best views (NBVs), sorted by achievable information gain. Although being designed for waypoint generation in autonomous airborne mapping of outdoor environments, it works directly on raw point clouds and thus can be used with any sensor generating spatial occupancy information (e.g. LIDAR, kinect or Time-of-Flight cameras). To satisfy time-constraints introduced by operation on UAVs, the algorithm is implemented on a highly parallel architecture and benchmarked 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/.**

I. Introduction

For many real-world applications, 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 successful implementation of localization and mapping for a UAV, our research focus has been on 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 [1].

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.

This paper is organized as follows: In the next section, we present a short overview of work in this and related fields. Following, we briefly introduce our UAV and the idea behind our approach, and then explain the data structures and algorithms necessary to generate waypoints providing high information gain. In section VI, we show waypoints generated for point clouds captured during real flights and analyze the real-time applicability of the algorithm. Finally, we share our thoughts on current limitations and future developments.

[1]Benjamin Adler, Junhao Xiao and Jianwei Zhang are with the Institute of Technical Aspects of Multimodal Systems, Department of Computer Science, University of Hamburg, Germany. {adler,xiao,zhang}@informatik.uni-hamburg.de

Videos showing the application of the presented approach for real-time generation of multiple NBVs are available at http://tams.informatik.uni-hamburg.de/videos/.

II. Related Work

The art gallery problem has been researched extensively, with most contributions presenting algorithms operating on polygons in two-dimensional space [2]. Because the problem statement presumes the map to be known a-priori, these works do not present sufficient solutions for NBV planning in unknown evironments. There have been multiple publications surveying active perception planning for reconstruction and inspection, the most recent being [3], which classifies methods as either surface-based, volume-based or global. We detect missing information using a surface-based approach, while rating possible information gain of sensor poses using volumetric data structures.

Reasoning over yet-unexplored spaces using probabilistic methods, like in [4] 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. The authors of [5] define a mutual information gain resulting from travelling 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". Approach [6] is used for ground-based outdoor reconstruction, based on building voxel-grids from acquired data and ray traversal for NBV computation, but requires "a 2-D map with which it plans a minimal set of sufficient covering views". A frontier-based approach is commonly used in NBV problems [7], [8], 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" [9].

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.

III. Experimental Platform

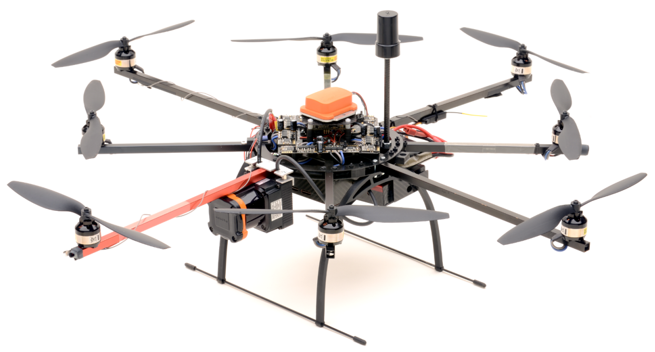

Fig. 1. The experimental flying platform with mounted GNSS-antenna and -receiver, laser scanner, IMU and processor-board.

The UAV consists of an "Okto 2"-multicopter from the mikrokopter project (see fig. 1). For self-localization, it is equipped with a commercial INS system, featuring a MEMS IMU and a dual-frequency GNSS[1] RTK[2] receiver supporting GPS and GLONASS constellations. Given sufficient satellite reception, the system's position is solved to a precision of 2 centimeters, while roll and pitch angles exhibit maximum errors of 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° after initial alignment. A Hokuyo UTM-30LX laser range finder is mounted below the vehicle's forward arm with its front side facing downwards. It is connected to an onboard-computer that fuses data from both sensors in real-time, creating a point cloud that is streamed to the ground station during flight using an IEEE 802.11n wireless network. For a more detailed description of the platform's hardware, see [10].

IV. Approach

The algorithm is inspired by other researcher's contributions concerned with creating watertight 3D models of real-world environments: watertightness is not only a desireable property for completely reconstructed models, but also a helpful test for finding gaps that have remained throughout the mapping process. The challenge here is to find gaps of a desired minimum size in a small amount of time.

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. After the collider-cloud C has been populated with an initial, downsampled set of N_C points $(c \in C)$ from the onboard laser scanner, gaps are detected by using a particle system which simulates pouring water in the form of N_P particles $(p \in P)$ over C (see fig. 2(b)). 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. Even though being designed for application on a UAV, the implementation can be applied to point clouds captured by any platform.

