ISPRS Journal of Photogrammetry and Remote Sensing xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

ISPRS Journal of Photogrammetry and Remote Sensing

journal homepage: www.elsevier.com/locate/isprsjprs

Removing non-static objects from 3D laser scan data

Johannes Schauer*, Andreas Nüchter

Informatics VII: Robotics and Telematics, Julius-Maximilians-University Würzburg, Am Hubland, Würzburg 97074, Germany

ARTICLE INFO

Keywords:
3D point clouds
Change detection
Moving object detection
Voxel traversal
Post-processing
Sphere quadtree
Mobile mapping
Terrestrial scans

ABSTRACT

For the purpose of visualization and further post-processing of 3D point cloud data, it is often desirable to remove moving objects from a given data set. Common examples for these moving objects are pedestrians, bicycles and motor vehicles in outdoor scans or manufactured goods and employees in indoor scans of factories. We present a new change detection method which is able to partition the points of multiple registered 3D scans into two sets: points belonging to stationary (static) objects and points belonging to moving (dynamic) objects. Our approach does not require any object detection or tracking the movement of objects over time. Instead, we traverse a voxel grid to find differences in volumetric occupancy for "explicit" change detection. Our main contribution is the introduction of the concept of "point shadows" and how to efficiently compute them. Without them, using voxel grids for explicit change detection is known to suffer from a high number of false positives when applied to terrestrial scan data. Our solution achieves similar quantitative results in terms of F_1-score as competing methods while at the same time being faster.

1. Introduction

When 3D laser scanners are used to create digital maps and models, it is hard to imagine scenarios where non-static or moving objects are supposed to be part of the final point cloud. Examples for point cloud data that is supposed to be free of moving objects are:

- an indoor office for intrusion detection or workspace planning,
- a factory or industrial sites for industry 4.0 applications,
- a mining site to monitor progress and watch for hazards,
- an urban environment for city planning and documentation purposes,
- a historical site for archaeology and digital preservation purposes,
- and environments for gaming and virtual reality applications.

In all these examples, it is undesirable to have moving objects be part of the final point cloud. The easiest approach to achieve a point cloud free of moving clutter is to scan an environment that is completely static. Unfortunately, in realistically-scaled real world scenarios this is hard or even impossible to achieve. Factories and mining sites would have to suspend work for the duration of the scan, thereby causing production losses and making it infeasible to carry out scans regularly. Closing off large sections of an urban environment and freeing it of pedestrians, moving and parked cars and bicycles comes with great bureaucratic challenges and heavily inconveniences the local residents.

One way to solve this dilemma is to take multiple scans from the exact same location and then only keep those points in volumes found to be occupied by most scans. But this solution comes with several disadvantages. Not only does this method take considerably more time than just taking a single scan, it is also unclear how many scans one has to take or how to find a good heuristic to select the right threshold that classifies a volume as static. If the threshold is too high, then static points only seen a few times will not be recorded. The lower the threshold the more dynamic points will wrongly be classified as static. The method we propose solves all of these issues. We successfully applied our method to various point clouds from our scan repository.[1] These scans were not recorded with our algorithm in mind, proving that our method will probably apply to many existing regular terrestrial scan dataset.

1.1. Our approach

The input to our algorithm is registered 3D range data, typically acquired by a 3D laser range finder from multiple vantage points. While we only test our approach with LIDAR scans, it is in principle also compatible with scans obtained from RADAR or RGB-D systems or point clouds from stereo vision. Any input which allows associating every

* Corresponding author.
E-mail addresses: johannes.schauer@uni-wuerzburg.de (J. Schauer), andreas@nuechti.de (A. Nüchter).
URL: http://www.nuechti.de (A. Nüchter).

[1] http://kos.informatik.uni-osnabrueck.de/3Dscans/.

https://doi.org/10.1016/j.isprsjprs.2018.05.019
Received 3 October 2017; Received in revised form 24 March 2018; Accepted 29 May 2018

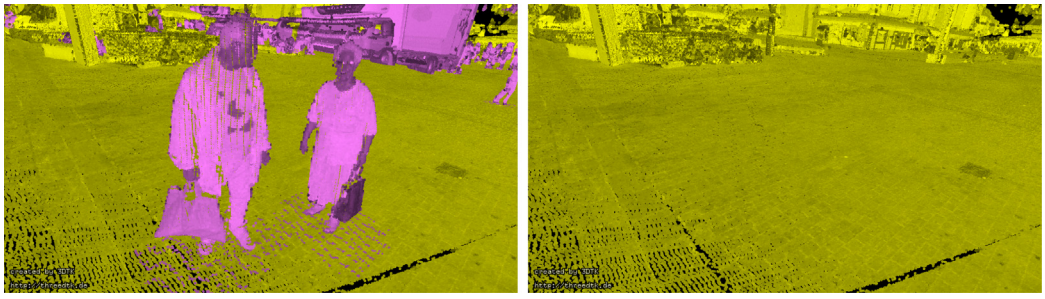


Fig. 1. After identifying non-static points (in magenta on the left) they are removed without artifacts (right). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

measured point with the line of sight from which it was measured is theoretically suitable for our method. In terms of terrestrial laser scan data, a suitable format are multiple point clouds, each in the scanner's own local coordinate system together with registered 6DOF positions of the laser scanner for each point cloud. It would make the data unsuitable for our approach if all scans were merged into a single point cloud and transformed into a global coordinate system, thus loosing the association between measured points and the vantage points from which they were each measured.

Retaining that information is imperative to our approach because we identify dynamic points by traversing the lines of sight under which each point in the dataset was measured through a voxel occupancy grid. Essentially: all points in voxels that intersect with a line of sight are then classified as dynamic because if they were static, points behind the voxel shouldn't have been visible. This implies, that our approach is only able to detect change in volumes where two or more scans overlap and suppresses apparent changes created by occlusion. This makes our method an "explicit" change detection algorithm.

Our algorithm makes very few requirements on the underlying geometry of the scanned data, vantage points and the temporal separation between individual scans. The vantage points together with the geometry of the scene must be chosen such that the volumes of interest are not occluded from the sensor. Instead, the volumes that one wants to remove moving objects from must have been observed at least by two different scans. Furthermore, the temporal difference between these two scans must be large enough such that any object that one considers "dynamic" in the observed volume was moved to a different location. But if a given voxel volume was observed more than twice, then it is sufficient that the voxel was seen as "free" by only a single scan.

Our method performs best in environments with clear surface normals but in their absence, false positives are easily removed by a fast clustering algorithm. To avoid artifacts due to the voxel discretization we also show an algorithm that reliably removes them without reducing the quality of the remaining point cloud. An example of the output of our algorithm is shown in Fig. 1 where pedestrians in the foreground and cars in the background are classified as non-static and subsequently removed.

1.2. Contribution

Our main contributions are:

- an algorithm that is able to identify and remove dynamic points in 3D point clouds
- an improved and extended version of the voxel traversel algorithm by Amanatides et al. (1987)
- an algorithm to efficiently compute point shadows
- an approach that doesn't classify whole voxels as dynamic but only subsets of points in a voxel, achieving sub-voxel accuracy

We publish the source code of our approach as part of *3DTK – The 3D Toolkit*.[2] Except for the Wolfsburg dataset, all datasets we present in this paper are publicly available as well.[3] Furthermore, we provide the shell scripts that allow to precisely reproduce the F1-scores displayed in the results section.[4]

1.3. Organization of this paper

This paper is organized as follows. In the next section we discuss other work related to the topics covered in this paper. Section 3 gives an overview of our approach. The following five sections then detail our method. Section 4 describes our improvements to the voxel traversal algorithm by Amanatides and Woo. In Section 5 we extend our method that was previously limited to scan slices (Schauer and Nüchter, 2017) to the more general setup of terrestial scan data by introducing the concept of "point shadows". Computing the latter requires frequent lookup of angular neighbors for which we use a sphere quad tree as described in Section 6. To remove small instances of voxels wrongly classified as dynamic we employ a clustering algorithm which we detail in Section 7. Section 8 then describes a way to also remove false negatives introduced due to the voxel occupancy grid. Finally, we show qualitative and quantitative results in Section 9, show performance graphs and compare our method to a competing solution. Section 10 handles the limitations of our method and in Section 11 we describe the direction of future research in this area before we draw conclusions in Section 12.

2. Related work

Our solution falls into the realm of change detection (Qin et al., 2016) but only few publications deal with classifying points as either dynamic or static. Even fewer approaches compute the free volume between a measured point and the sensor itself. Most solutions for change detection compare incoming geometries or point clouds in a way that results in "change" merely due to occlusion or incomplete sensor coverage. One example for such an approach is the method by Vieira et al. which uses spatial density patterns Vieira et al. (2014). Or the solution shown in Liu et al. (2016) which just computes the difference in voxel occupation between two input scans. But for our purpose of "cleaning" scans, it is undesirable to remove these parts from the dataset. Doing so would mean to remove potentially useful data from the input. Instead, we designed our algorithm to be conservative. It only removes volumes which it is able to confidently determine to be dynamic. Volumes which it cannot make a decision upon, for example because they were only measured by a single scan, are left untouched. Meeting this requirement is only possible by computing unoccupied

[2] http://threedtk.de.
[3] http://kos.informatik.uni-osnabrueck.de/3Dscans/.
[4] https://robotik.informatik.uni-wuerzburg.de/telematics/download/isprs2018/.

volumes and detecting change explicitly. The changes we are interested in can only be detected if a given point falls into the volume that another measurement observed as free.

The work most similar to ours is the seminal work by Underwood et al. (2013). It is able to detect changes between two scans by ray tracing points in a spherical coordinate system. But since their algorithm is limited to comparing no more than two scans at a time it is not directly applicable to our use case without either additional heuristics or quadratic runtime with respect to the number of scans. Given N input scans and without additional processing to find scan pairs with a "meaningful" overlap in their observed volume, the only way to find all changed points is to compare all possible pairs of scans. With N scans this results in a worst case scenario of $\frac{N(N-1)}{2}$ comparisons and thus quadratic runtime. Our approach is of linear complexity relative to the input size because all comparisons are made against a global occupancy grid and not directly against point data from other scans. The authors publicly provide their code and their datasets which we thus use to benchmark our own method against theirs.

Similar to the approach by Underwood et al., the solution by Gálai and Benedek (2017) finds changes by comparing range images. One range image is obtained directly from a live laser scanner while the other is the projection of a known-static point cloud of the environment into the current position of the laser scanner. Differences in the range image data is then classified as change and projected back into the 3D space.

Another approach close to ours is the method by Xiao et al. (2013, 2015). Similar to our method and the method by Underwood et al. their algorithm also considers the volume by laser rays and fuses multiple rays into a larger volume using the Dempster-Shafer theory for intra-data evidence fusion and inter-data consistency assessment. Similar to our method, they rely on surface normals but unlike ours, the method detects changes at the point-level without voxelization.

Asvadi et al. (2016a,b)also use a voxel data structure to partition the input point cloud into static and dynamic points but instead of recording free voxels, they count how often a voxel is occupied. Due to varying occlusion they have to make a number of assumptions about their environment and employ several heuristics that are not necessary with our algorithm. Furthermore, their approach requires a ground surface estimation – in contrast to our approach which does not require any such planar features to be present.

The creators of OctoMap (Hornung et al., 2013) also use the same algorithm as we do by Amanatides et al. (1987) to cast rays. But instead of voxels they use an oct-tree data structure to find free volumes. They also employ a similar approach to avoid marking volumes as free in situations where rays meet a surface at a very shallow angle by grouping multiple scan slices together. We improve on their work by generalizing their approach for scan slices to terrestrial scans. The OctoMap approach is also used by other implementations like the one seen in Ruixu Liu (2017).