This algorithm has previously been implemented in a similar form using the Bullet physics library: 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 too slow to handle the number of collisions necessary for rapid detection of small gaps. Thus, the algorithm was ported to CUDA running on a graphics card.

A. Implementation on the GPU

To simulate N_P particles being poured over a point cloud of N_C points, the algorithm requires multiple data structures to be allocated in the graphics card's memory. The basic structures are depicted in fig. 3.

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
...	...	...	...	...	...

Fig. 3. Four vectors of float4 are allocated, storing N_C collider positions C_{pos} as well as N_P particle positions P_{pos}, particle velocities P_{vel} and particle 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.

To speed up collision detection, further data-parallel primitives have been utilized: as a spatial decomposition technique, two spatial hash tables (see fig. 4) using a common 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

[1]Global Navigation Satellite System

[2]Real Time Kinematic

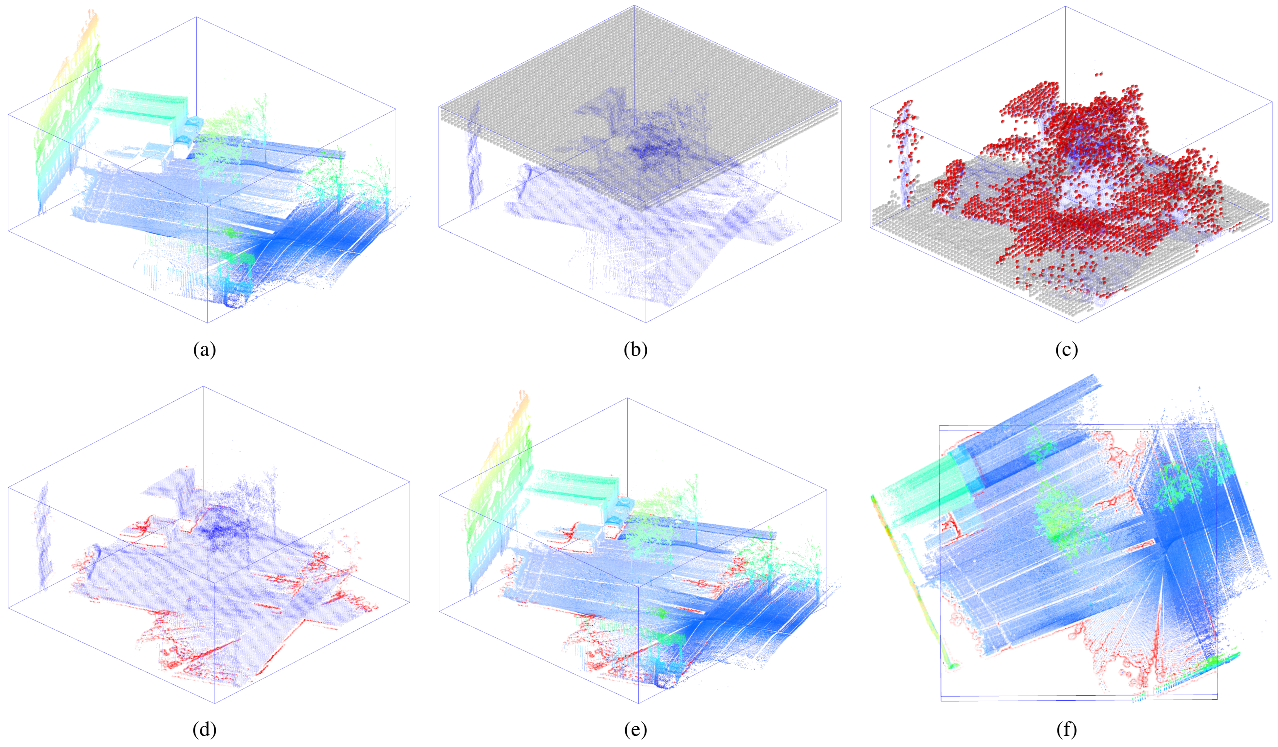

Fig. 2. Overview of the process: a) shows predefined bounding box b and the initial point cloud from the onboard laser scanner. b) shows 16k particles $p \in P$ in gray being poured over downsampled cloud of colliders $c \in C$ in blue. c) depicts falling particles $p \in P$ that have collided with any $c \in C$ in red, others remain gray. d-f) overlay a visualization of G_{IG} over sparse and dense point clouds, showing cells with high information gain in red.

enable efficient access to both particles P and colliders C. In the following paragraphs, we describe only the spatial hash table storing the particles, since the second table storing the colliders is organized in the same fashion.

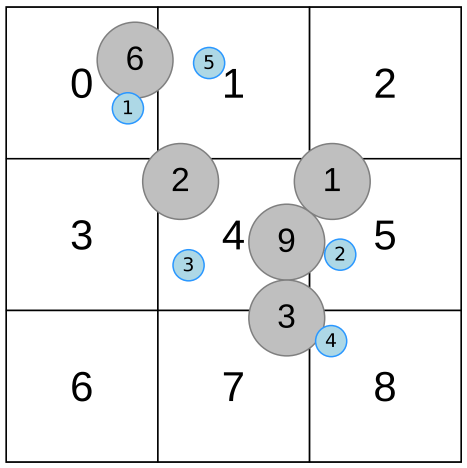

Fig. 4. Particles $p \in P$ (shown in gray) and colliders $c \in C$ (i.e. points from downsampled point cloud, blue) located in cells of a uniform grid (showing a two-dimensional grid for clarity). Cells are labeled using their hash values, while particles and colliders are labeled with their index in P_{pos} and C_{pos}, respectively.

The SHT requires two vectors P_{index} and $P_{cellhash}$, associating the particle's index in P_{pos} with the hash of G_{SHT}'s cell containing it. This is shown in fig. 5.

In order to quickly find all particles within a given grid cell during the following collision stage, the particle vectors shown in fig. 3 are sorted according to the hash value of their containing cell in G_{SHT}. Afterwards, a vector $CellStart$ of $N_{G_{SHT}}$ integers is allocated, where $CellStart[h]$ stores the index of the first particle in P_{pos} that is contained in the grid cell with hash h. If the cell does not contain any particles, its value is set to $UINT_{max}$. In analogous fashion, $CellStop$ is allocated, and $CellStop[h]$ stores the index of the last particle in P_{pos} that is contained in the grid cell with hash h. An example is shown in fig. 6.

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

Fig. 5. The resulting spatial hash table for particles contains N_P entries and is sorted according to cell hash. This data is used to build the cell lookup table presented in fig. 6

B. Algorithm

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 threadId into $P_{index}[threadId]$ and the hash of the cell con-