Besides voxels and oct-trees other data structures to store occupation information in are elevation maps (Herbert et al., 1989; Pfaff et al., 2007), multi-level surface maps (Triebel et al., 2006), and Gaussian Mixture Model (Andreasson et al., 2007; Núñez et al., 2010; Drews-Jr et al., 2013).

Existing methods that require computation of free volume for robotic path planning are known to use a 3D Bresenham ray casting kernel (Hermann et al., 2014) carrying out the ray casting in many parallel threads on the GPU. GPU-based ray casting techniques were first shown by Roettger et al. (2003) and Kruger and Westermann (2003) and are today often implemented using OpenGL and CUDA (Weinlich et al., 2008).

3. General design

Our initial approaches were inspired by how humans tend to distinguish between static and dynamic objects: If an object is seen as immobile for long enough, then we will classify it as static. While this approach would probably work well for a scanner with a static position relative to the environment that we consider static, it seems to be an unfeasible approach in the mobile mapping scenario. Due to the scanner moving and the resulting variable occlusion of objects, it is hard to calculate how long an object *should* be visible and not vanish because of an occlusion. Furthermore, without prior knowledge about whether the occluding object is actually static a chicken-and-egg problem is created. We need to know about which objects are static before they are considered for occlusion testing. But to reliably test for occlusion we need to know which objects are static.

Thus, instead of counting how long or how often an object is seen as static, our algorithm does the opposite and instead tests whether any seen object was at any point in time *see-through*. Since we want to avoid any higher-level processing like object recognition, our "objects" here are the voxels of a regular voxel grid. This spacial approximation of the sensor data has the advantage, that it is computationally easy to enumerate all voxels along a ray with the scanner at its origin. The ray is forming a line of sight. To generate the regular voxel grid from a set of input points, we define:

Definition 1. The voxel address of a given 3D point is a three-tuple computed from the Cartesian coordinate of the point, each divided by the voxel size and rounded to the next smallest integer.

Thus, our voxel grid forms a cubic lattice with each point in $\mathbb{R}^3$ belonging to exactly one grid cell. Since the Cartesian coordinates may be negative, negative voxel addresses exist as well. The grid implicitly forms an occupancy map where voxels with one or more points in them are occupied and those with zero points in them are unoccupied.

To determine the set of see-through voxels, we model the laser beam as a ray and enumerate all voxels intersecting with that ray. Voxels that are see-through and contain points must be dynamic. An additional advantage of this approach is, that it will automatically *not* remove points that were only measured very seldom or even only once. No heuristic about object movement is required.

Fig. 2 displays the general idea of our algorithm in a two-dimensional scenario. The circular object in areas C1 and C2 is dynamic and only seen by the first scan (Fig. 2a). Since the second scan (Fig. 2b) measures the red points in A2 and B2 with a line of sight that crosses area C1, the three points that were measured in C1 in the first scan are dynamic.

Fig. 2 also shows how the algorithm does not remove points from areas that were only visible in a single scan. For example the green points in areas A3 and A4 are only seen from the scanner position in Fig. 2a. Still, they are not removed because these areas are never marked as see-through by other scans (for example the second scan in Fig. 2b). The same holds for the red points in area C2. They are only seen by the second scan in Fig. 2b because the circular moving object in C1 and C2 occludes the points during the first scan in Fig. 2a. Still, the points remain classified as static because their containing areas are never marked as see-through.

Our algorithm goes through the following stages:

1. Loading point cloud data from input files in scanner-local coordinate system together with the registered 6DOF scanner position
2. Creation of voxel occupancy grid. Each voxel stores the set of scan indices that have a point in that voxel
3. Computation of maximum traversal distances through the voxel grid for each point by using "point shadows"
4. Traversing lines of sight through the voxel grid for each scanner location to each measured point by that scan, identifying see-through voxels
5. Clustering for false positive noise removal
6. Removal of false negatives through our approach to sub-voxel accuracy

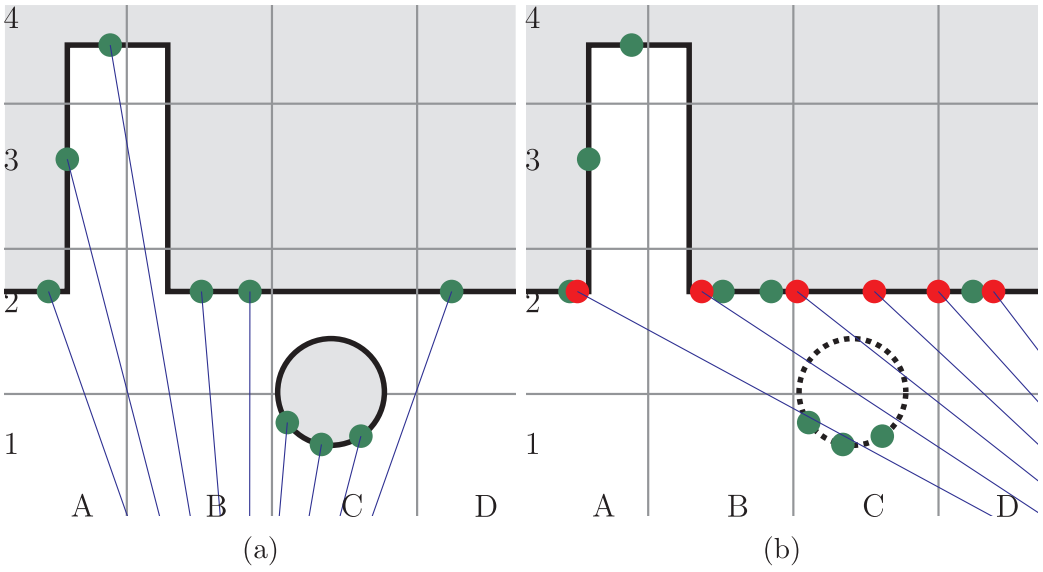


Fig. 2. The algorithm applied to a 2D scenario. The gray raster marks the 2D voxel (pixel) boundaries. Blue lines mark the scanner lines of sight. Dark lines are object boundaries. Gray areas mark solid space while white areas mark free space. **Left:** The scene as scanned from a center position (ray origin not part of the figure). The scanner measures the green points. **Right:** The scene as scanned from a position to the right. The circular object in areas C1 and C2 moved away and its former position is marked with dotted lines. The scanner measures the red points. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

7. Writing out results

The main component of our method is a global occupancy grid which we store as a voxel data structure. Each voxel holds a set of scan identifiers. A scan identifier is added to a given voxel if any point of that scan falls into the voxel. Thus, precise point coordinates are not stored in the grid. Instead, the data structure represents the union of all voxels that the input scans measured points in. For example in Fig. 2 on the right, voxel B2 stores the information that it contains points from the green as well as from the red scan but neither their number nor the coordinates of these points is stored in the voxel grid. Thus, the global occupancy grid typically requires several orders of magnitude less memory than the sum of the input data.

By traversing the occupancy grid from each sensor origin to the coordinates of each measured point, we find voxels that intersect with the line of sight of the sensor but contain a non-empty set of scan identifiers. These voxels are then classified as “see-through” or “dynamic”. At the end of the algorithm, this information serves as a binary classifier determining whether a given input point should be removed or not. A point is removed if it falls into a voxel that was marked as “see-through”.

The reason why we store a set of scan identifiers in each voxel instead of just storing a binary occupied/unoccupied property is to be able to abort traversal early and avoid self-intersections. The point measured in voxel A4 in Fig. 2 on the left has a line of sight intersecting with at least three voxels that must not be marked as free: A3, B3, and B2. To avoid wrongly marking these voxels as free, traversal is aborted once a voxel is encountered containing the same scan identifier as the scan the current target point belongs to. This means that the ray toward voxel A4 aborts before marking voxel B2 as free. Another application for storing sets of scan identifiers in each voxel is our solution to achieve sub-voxel accuracy as explained in Section 8.

4. Fast voxel traversal

To enumerate all see-through voxels from the laser origin until the measured point, we used an approach based on the algorithm proposed by Amanatides et al. (1987). We improve the algorithm by making it adhere to a stricter definition of what it means for a ray to intersect with a voxel, by eliminating accumulation of floating point errors and by adding support for rays starting exactly at a voxel boundary. None of the existing open-source implementations (Octomap (Hornung et al., 2013), MRPT (Blanco-Claraco, 2014), PCL (Rusu and Cousins, 2011), yt (Turk et al., 2011)) supports any of these properties and thus we detail of our approach here.

Even though our additions to Amanatides and Woo’s original algorithm increase the number of possible instructions per loop cycle we were unable to measure a difference in runtime of the algorithms on an Intel Core i5 platform. We assume that this is because the bottleneck of the algorithm is the required non-local memory access and not the raw instructions per traversed voxel.

4.1. Approach by Amanatides and Woo

The original approach by Amanatides and Woo for fast voxel traversal is also often called “3D Bresenham algorithm” because it is very similar in nature. The core of the algorithm in two dimensions is displayed in Algorithm 1 will traverse a ray $\vec{u} + t\vec{v}$ for $t \geqslant 0$. Extending it into three dimensions just adds an additional set of `Z` variables and finds the minimum of all three `tMax` values in each loop iteration.

Algorithm 1. Fast Voxel Traversal Algorithm by Amanatides and Woo

```
1: while true do
2:     if tMaxX < tMaxY then
3:         tMaxX ← tMaxX + tDeltaX
4:         X = X + stepX
5:     else
6:         tMaxY ← tMaxY + tDeltaY
7:         Y = Y + stepY
8:     NEXTVOXEL X, Y
```

The variables used in the algorithm are initialized as follows:

- X, Y are the starting voxel coordinates, i.e. the voxel in which the ray origin $\vec{u}$ is found
- $stepX$, $stepY$ are set to 1 or -1 depending on whether X and Y are incremented or decremented, respectively, when traversing the grid. This is the sign of the x and y components of $\vec{v}$.
- $tMaxX$, $tMaxY$ are set to the value of t in which the ray crosses the first voxel boundary in x and y direction, respectively.
- $tDeltaX$, $tDeltaY$ store how far along the ray one must move in units of t to traverse exactly one voxel in x and y direction, respectively.

Neither the original publication nor any other publication since then explains in more detail how these variables are set up exactly. We found though, that many of the current limitations of existing

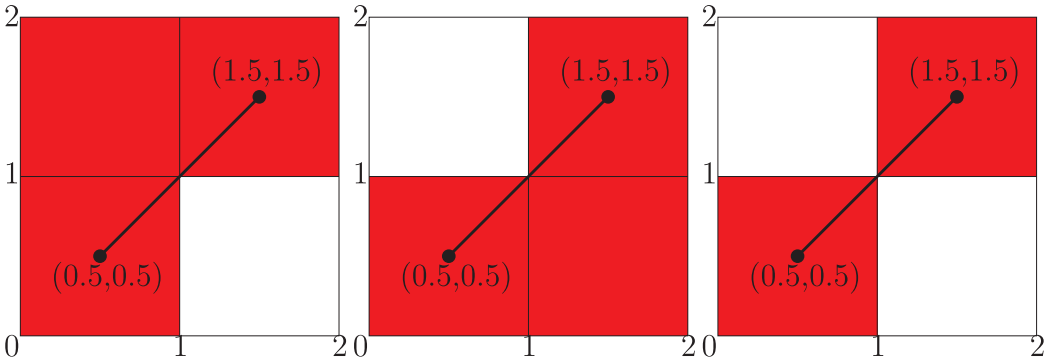


Fig. 3. Two-dimensional example visualizing the problem of existing implementations of the voxel traversal algorithm by Amanatides and Woo with a voxel size of 1. Traversed voxels are marked in red. A line segment from (0.5, 0.5) to (1.5, 1.5) includes the coordinate (1, 1) which belongs to voxel (1, 1) and neither voxel (0, 1) nor voxel (1,0) should be included in the result. **Left:** If the traversal algorithm checks the vertical dimension first, then voxel (0, 1) is added to the result. **Center:** If the traversal algorithm checks the horizontal dimension first, then voxel (1,0) is added to the result. **Right:** Our algorithm only traverses voxel (0,0) and voxel (1, 1). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

implementations come from unexpected corner-cases in how these variables are set up. Thus, in this section we also describe how to initialize the variables in detail.