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:     $P_{index}[i] \leftarrow i$
4:     $P_{cellhash}[i] \leftarrow$ 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:     $sharedHash[i+1] \leftarrow P_{index}[i]$
12:     SYNCHRONIZETHREADS( )
13:     if $sharedHash[i] \neq sharedHash[i-1]$ then
14:         $P_{cellStart}[P_{cellhash}[i]] \leftarrow i$
15:         if $i > 0$ then
16:             $P_{cellEnd}[sharedHash[i+0]] \leftarrow i$
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:     $force \leftarrow (0,0,0)$
27:     $cellHashes \leftarrow$ GETNEIGHBORHASHES($P_{pos}[i]$)
28:     for each $h \in cellHashes$ do
29:         // collide p against colliders
30:         for $j \leftarrow C_{cellStart}[h], C_{cellEnd}[h]$ do
31:             $force$ += COLLIDEDEM($P_{pos}[i], C_{pos}[j]$)
32:         end for
33:         // save particle pos in case of collider-collision
34:         if $force \neq (0,0,0)$ then
35:             $P_{colpos}[i] \leftarrow P_{pos}[i]$
36:         end if
37:
38:         // collide p against other particles
39:         for $j \leftarrow P_{cellStart}[h], P_{cellEnd}[h]$ do
40:             $force$ += COLLIDEDEM($P_{pos}[i], P_{pos}[j]$)
41:         end for
42:         $P_{vel}[i] \leftarrow (force + P_{vel}[i])$
43:     end for
44: end for
45:
46: // Integrate motion
47: for each core $i < N_P$ do in parallel
48:     $P_{vel}[i] \leftarrow damping * (P_{vel}[i] + (g * \Delta t))$;
49:     $P_{vel}[i] \leftarrow$ COLLIDEWITHBOUNDINGBOX($P_{pos}[i]$)
50:     $P_{pos}[i] \leftarrow P_{pos}[i] + (P_{vel}[i] * \Delta t)$
51:     if $P_{pos}[i].y < b.min.y$ then
52:         $P_{pos}[i].y \leftarrow b.max.y$
53:         if $Pcol[i] \neq (0,0,0)$ then
54:             $G_{IG}$[GETCELLHASH($Pcol[i]$)]+=1
55:             $Pcol[i] \leftarrow (0,0,0)$
56:         end if
57:     end if
58: end for
```

$Cell$ $Hash$	$P_{index}Start$ $(CellStart)$	$P_{index}End$ $(CellStop)$
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	$UINT_{max}$	*

$N_{G_{SHT}}$

Fig. 6. The cell lookup table: for each cell's hash value, $CellStart$ and $CellStop$ store the beginning and end indices of contained particles in P_{index}. $UINT_{max}$ in $CellStart$ denotes empty cells, while * in $CellStop$ denotes an undefined value.

taining $P_{pos}[threadId]$ 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 $CellStart$ and $CellStop$ 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). 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, $CellStart$ and $CellStop$ are used to quickly access the indices of contained particles. When collisions occur, $P_{vel}[threadId]$ is updated using forces computed by the discrete elements method and $P_{pos}[threadId]$ is copied to $P_{col}[threadId]$ (line 35), allowing this particle's last collision to be retrieved when 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 timestep 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, G_{IG}'s cells are sorted in order of decreasing information gain values, and their respective positions in $\mathbb{R}^3$ are copied into the CPUs memory space. After close waypoints are merged, they are passed to the path planner, which will try to steer the UAV as close to the given goal configurations as possible.

V. Optimization

For optimization, a locality preserving hashing function is used to assign hash values to G_{SHT}'s cells. Prior to collision detection, particle and collider positions are sorted according to the hash values of the cells containing them. This is not only a prerequisite for populating $CellStart$ and $CellStop$, it also 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 both particles and colliders also best leverages the L2 caches for global memory access that emerged with CUDA compute capability 2.0, as particle and collider positions will already be cached when neighboring threads need to fetch their positions in order to execute collision tests against them.

To reduce the amount of memory and the number of collision-test required, point cloud C is downsampled from the original, dense point cloud by removing points that have neighbors closer than a distance d. In theory, as long as $d < 2r$, particles $p \in P$ with a radius r will be unable to fall through gaps that appeared in C in the process of sparsing. In practice, as the simulation is performed using discrete timesteps, d should be chosen with a safety margin to avoid fast-falling spheres to pass through C due to coarse timesteps.

To render particle and collider positions efficiently, they are stored in OpenGL vertex buffer objects. During simulation, they are mapped into CUDA address space, avoiding copies between interleaving CUDA and OpenGL stages.