4.2. Definition of line-voxel intersection

The original algorithm is ambiguous in situation where the ray intersects with the voxel edges or corners. Existing implementations all handle this situation in different ways, so to eliminate ambiguity and to be able to verify the correctness of our approach we define what it means for a voxel to intersect with a line.

Definition 2. A line intersects with a given voxel if and only if any point on the line falls into the given voxel according to Definition 1.

By only being able to step into one voxel grid dimension at a time, existing implementations fail Definition 2 in cases where the traversed ray enters or exits a voxel through its corners or edges. Two-dimensional examples of these situations are shown in Figs. 3–5. The original algorithm by Amanatides and Woo forces the implementation to arbitrarily pick a dimension to step into first but no matter which dimension is picked, the result will contain wrong voxels and miss others that should be included.

4.3. Avoiding accumulation of floating point errors

The original algorithm by Amanatides and Woo increments the `tMax` variables by the corresponding `tDelta` value in each loop iteration. Since both variables are floating point values, this accumulates an error over time, especially once `tMax` becomes several orders of magnitude larger than `tDelta`. Additionally, when the ray is nearly parallel to the coordinate axis, small floating point errors will lead to

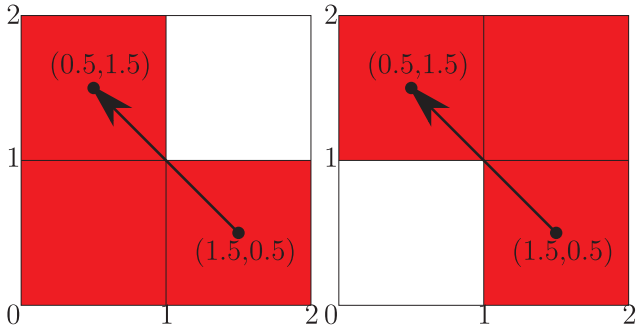


Fig. 4. A line is traversed from (1.5, 0.5) to (0.5, 1.5). The arrow indicates the direction. **Left:** If the traversal algorithm checks the horizontal dimension first, then voxel (0, 0) is added to the result even though point (1, 1) on the line belongs to voxel (1, 1). **Right:** Our algorithm correctly identifies voxel (1, 1) as part of the solution.

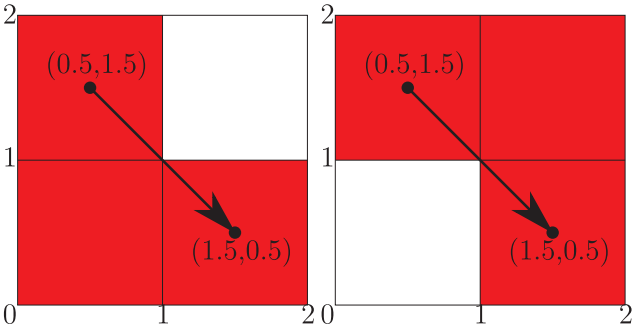


Fig. 5. A line is traversed from (0.5, 1.5) to (1.5, 0.5). The arrow indicates the direction. **Left:** If the traversal algorithm checks the vertical dimension first, then voxel (0, 0) is added to the result even though point (1, 1) on the line belongs to voxel (1, 1). **Right:** Our algorithm correctly identifies voxel (1, 1) as part of the solution.

incorrectly traversed voxels when the ray is about to cross the voxel boundary along the dimension the ray is nearly parallel to. To avoid these effects, we introduce new counter variables for each dimension. These variables are of integer type and count how much the loop has so far stepped into each direction. This allows to compute the new `tMax` values in each iteration by adding floating point values of similar magnitude which reduces errors. Additionally, the integer counter variables allow a more precise way to abort the algorithm by not having to rely on the potentially faulty `tMax` floating point values. In our adapted solution, the `tMax` values are only used to decide in which direction(s) to step next.

4.4. Rays starting exactly at a voxel boundary

The setup phase of the algorithm by Amanatides and Woo is of particular importance but neither completely explained in the original paper nor in the papers citing it. An important corner case which is not handled by the existing implementations is the case when a ray starts exactly on a voxel boundary and then continues into negative direction. To illustrate the problem, we use a one-dimensional example with a "voxel"-size of 1. Suppose the ray starts at coordinate 1 and goes into negative direction. The starting voxel is voxel 1. In this situation, we compute `tMax` as the value of `t` needed to go from 1 to 0, thus `tMax` equals `tDelta`. In the loop, we step one voxel into negative direction and increment `tMax` by `tDelta`. As a result we are now in voxel 0. But that conflicts with the current value of `tMax` which is now twice the value of `tDelta` and thus indicates that we already stepped *two* voxels instead the single step that was just carried out. It is also wrong to reset the value of `tMax` to zero before starting the loop because in the more-dimensional case that means that the first step in the loop is not made in the direction of the voxel sharing the voxel boundary the starting point

lies on. The correct solution is to add an additional step after the initial setup but before the loop starts. In this step one additional voxel into negative direction has to be added. Since existing implementations are not taking care of this special case, they will skip one voxel at the beginning and thus compute a result that will always be off-by-one.

4.5. Implementation

Since the required additions to the original algorithm by Amanatides and Woo are complex, we present in this subsection the complete pseudo code of the fixed voxel traversal algorithm. We split the function WALKVOXELS into two parts. Algorithm 2 shows the setup phase while Algorithm 3 the loop walking through the voxel grid. The algorithm makes use of three additional functions. VOXELOFPOINT returns the voxel coordinate of a given cartesian coordinate according to Definition 1. Care has to be taken in a C/C++ implementation because simple integer division will always round toward zero. A function showing a possible implementation in C/C++ is shown in Listing 1 in the appendix.

The VISITOR callback handles the current voxel, for example by adding it to a list of traversed voxels. But the behaviour of this function is up to the requirements of the user of WALKVOXELS. Finally the function MIN returns the smallest value of the arguments it is given. Many variables represent 3-tuples where elements are accessed using the `[]` operator with a zero-based index.

Algorithm 2. Extended voxel traversal algorithm (part 1)

1: **function** WALKVOXELS(*startpos*, *endpos*, *voxelsize*)
2: $startvoxel \leftarrow$ VOXELOFPOINT(*startpos*, *voxelsize*)
3: VISITOR(*startvoxel*)
4: $endvoxel \leftarrow$ VOXELOFPOINT(*endpos*, *voxelsize*)
5: **if** $startvoxel = endvoxel$ **then**
6: **return**
7: $direction \leftarrow endpos - startpos$
8: **if** $direction = (0, 0, 0)$ **then**
9: **return**
10: $curvoxel \leftarrow startvoxel$
11: **for** $i \leftarrow 0, 2$ **then**
12: **if** $direction[i] = 0$ **then**
13: $tMax[i] \leftarrow \infty$
14: $maxMult[i] \leftarrow \infty$
15: **else**
16: **if** $direction[i] > 0$ **then**
17: $step[i] \leftarrow 1$
18: **else**
19: $step[i] \leftarrow -1$
20: $tDelta[i] \leftarrow \frac{step[i] \cdot voxelsize}{direction[i]}$
21: $tMax[i] \leftarrow tDelta[i]\left(1 - \frac{step[i] \cdot startpos[i]}{voxelsize} \bmod 1\right)$
22: $maxMult[i] \leftarrow step[i] \cdot (endvoxel[i] - startvoxel[i])$
23: **if** $step[i] = -1 \vee tMax[i] = tDelta[i] \vee startvoxel[i] \neq endvoxel[i]$ **then**
24: $curvoxel[i] \leftarrow curvoxel[i] - 1$
25: $maxMult[i] \leftarrow maxMult[i] - 1$
26: **if** $curvoxel \neq startvoxel$ **then**
27: VISITOR(*curvoxel*)
28: $startvoxel \leftarrow curvoxel$
29: **if** $curvoxel = endvoxel$ **then**
30: **return**

Algorithm 2 mostly implements the standard setup from the algorithm by Amanatides and Woo. Again, if implementing the algorithm in C/C++, care has to be taken to carry out the modulo operation correctly (line 21), as the native '%' operator only operates on integers and the `fmod` function from `math.h` only computes the remainder of two floating point numbers despite its name. An example implementation of floating point modulo operation in C/C++ is given in Listing 2 in the appendix.

The additions start in line 22 which sets up the 3-tuple *maxMult* containing the voxel difference between the start and end voxel. Line 23 then checks whether the special condition explained in Section 4.4 is met: if a step has to be done into negative direction and the ray starts at the voxel boundary along that dimension, then that step is already carried out before the main loop starts. This results in a *second* call to VISITOR if necessary in line 27.

Algorithm 3. Extended voxel traversal algorithm (part 2)

31: $mult \leftarrow (0, 0, 0)$
32: $tMaxStart \leftarrow tMax$
33: **loop**
34: $stepped \leftarrow (false, false, false)$
35: $minVal \leftarrow$ MIN($tMax$)
36: **for** $i \leftarrow 0, 2$ **do**
37: **if** $minVal = tMax[i]$ **then**
38: $mult[i] \leftarrow mult[i] + 1$
39: $curvoxel[i] \leftarrow startvoxel[i] + mult[i] \cdot step[i]$
40: $tMax[i] \leftarrow tMaxStart[i] + mult[i] \cdot tDelta[i]$
41: $stepped[i] \leftarrow true$
42: **if** $((stepped[0] \vee stepped[1]) \wedge (stepped[0] \vee stepped[2]) \wedge (stepped[1] \vee stepped[2])) \vee (step[0] = 1 \wedge step[1] = 1 \wedge step[2] = 1) \vee (step[0] = -1 \wedge step[1] = -1 \wedge step[2] = -1)$ **then**
43: $addvoxel \leftarrow curvoxel$
44: **for** $i \leftarrow 0, 2$ **do**
45: **if** $stepped[i]$ **then**
46: **if** $step[i] < 0$ **then**
47: **if** $mult[i] > maxMult[i] + 1$ **then**
48: **return**
49: $addvoxel[i] \leftarrow addvovel[i] + 1$
50: **else if** $mult[i] > maxMult[i]$ **then**
51: **return**
52: **if** $addvoxel \neq curvoxel$ **then**
53: VISITOR(*addvoxel*)
54: **for** $i \leftarrow 0, 2$ **do**
55: **if** $stepped[i] \vee mult[i] > maxMult[i]$ **then**
56: **return**
57: VISITOR(*curvoxel*)

Algorithm 3 executes the voxel traversal. The difference to the original algorithm is twofold. Firstly, as explained in Section 4.3, our version increments *tMax* not by directly adding *tDelta* but by adding a multiple of it to the initial *tMax* value (line 40). This makes it necessary that the additional counter *mult* is kept updated for each dimension (line 38). As a side-effect the algorithm also uses the value of *mult* to decide when the target voxel is reached (lines 47, 50, and 55). Secondly, our version is able to step into more than one direction at once. This is achieved by stepping into every direction that shares the minimum *tMax* value *minVal*. As explained in Section 4.2, additional checks need to be carried out if a step was done into more than one direction at the same iteration step to also consider potentially "graced" voxels. These checks are done in lines 42 to 53 and lead to the VISITOR callback being executed one additional time within a given loop iteration if necessary.