VI. Results

The implementation of [10] 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 ocurred 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 fig. 8, testing a point cloud with 10k points for watertightness using 1k to 64k particles 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 pointcloud visualization using OpenGL core profile are very suitable for processing on Single Instruction Multiple Data (SIMD) architectures.

Because the proposed algorithm causes the information gain in G_{IG}'s cells to steadily increase during simulation, termination criteria are non-apparent: a tradeoff 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 7 presents the *normalized* information gain values of the point cloud depicted in fig. 2(f) 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 $64^3 m$, 16k particles ($p \in P$) with a radius of $0.25m$ and 64k colliders ($c \in C$) allows generation of multiple NBVs in less than $3s$ using a NVIDIA GTX 670 graphics card.

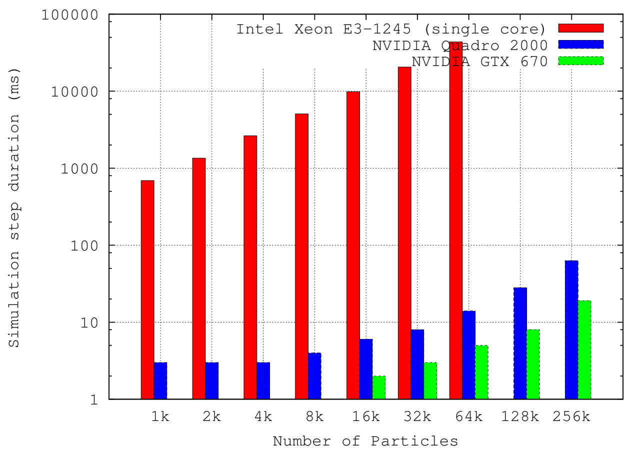

Fig. 8. This graph shows the maximum time required for a single timestep of the simulation containing 16k colliders against the number of particles. Visualization was disabled in both tests. Note the logarithmic scale of the ordinate axis. The runtime of the proof-of-concept implementation for the CPU is up to three orders of magnitude longer than those of the GPU-based implementations.

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 I. 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.

VII. Outlook

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

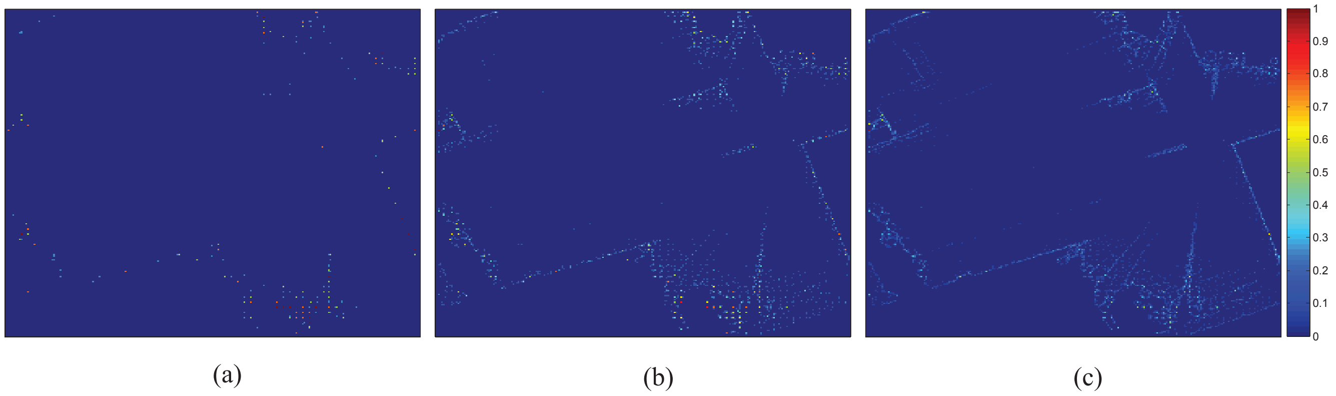

Fig. 7. Grid of information gain with $256 * 32 * 256$ cells, projected as in fig. 2(f). Maximum information gain is shown normalized using a heat map visualization a) after 400, b) after 700 and c) after 1000 simulation steps.

TABLE I
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
$CellStart_P$	uint	4 bytes	$N_{G_{SHT}}$
$CellStop_P$	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
$CellStart_C$	uint	4 bytes	$N_{G_{SHT}}$
$CellStop_C$	uint	4 bytes	$N_{G_{SHT}}$
G_{IG}	char	1 byte	$N_{G_{IG}}$

collider positions will be researched, as it allows doubling both capacity and 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 $c \in C$ 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.

To 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 (shown in fig. 4). Tests are currently being done with Hilbert- and Z-Order curves (Morton code).

We have successfully developed a planar surface based outdoor mapping system in our previous work [11], 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

[1] C. Connolly, "The determination of next best views," in *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, vol. 2, pp. 432–435, 1985.

[2] J. O'Rourke, *Art Gallery Theorems and Algorithms*. New York, NY: Oxford University Press, 1987.

[3] W. R. Scott, G. Roth, and J.-F. Rivest, "View planning for automated three-dimensional object reconstruction and inspection," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 64–96, 2003.

[4] C. Potthast and G. S. Sukhatme, "A probabilistic Framework for Next Best View Estimation in a Cluttered Environment." Sept. 2011.

[5] M. Bryson and S. Sukkarieh, "Active airborne localisation and exploration in unknown environments using inertial SLAM," in *Aerospace Conference, 2006 IEEE*, p. 13 pp., 2006.

[6] P. Blaer and P. K. Allen, "Data acquisition and view planning for 3-D modeling tasks," in *IROS*, (Marina, San Diego, California, USA), pp. 417–422, IEEE, Oct. 2007.

[7] B. Yamauchi, "Frontier-Based Exploration Using Multiple Robots.," in *Agents*, pp. 47–53, 1998.

[8] R. Shade and P. Newman, "Choosing where to go: Complete 3D exploration with stereo.," in *ICRA*, (Shanghai, China), pp. 2806–2811, IEEE, May 2011.

[9] A. Mobarhani, S. Nazari, A. H. Tamjidi, and H. Taghirad, "Histogram based frontier exploration," in *IROS*, (San Francisco, CA, USA), pp. 1128–1133, IEEE, Sept. 2011.

[10] B. Adler, J. Xiao, and J. Zhang, "Towards Autonomous Airborne Mapping of Urban Environments," in *Multisensor Fusion and Information Integration (MFI), 2012 IEEE International Conference on*, Sept. 2012.

[11] J. Xiao, B. Adler, H. Zhang, and J. Zhang, "Planar segments based 3d point cloud registration in outdoor environments," *Journal of Field Robotics (accepted on March 7th, 2013)*, 2013. http://tams.informatik.uni-hamburg.de/people/xiao/publications/Junhao_JFR2013.pdf.