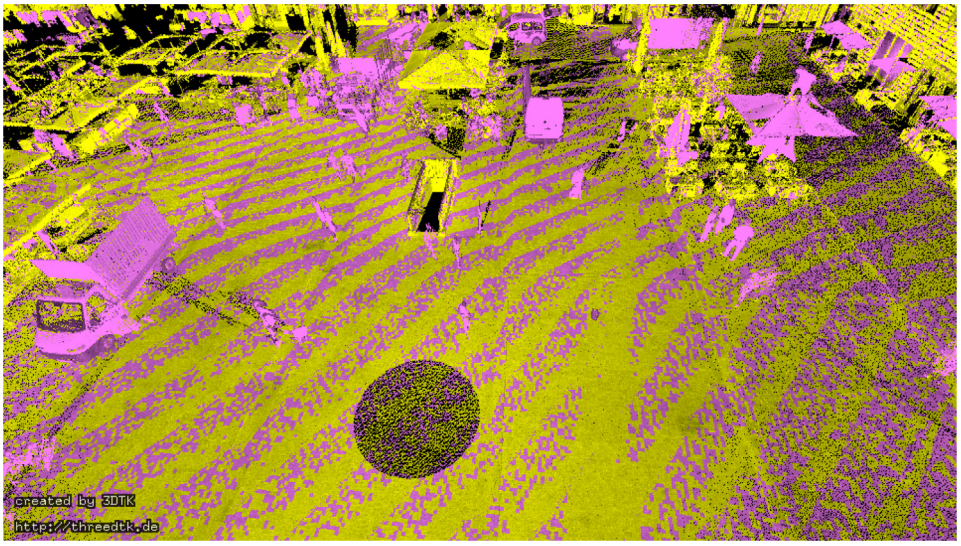


Fig. 6. Result of naive approach to detecting dynamic points by directly using the method from mobile mapping. Dynamic points are marked in magenta. There are several "stripes" of false positives on the ground.

5. Panorama scans from terrestrial mapping

Attempting to apply the same technique from mobile mapping to terrestrial scans will result in unexpected false-positives as they are visualized in Fig. 6. The effect is created from the alignment of the ground relative to the voxel grid together with an effect shown in Fig. 7a. From that Figure it seems apparent that the problem is the small incident angle but as Fig. 7b shows, similar problems also occur at a high incident angle. The underlying problem is that 3D point cloud data only samples the underlying continuous objects. And it is doing so at different sampling rates per volume depending on the distance from the scanner and the incident angle on a surface.

A solution implies not traversing the lines of sight towards the red points in A2 and A3 in Fig. 7a and b until the last voxel but stopping early enough such that actually static voxels are not marked as free. But where to stop traversing the voxel grid toward a given point must *not* be a function of the point toward which the traversal is done but a function of the points "in front" of it as seen from the scanner position. For example, the problem is not solved by computing the surface normal at a given point and only traversing the line of sight toward that point up to one voxel diagonal away from that surface. This approach solves the problem in Fig. 7a but not the problem shown in Fig. 7b. Instead, the offset in Fig. 7b is determined by the green point in D3.

To calculate this offset or "clipping distance", we create the concept of points closer to the scanner "shadowing" points further away from the scanner. For each point in a scan, we compute whether it's in the "shadow" of a point closer to the scanner and if yes, stop the traversal through the voxel grid at the point that is casting the shadow. Like with real shadows, the "shadow" a point casts is the larger the closer it is to the scanner. Since points by themselves do not have a volume, we choose a sphere with the radius of one voxel diagonal and the casting point in its center as the object casting the shadow. To also cater for situations as shown in Fig. 7a we do not simply clip the rays toward the shadowed points by the distance of the point casting the shadow from

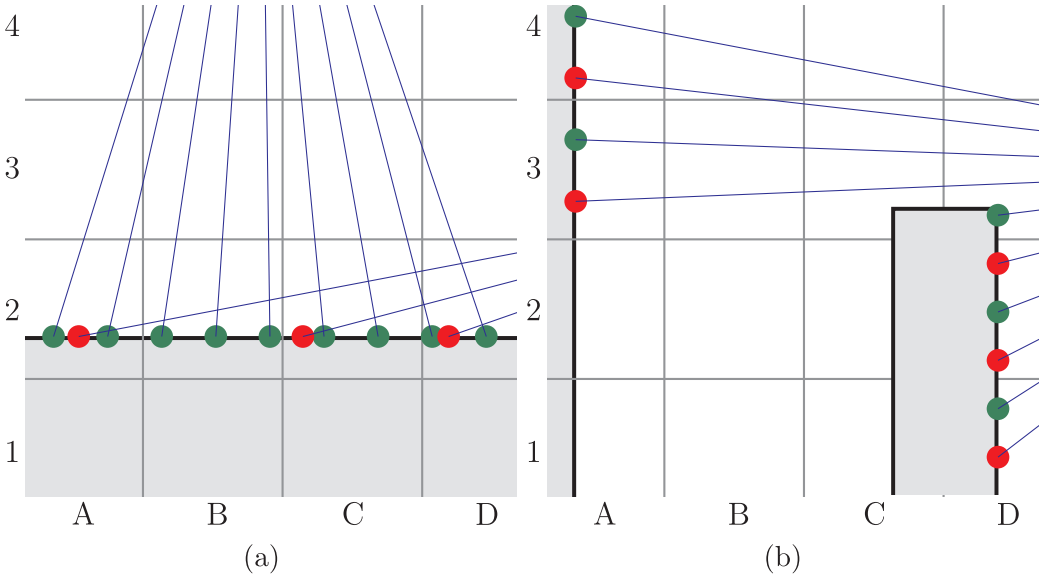


Fig. 7. Two examples for false positive when applying a naive approach to find dynamic voxels. The raster represents the 2D voxel boundaries. Blue lines mark the scanner lines of sight. Dark lines are object boundaries. Gray areas mark solid space while white areas mark free space. Round dots represent the measured scan points of two scans in red and green, respectively. **Left:** Due to the surface being scanned at a shallow angle, the scan measuring the red points will wrongly mark voxel B2 as free when traversing the line of sight up to the red point in A2. **Right:** Due to the tip of the structure in D3 only measured by the green scan, it will be wrongly marked as free when traversing the line of sight up to the red point in A3. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

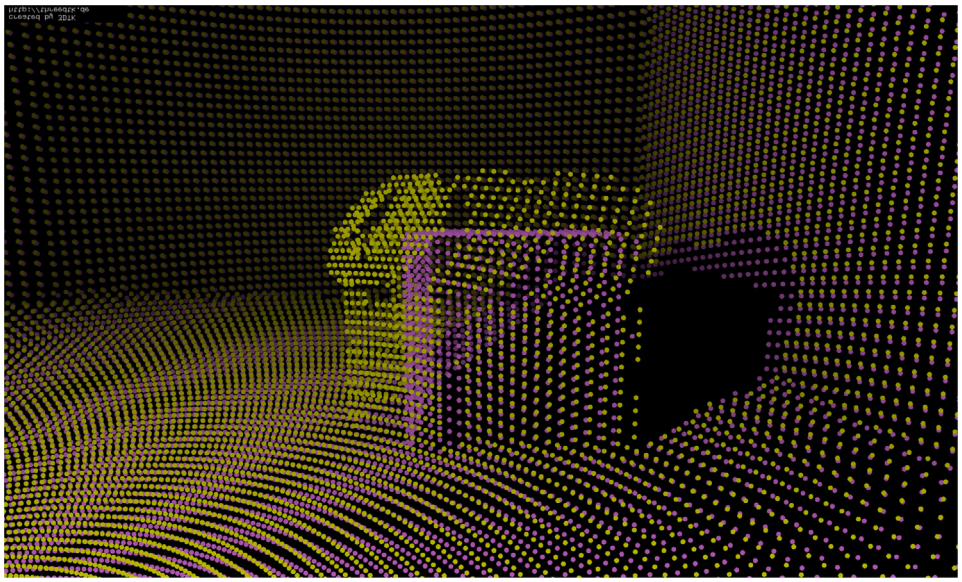


Fig. 8. Synthetic dataset "sim" from Underwood et al. (2013). Magenta points represent the actual dataset: a small cube on the floor of a bigger cube. For each point, the maximum search distance was computed and is represented by a yellow point. The scanner location is in the upper left. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

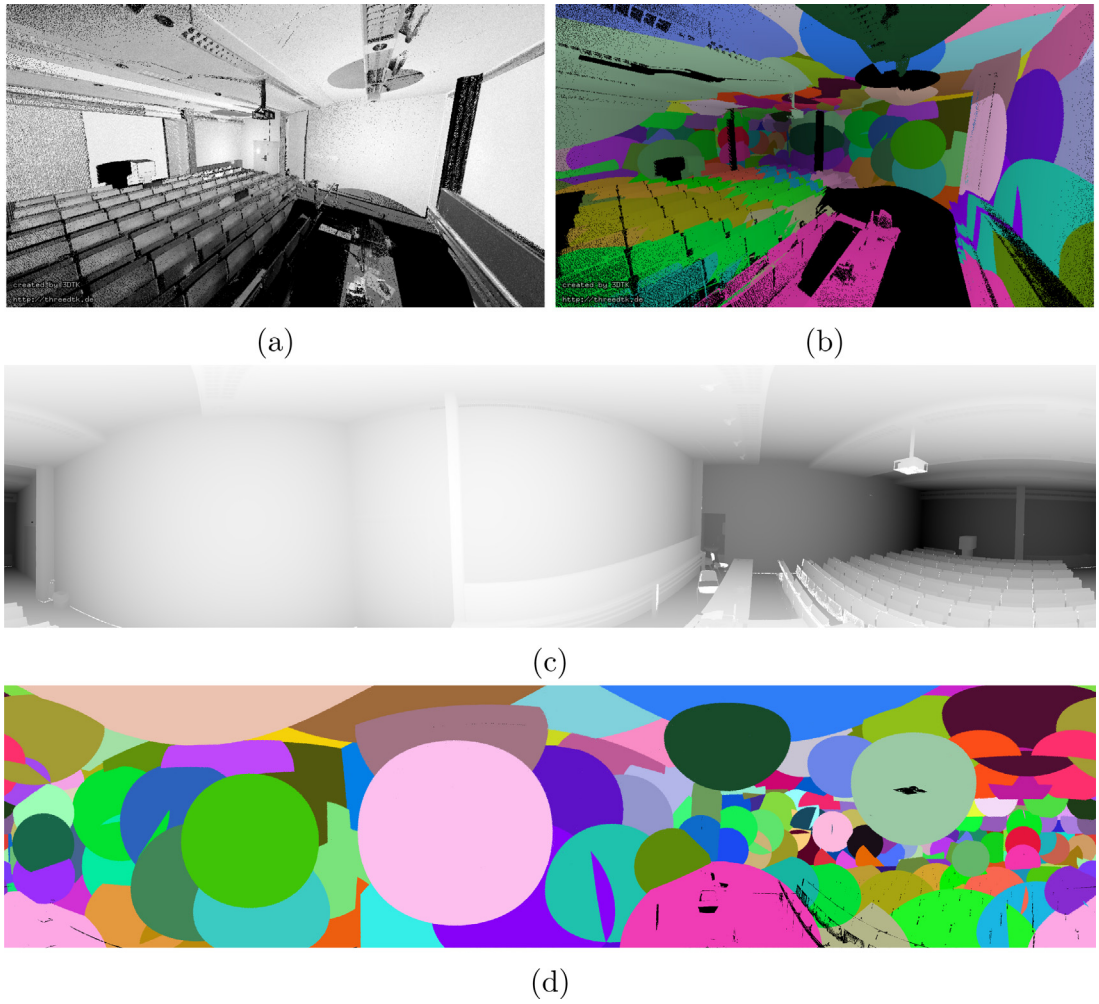
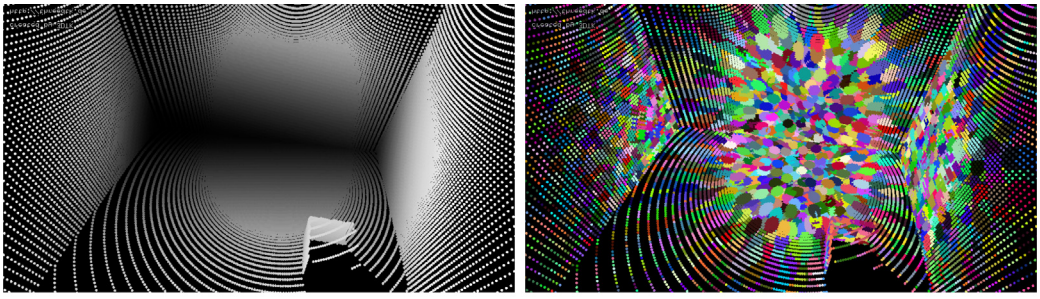


Fig. 9. The lecture hall dataset with points colored such that all points with the same color are shadowed by the same point. For visualization purposes, a very large voxel size of 20 cm was chosen. **Top-Row:** Perspective Projection **Center- and Bottom-Row:** Panorama Projection **Upper right and Bottom:** Colored by point shadows **Upper-Left:** Colored by reflectance **Center:** Colored by point distance. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) (b)

Fig. 10. The sim dataset with points colored such that all points with the same color are shadowed by the same point using a voxel size of 0.6. **Left:** Colored by distance **Right:** Colored by point shadows. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the scanner but instead compute the surface normal at the point casting the shadow and clip the traversal distance of all points in the shadow to be at least one voxel diagonal away from that surface.

Fig. 8 visualizes the idea of shadows clipping the traversal distance to points behind them. The figure comes from the synthetic dataset "sim" from Underwood et al. (2013) and magenta points represent the scene: a small cube on the floor of a bigger cube. The scanner location is in the upper left of the image. Each of the magenta points of the scene has an associated yellow point which represents up to where the line of sight from the scanner toward it will be traversed. The corner of the cube in the center of the image shows a disk of yellow points. The disk is created because we let a sphere cast the shadow and the orientation of the disk results from the value of the normal vector at the corner of the cube. Further to the right, two more disks are visible, shadowing more points. The shadow is explicitly visible in the background to the right. The effect of creating disk-shaped shadows is only present at the corner points closest to the scanner. For the rest of the scene, the flat surfaces of the environment are shadowed by flat surfaces as well. The surface of the scene is quasi "eroded" into the empty space to create the traversal offset for each point while at the same time taking surface normals into account.

Figs. 9 and 10 visualize which points shadow other points in a real scene and a synthetic dataset, respectively. Every point with the same color is shadowed by a common point. The colored shapes are elliptical disks representing cuts of a cone. The cone shape is the volume that is shadowed by a given point. Points closer to the scanner shadow a larger volume and thus create bigger blobs of color in these figures. The dependency is best visible from the panorama images in Figs. 9a and b. Bright areas in the upper panorama images represent points close to the scanner. These areas create big shadows as is shown in the lower panorama image. Darker points are further away and create smaller shadows.

5.1. Implementation

It is very costly to iterate over all points in a scan and for each one find the point which potentially shadows them, especially because no such point may exist and also because the shadow size of each point varies by its distance from the scanner. Thus, instead of determining which point shadows a given point, we sort all points by distance from the scanner and then find all points falling into each of their shadows. By not processing points which already fell into the shadow of another point, only very few computations are required even for large scans because usually few points in the "foreground" shadow many points in the "background".

Algorithm 4. Compute maximum search ranges for every point in a scan

1: **for** $p \leftarrow$ SORTPOINTSBYDISTANCE$points$ **do**
2: **if** $maxrange[p]$ **then**
3: **continue** ▷ Point was already processed
4: **if** $|p| < voxeldiagonal$ **then**
5: **error** ▷ Point is too close to the scanner
6: $angle \leftarrow 2 \cdot \arcsin\left(\frac{voxeldiagonal}{|p| - voxeldiagonal}\right)$
7: $neighbors \leftarrow AngularRangeSearch points, p, angle$
8: $normal \leftarrow CalcNormneighbors$
9: $anglecos \leftarrow normal \cdot \|p\|$
10: **if** $anglecos >= 0$ **then**
11: $normal \leftarrow -1 \cdot normal$ ▷ Normal vector toward scanner
12: $pbase \leftarrow p + voxeldiagonal \cdot normal$ ▷ plane base
13: $dividend \leftarrow pbase \cdot normal$
14: $divisor \leftarrow normal \cdot \|p\|$
15: **if** $divisor = 0$ **then** ▷ Parallel case
16: $maxrange[p] = 0$
17: **continue**
18: $maxrange[p] = dividend/divisor$
19: **if** $maxrange[p] < 0$ ▷ Scanner behind plane
20: $maxrange[p] = 0$
21: **for** $q \leftarrow neighbors$ **do**
22: **if** $p = q$ ▷ Skip the current point
23: **continue**
24: $divisor \leftarrow normal \cdot \|q\|$
25: **if** $divisor = 0$ **then** ▷ Parallel case
26: **continue**
27: $d \leftarrow \frac{dividend}{divisor}$
28: **if** $d > |q|$ **then** ▷ Don't lengthen
29: **continue**
30: **if** $d < 0$ ▷ Scanner behind plane
31: $d \leftarrow 0$
32: **if** $maxrange[q] < d$ **then** ▷ Already shadowed by a closer one
33: **continue**
34: $maxrange[q] \leftarrow d$

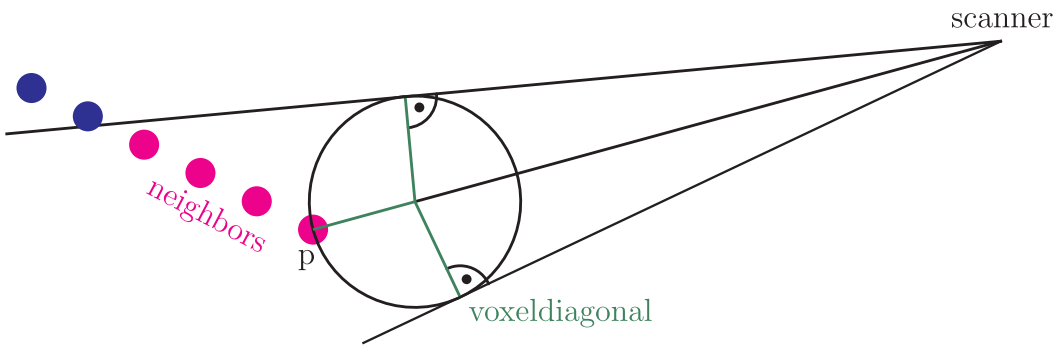


Fig. 11. A scanner on the right measured the points in blue and magenta on the left. All distances equal to one voxel diagonal are highlighted in green. This visualizes lines 6 and 7 from Algorithm 4. The closest point to the scanner p gets processed. Points falling into the shadow of p created by the sphere in front of it are marked in magenta. Remaining points are marked in blue. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Algorithm 4 computes for each input point in the array *points* the distance up to which the line of sight from the scanner toward that point will be traversed. All members of the array *points* are given in the local scanner coordinate system and the results are stored in the associative array *maxrange*. The algorithm uses three additional functions. SORTPOINTSBYDISTANCE sorts the input points by their distance from the scanner. Since the points are given in the local scanner coordinate system, this amounts to comparing vector lengths. The function ANGULARRANGESEARCH returns all points which are seen under a given angular radius around a given point from the perspective of the scanner. The function CALCNORM computes a normal vector of the given input points via singular value decomposition of the covariance matrix of the input points.

The loop iterates over all the points of a single scan. The traversal has to be done in ascending order of their distance from the scanner. By doing so and by skipping points that were already handled (line 3) the procedure finishes very quickly as only a small subset of points actually has to go beyond line 3.

Fig. 11 visualizes lines 6 and 7 from Algorithm 4. Point p is processed first by computing the angle under which the scanner sees a sphere with its center one voxel diagonal in front of p and with the radius of one voxel diagonal (line 6). The function ANGULARRANGESEARCH then finds the magenta points as neighbors of p in line 7.

The neighbors are then used by CALCNORM to compute their normal vector (line 8) which is ensured to point toward the scanner (line 11). The base of a plane is then computed in line 12 and visualized in Fig. 12. That base lies one voxel diagonal away from p in the direction of the computed normal vector of the neighbor points. Lines 13 to 20 then compute the distance up to which the voxel grid will be traversed towards p and stored in the associative array *maxrange*$[p]$. As given in Fig. 12, the search distance is clipped to the intersection of the computed plane with the line connecting the scanner and p. Lines 13 and 14 compute the intersection using the algebraic method of computing line/plane intersections. Line 15 and 19 cater for two rare cases. Should the plane either be parallel to the scanner or should the scanner be located behind the plane, then the distance from the scanner up to p will not be traversed through the voxel grid and thus *maxrange*$[p]$ is set to zero.

Fig. 13 visualizes lines 21 to 34 from Algorithm 4. The loop iterates over all points q in the angular neighborhood of p except p itself. For each point q, the intersection with the plane is computed. For that intersection test, only the divisor has to be updated for each q as the dividend contains the plane properties and stays the same. The same tests for parallelism and the distance being negative are done for q as they were for p. Additional checks include to not lengthen the traversal distance (line 28) and to take care not to update a maxrange with a larger value than what might've been computed in an earlier iteration (line 32).

6. Sphere quadtree

To efficiently compute the result of the function ANGULARRANGESEARCH from Algorithm 4 we use triangle quadtrees on a sphere surface, also known as hierarchical triangular meshes (Szalay et al., 2007) or sphere quadtrees (Fekete, 1990). That data structure has so far mostly been used for Geographic Information Systems to model features on top of the earth surface (Goodchild and Shiren, 1992) or in astronomy to map objects in the sky (Budavári et al., 2010). For our purposes we use it as a search tree to find all points in a certain angular neighborhood in a terrestrial panorama scan with an average lookup complexity of $O(\log n)$. We create one sphere quadtree per scan with its center at the origin of the scanner-local coordinate system.

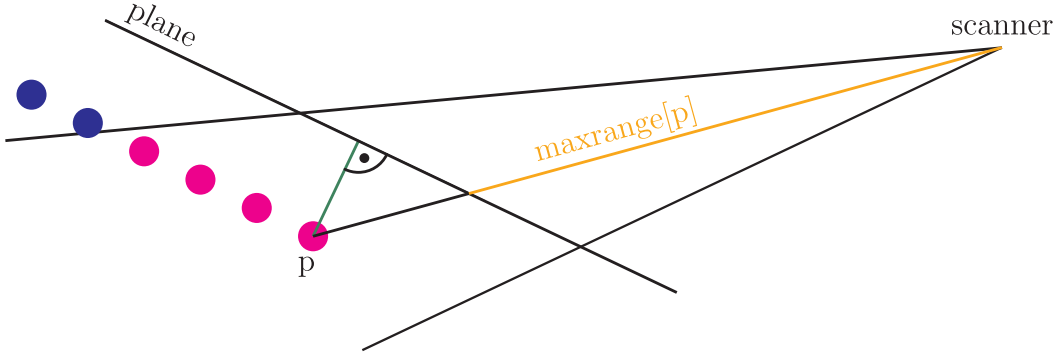


Fig. 12. A plane is added, orthogonal to the normal vector of the magenta points and one voxel diagonal away from them in scanner direction. That plane is then used to clip the search distance through the voxel grid towards p (shown in orange). This visualizes lines 8 to 20 from Algorithm 4. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

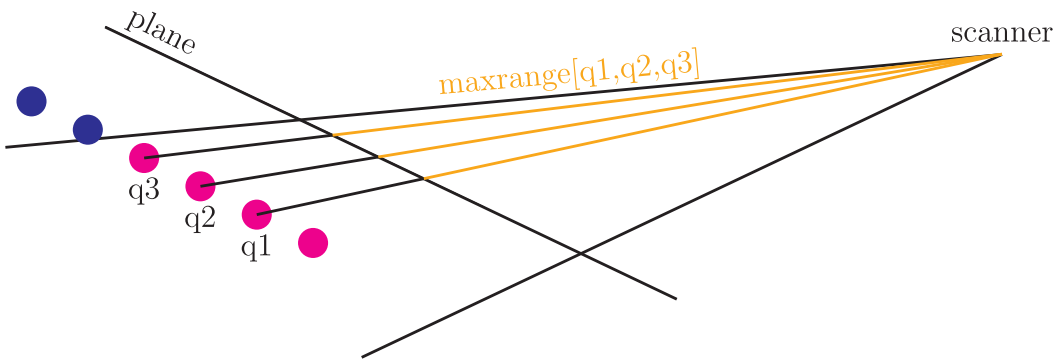


Fig. 13. The plane is also used to cap the search distance up to the angular range neighbors of p. The neighbors of p are $q1$, $q2$ and $q3$ marked in magenta and the maximum search distance marked in orange. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

To this end, we project all points of a single scan onto the surface of a unit sphere. This operation is done by computing the normalized vector of each point and thus, points in the data structure are stored in Euclidean coordinates of normal vectors. Storing points with their length normalized avoids to repeatedly normalize them for angle computation when searching the data structure. Since sphere quad tree queries only require the angular position, discarding the distance information of the inserted points does not pose any disadvantage.

The data structure consists of eight quadtrees. Each quad tree recursively subdivides an eighth of a sphere surface into triangles where each triangle is subdivided into four more triangles until leaf nodes in the graph contain no more than a certain maximum number of points. We chose the eight sides of an octahedron as the top-level structure providing the base triangles for the sphere quadtrees because by aligning the octahedron with the coordinate axis, it is very efficient to decide for every new point p into which octree to insert it. We achieve this by arranging the eight octrees in an array of length eight in an ordering such that the index of the octree into which each new point has to go is computed using only 13 instructions in assembly and without any branching:

$$idx = (p.\,x > 0) \ll 2 \| (p.\,y > 0) \ll 1 \| (p.\,z > 0)$$

Using an icosahedron as the base structure results in a more uniform distribution of triangles over the sphere surface but at the expense of much more costly geometric computations once a new point is inserted or once the data structure is queried for angular neighbors.

A given triangle is subdivided into four new triangles by averaging the vertices of each pair of vertices making the three sides of the triangle and normalizing the result. The three new vertices together with the three existing vertices then form the four new triangles and thus the four new quadtree nodes. Because of the normalization step, all vertices of all triangles remain on the surface of the unit sphere. Points are sorted into each of the four new triangles by computing the triple product of the point with the three newly created edges. Thus, on average, 1.5 computations of the triple product are required on each level of the triangle quadtree to reach the right leaf node to insert the new point into.

A visualization of how the points from a terrestrial scan are inserted into the data structure is shown in Figs. 14 and 15. Each subfigure shows one additional recursion step. Fig. 15b shows the reflectance values of the recorded points of the original input scan projected on a perfect sphere in the same orientation as the sphere quadtrees are displayed. Particularly Fig. 15a allows to clearly recognize the shape of the scanned buildings shown in Fig. 15b. The density of subdivisions per sphere surface stems from the structure of the underlying data. The unmodified input data from the laser range finder results in a very homogeneous subdivision of the sphere quad trees because of the regular angular resolution of the laser beam sweeps. Thus, for visualization purposes we reduced the input data to 10 random points per 30 cm voxel before inserting it into the octrees. Due to this reduction step, more points are seen under the same angle if the points are further away from the scanner location. This leads to a higher triangle subdivision in regions of far-away points.

7. Clustering for noise removal

In certain situations the voxel traversal algorithm will result in false positives: voxels marked as free even though they contain static points. These situations arise if no good normal vector could be computed from the underlying point cloud data. False positives usually manifest themselves in only one or two adjacent voxels being marked as free. Most true positives are groups of connected voxels of much larger number. Thus, to reduce the number of false positives we cluster the set of voxels that were marked as dynamic and then remove those clusters with a number of voxels below a certain threshold from that set. The method introduces new false negatives in situations where moving objects in a scene occupy less volume than the given threshold.

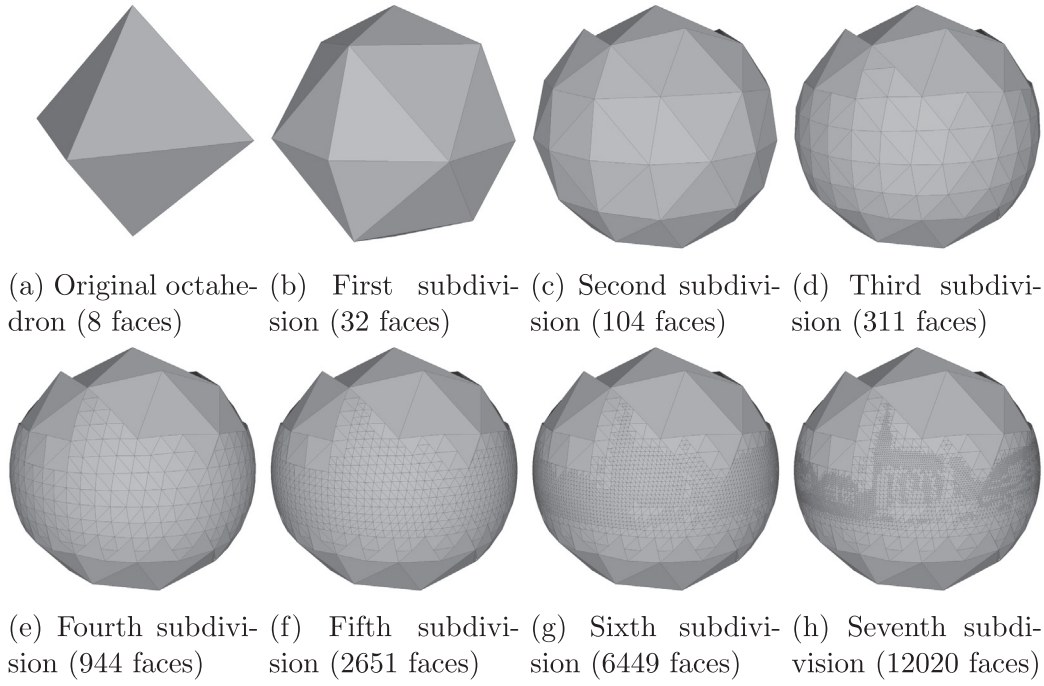
We define clusters through the neighborhood relationship between voxels. We consider the neighbors of a voxel as all voxel adjacent to it or more precisely:

Definition 3. A voxel A is a neighbor of another voxel B if each coordinate component of A does not differ from the respective coordinate component of B by more than 1.

This means that every voxel has 26 neighbors: six adjacent to its sides, twelve adjacent to its edges and eight adjacent to its corners. We then assign the same cluster identifier to all groups of voxels that share a transitive neighborhood relationship. Or in other words: different clusters are separated by at least one free voxel between them.

Due to the voxel datastructure, computing the cluster identifier that each dynamic voxel belongs to is straight forward: We iterate through all voxels that were marked as free and then for each voxel, identify the clusters its neighbors belong to. If no neighbor belongs to a cluster, the current voxel will start a new cluster. If only one cluster was found in the neighborhood, then the current voxel is added to it. If more than one cluster was found in the neighborhood, then all these clusters are merged into a single cluster and the current voxel is added to it.

This clustering technique is very fast not only because of its linear computational complexity but also because typical scenes only contain comparatively few dynamic voxels. Finally, clustering by voxels allows quick clustering of the underlying points which may be an order of magnitude more in number while taking advantage of the already existing voxel data structure. Solutions working on the raw point data for clustering are understandably slower.



(a) Original octahedron (8 faces) (b) First subdivision (32 faces) (c) Second subdivision (104 faces) (d) Third subdivision (311 faces)

(e) Fourth subdivision (944 faces) (f) Fifth subdivision (2651 faces) (g) Sixth subdivision (6449 faces) (h) Seventh subdivision (12020 faces)

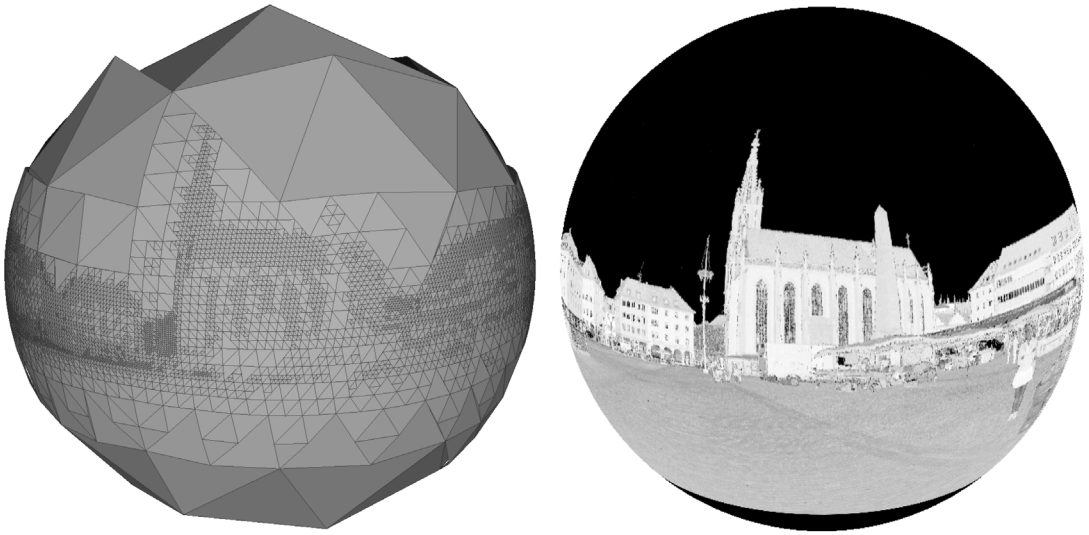
Fig. 14. Starting from an octahedron, triangles are recursively subdivided into four new triangles (with vertices on the surface of a unit sphere) until less than a given number of points recorded by the scanner (maximum leaf-node size was 100) falls into each triangle.

8. Sub-voxel accuracy

In this section we present an algorithm that addresses a specific kind of false negatives our algorithm produces. In the common case where a dynamic object is seen directly adjacent to a static object, false negatives are introduced because the voxel grid is only traversed up to the maximum traversal range computed from the point shadows. For example, for a person standing on the ground, the person might be removed but their feet remain.

To avoid these false negatives we introduce an algorithm that is able to produce a result with sub-voxel accuracy: instead of marking a full voxel as dynamic and removing all points from it, we just remove a subset of points from a voxel. That subset will include the dynamic points that were not marked before and thus reduce the amount of false negatives.

We use Fig. 16 to illustrate our approach to achieve subvoxel accuracy and reduce the number of false negative classifications. The input is shown in Fig. 16a. The green scan only measures the static horizontal surface while the red scan measures parts of the static surface and a vertical dynamic structure. After walking the voxel grid to find voxels seen as free by the scan resulting in the green points we end up with the situation displayed in Fig. 16b. Voxels B3, B4, C3 and C4 got correctly classified and points in them were removed. What remains are false negatives in voxel B2 and C2. Classifying these voxels as



(a) (b)

Fig. 15. Visualization of the nodes of a sphere octree from a terrestrial scan on the left with the reflectance values of that scan as a texture on a sphere in the same orientation on the right. The final data structure contains 580,000 points. **Left:** Eighth and final subdivision (13,769 faces) **Right:** A perfect sphere textured with the reflectance values of the laser scan.

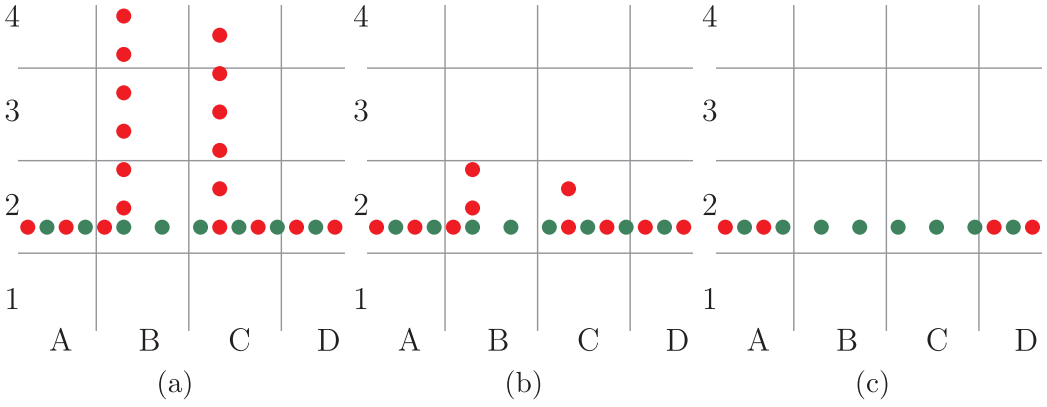


Fig. 16. Two scans (red and green points) of a horizontal surface and dynamic points only seen in the red scan. **Left:** The original input with both scans (nothing removed). **Center:** Voxels B3, B4, C3, and C4 were marked as see-through and thus got their points removed. Artifacts still exist in voxels B2 and C2. **Right:** Voxels B2 and C2 got the red points removed because the red points were removed from an adjacent voxel. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

dynamic is wrong because that would also remove points that were correctly measured by the green scan as part of the static horizontal surface.

The algorithm we use to remove these false negatives from voxels B2 and C2 will compute a situation as is seen in Fig. 16c: all red points are removed from voxels B2 and C2. This removes the false negatives while at the same time introducing some false positives because some of the red points in voxels B2 and C2 were correctly classified as static. Thus, our approach to achieve subvoxel accuracy implements a trade-off. We remove remaining false negatives at the cost of more false positives. We accept this trade-off because qualitatively speaking the result shown in Fig. 16c is superior to the result in Fig. 16b. Even though we now classified too many points as dynamic, after removing them from the scene, there are still enough static (green) points left in the respective voxels to not create any “holes” in the scene. Thus, our approach to achieve subvoxel accuracy is particularly interesting for situations where our algorithm is used to acquire a scan that only contains the static environment. Another possible use case are situations in which one is interested in extracting point clouds of individual moving objects for later processing. In that case, extracting too few points would result in an incomplete pointcloud model. Quantitatively speaking the algorithm worsens the result. When comparing the raw F_1-scores of the results before and after applying the algorithm for subvoxel accuracy, the F_1-score is typically worse afterwards due to the introduction of more false positives.

The algorithm works as follows: similarly to the clustering algorithm, we iterate over all voxels that were marked as free. For each of these free voxels we record which scan identifier it contains. For voxel B3 in Fig. 16 that is just a single scan identifier: ”red”. First, we gather the neighbor voxels according to Definition 3. Secondly, we iterate over all neighbor voxels that were classified as static. Thirdly, for each of the static neighbor voxels we remove all the points coming from scan identifiers marked as free in the original voxel. For Fig. 16 this removes the red points from voxel B2. In summary, the algorithm deletes points belonging to a scan that was found in an adjacent dynamic voxel from each static voxel. To completely prevent that “holes” in the scan are created by this method, we never remove points from voxels which would not contain any points anymore after the removal.

A qualitative comparison of a scene with and without the subvoxel algorithm applied is found in Fig. 17. As shown in the upper two images, without the algorithm, artifacts are remaining close to the ground. These artifacts are removed in the lower two images. Points from the ground are also missing but not visible because the respective voxels still contain points from all the other scans that measured that area of the ground.

9. Results

In this section we present how our algorithm performs in quantitative and qualitative terms as well as in terms of runtime on commodity hardware. We do this by showing the results of running the algorithm by itself as well as by comparing it with other solutions in terms of runtime and solution quality.

9.1. Quantitative assessment

To quantify the output of our approach we made use of four datasets with points that come with labels indicating whether they are dynamic or static. We use these annotations to accurately compute and compare false positives and negatives, precision, recall and the F_1-score. Three datasets were provided[5] together with a competing change detection implementation by Underwood et al. (2013). The fourth dataset “lecturehall” was recorded by us to have a high resolution and high precision dataset only consisting of two scans. The quantitative results in this section can be reproduced by executing the `run.sh` shell script that we provide for download.[6] The script will download and compile our software as well as the software by Underwood et al., download the necessary datasets and finally run both solutions on each dataset.

The four datasets are shown in Fig. 18. We list the number of points and the number of scans of each dataset in Table 1. The last column lists the number of comparisons that we carried out when running the algorithm by Underwood et al. on the datasets. The number is usually equal to $\frac{N(N-1)}{2}$ because all scans overlap. The only exception is the campus dataset where we selected only the scan pairs that shared a significant overlap.

The sim dataset in Fig. 18a is a synthetic dataset where virtual laser range finders measure a cube in one of two positions from four different vantage points. The lab dataset in Fig. 18b is from a robot moving through a cluttered lab environment with small boxes being present or not at multiple locations. The third dataset carpark from Fig. 18d consists of four stationary scans in a carpark environment where a car was moved into different positions for each scan. The fourth dataset lecturehall is shown in Fig. 18e and is a small lecture hall scanned from two vantage points by a Riegl VZ-400 laser scanner. One of the scans has two people in it while the other does not. The campus dataset was recorded with the same scanner and Fig. 18f shows a top view of the registered complete pointcloud. Renderings of the Würzburg dataset are shown in Figs. 22 and 24. The last two datasets come without labels indicating a ground truth for static and dynamic points.

In contrast to the results shown in the paper by Underwood et al. we pass all points of the three Underwood datasets into each algorithm and not only a subset of them. The third scan of the carpark dataset was wrongly aligned, so we registered the scans again using the ICP implementation from `slam6D` before passing the points to each algorithm.

We computed the results without running clustering for noise removal in the end. Since the clustering algorithms are in principle

[5] http://www.acfr.usyd.edu.au/papers/icra13-underwood-changedetection.shtml.
[6] https://robotik.informatik.uni-wuerzburg.de/telematics/download/isprs2018/.

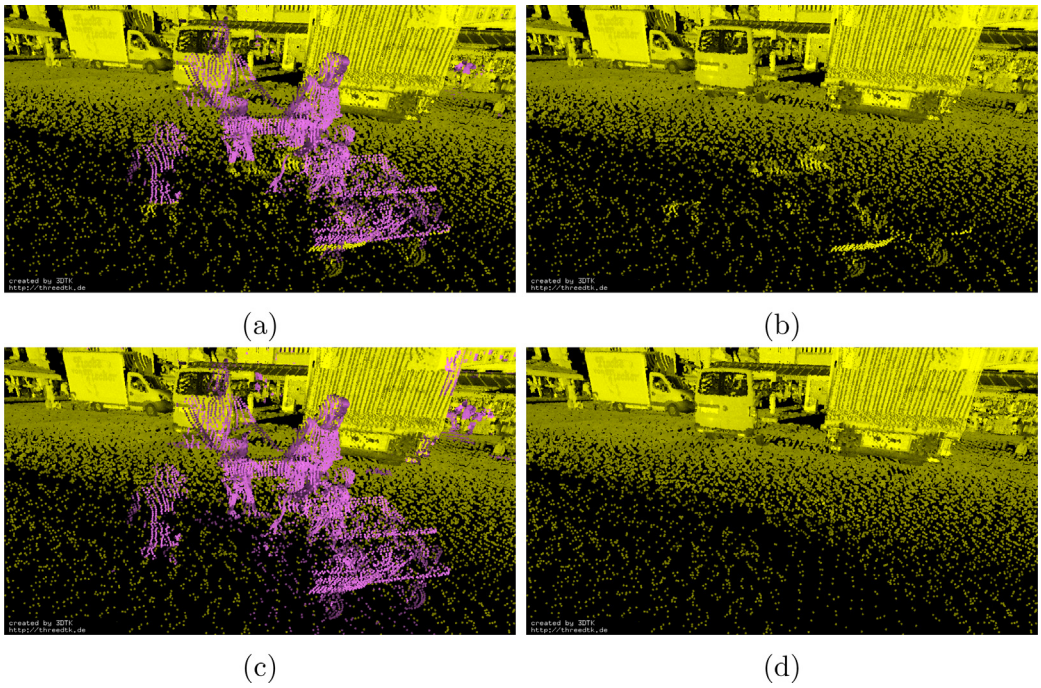


Fig. 17. Result of applying the algorithm to achieve sub-voxel accuracy on an actual dataset. Dynamic points are magenta while static points are yellow. **Upper-Left:** No subvoxel accuracy with dynamic points in magenta. **Upper-Right:** No subvoxel accuracy with leftover false negatives on the ground. **Lower-Left:** With subvoxel accuracy and dynamic points in magenta. **Lower-Right:** With subvoxel accuracy no false negatives remain on the ground. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

independent of the method that was used to partition the input point cloud into static and dynamic points, it would not allow to make any meaningful statements about the respective underlying change detection algorithms anymore. Furthermore, by choosing the correct cluster size for true positives, it is possible to achieve nearly ideal results with any algorithm that produces only few false negatives which is the case for both compared algorithms.

The algorithm by Underwood et al. was executed in the variant that compares individual pairs of scans. This choice was made because the results in the respective paper suggest better F_1-scores in the non-clustering case when working on pairs compared to combining multiple scans. Pairs were chosen such that the measured scene is always different between the two scans. To achieve optimal results, we ran their algorithm on a discrete set of parameters T_a and $T_r(m)$ to find the combination yielding optimal results.

We ran our algorithm multiple times as well, each time with different voxel sizes. We didn't apply our approach of achieving sub-voxel accuracy because it can lower the F_1-score.

The results are shown in Table 2. We achieve similar F_1-scores on the synthetic "sim" dataset. False negatives are introduced in our method due to the alignment of the floor with the voxel grid, preventing a perfect score. Our method is outperformed in the "lab" dataset. The dataset is challenging because of its very noisy nature (see Fig. 18f) and because the dynamic objects are very small. Our algorithm correctly identifies the moving boxes in the "lab" dataset and does not introduce false negatives. But it generates comparatively large number of false positives on corners and edges of the environment. Since only 0.19% of all points in the dataset are labeled as dynamic, it only requires few voxels marked as false positives to produce a bad F_1-score. Our method slightly outperforms the approach by Underwood et al. in the "carpark" dataset. The best F_1-score we achieved for the carpark dataset with the Underwood method differs from the value they present in their paper because we used their full dataset including the last scan as well as all scan lines. Both methods result in equal scores on the "lecturehall" dataset.

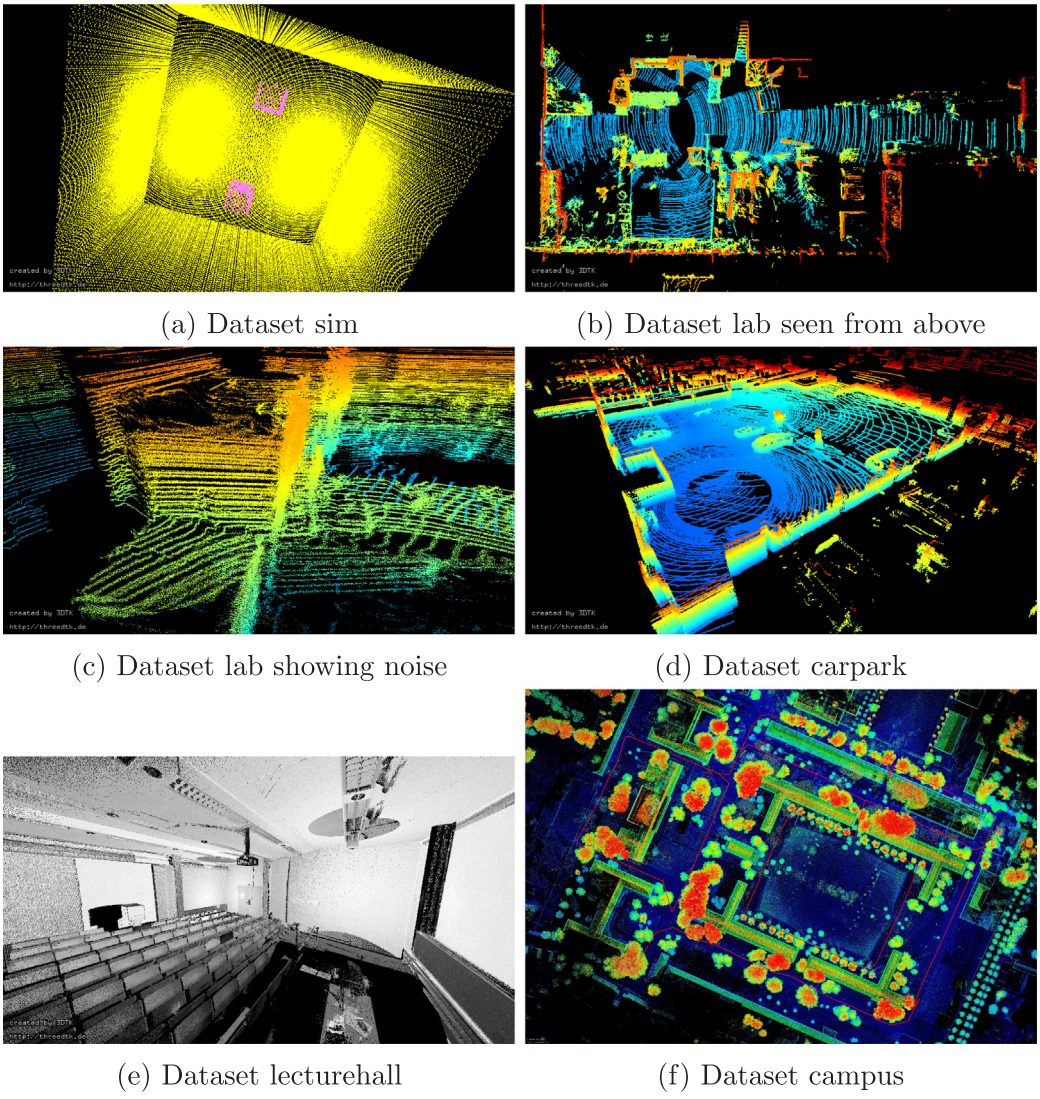
Our algorithm produces false positives in situations where scans are either not correctly registered or due to sensor noise. An example is a flat surface where not all points lie on the surface. The points "in front" of the surface in scanner direction will then be marked as "see through" even though they belong to a static object. Another source of false positives arises when surface normals are wrongly computed and thus point shadows are not determined correctly. This in turn will lead to false positives as they were shown in Fig. 7. Due to the very noisy nature of the "lab" dataset there were many sources of both of these issues, leading to a high number of false positives. Another source of false positives are mirrors and transparent objects. Lastly – if enabled – some false positives are introduced by our approach to subvoxel accuracy.

False negatives are created either in situations where a volume was only seen by a single laser scan or in volumes that were "shadowed" by closer points. We observed the latter problem in a dataset where we placed the scanner directly on the ground instead of on a tripod to take a scan. This resulted in points from the ground directly adjacent to the scanner to shadow most of the lower part of the scan and thus make it impossible for our algorithm to classify any points close to the ground as dynamic. Additionally, false negatives are introduced if the chosen voxel size is so small, that rays are able to penetrate objects without intersecting a voxel with points in it. Since the point density typically decreases with their distance from the sensor, this effect also occurs at very far distances. Applying a clustering filter can also introduce false negatives if the dynamic object is smaller than the chosen minimum cluster size.

We also observe how the optimal input parameters to the algorithms T_a, T_r and the voxel size are different for each dataset despite the lab and the carpark dataset being recorded with the same sensor. More research is needed to determine if the input parameters may be predicted upfront without requiring manual labeling of a training dataset.

9.1.1. F_1-score by voxel size

The only variable of our algorithm is the voxel size. We display the dependency of the F_1-score on the voxel size in Fig. 19. Since the computation of voxel shadows and ray traversal ranges is essential for our approach, the Figure also shows the F_1-scores yielded from different methods to acquire the point set for computation of the normal vector. Since our main method described in Section 5 uses all points seen under a certain angle for normal computation we call that method "angle" in Fig. 19. This method consistently achieved the best quantitative results on all datasets we tested our method on. Additionally, it is also the



(a) Dataset sim (b) Dataset lab seen from above

(c) Dataset lab showing noise (d) Dataset carpark

(e) Dataset lecturehall (f) Dataset campus

Fig. 18. Datasets from Underwood et al. (2013) and our own dataset lecturehall.

Table 1
Overview of the used datasets and their properties.

name	#points	#scans	#cmp
sim	387838	8	28
lab	5815910	12	66
carpark	1965017	4	6
lecturehall	44574647	2	1
campus	2227455077	146	3456
würzburg	86585411	6	15

Table 2
Comparison of F_1-scores achieved by our method compared with the method by Underwood et al.

dataset	Underwood			3DTK	
name	T_a	$T_r(m)$	F_1-score	voxel size(m)	F_1-score
sim	1.4	0.1	**0.98**	0.6	**0.98**
lab	1.2	0.2	**0.71**	0.175	0.42
carpark	1.0	0.35	0.78	0.125	**0.83**
lecturehall	0.8	0.3	**0.96**	0.1	**0.96**

fastest method which is explained by it being the only method that doesn't require an additional search tree to be computed. All the other methods execute searches in a k-d tree which stores and queries points by their cartesian coordinates and not their angular coordinates. The "knearest" and "range" methods compute the points neighbors for normal computation by finding the k nearest points around the query point or by retrieving all points in a radius of one voxel diagonal, respectively. The "knearest-global" and "range-global" methods do the same but using a k-d tree that was computed for the global point cloud instead of operating on the point cloud for each individual scan. The "1nearest" method completely bypasses normal computation and in contrast to all the other methods does not utilize the algorithm displayed in Section 5 for computing the maximum traversal distances toward each point at all. Instead, it operates by finding all points within a radius of one voxel diagonal of the line of sight toward each point and storing as the maximum traversal distance the distance of the closest point from the scanner inside this volume. Since this method requires a k-d tree query for every single point in the dataset it is the slowest of all the methods.

9.1.2. F_1-score by rotation and translation

Due to discretizing the measured volume by a voxel grid we expect the quality of our solution to heavily depend on how the data is aligned relative to the voxel grid. Thus, we compute the F_1-scores of various

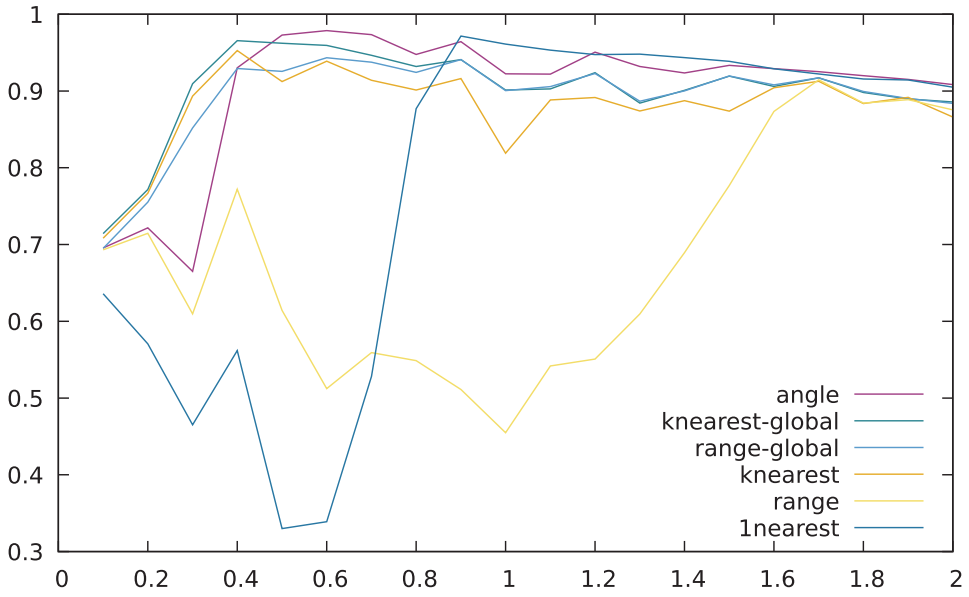


Fig. 19. F_1-score per voxel size for different methods to acquire the points for normal computation in the sim dataset. Voxel sizes differ by 0.1 from each other and measurements are connected with a line for visual clarity.

rotations and translations for the "sim" dataset. We chose the dataset because all its implicit surfaces are orthogonal to each other and should thus yield the most meaningful results.

To compare the influence of rotation on the results we compute overall 1000 rotations of the "sim" dataset around all three coordinate axes with a voxel size of 1.0. Specifically we compute all permutations of rotations between 0 and 45 degrees in five degree steps around all three coordinate axes ($10 \times 10 \times 10 = 1000$). We do not check beyond 45 degrees as the results are symmetric due to the orthogonal nature of the voxel grid.

We visualize our results using the histogram seen in Fig. 20. The figure displays the frequency of the achieved F_1 scores in bins of 0.001 in width. The shape of the histogram suggests a gaussian distribution. Fitting a gaussian function through our data reveals a standard deviation of 0.006. The position of the gaussian at 0.93 aligns with the results we achieve for the voxel size of 1.0 and no rotation. The low standard deviation of our results suggests a negligible influence of the alignment of flat surfaces relative to the voxel grid.

Similarly, we translated the "sim" dataset along all three coordinate axes to evaluate the relationship between the F_1-score and the positional offset of the data relative to the voxel grid. To this end, we computed all permutations of translating the dataset along all three coordinate axes by distances ranging from 0.0 to 1.0 in steps of 0.05. Larger shifts were not investigated because the results repeat themselves due to the chosen voxel size of 1.0.

Evaluating the results for shifts along all three coordinate axes revealed that only translation along one coordinate axis had a considerable effect on the F_1-scores. That axis was the one perpendicular to the ground that the moving boxes are placed upon. This makes sense because false negatives are introduced depending on how much of the volume where each box touches the ground intersects with the voxel that is still part of the ground. We show the F_1-scores for shifts along that axis in Fig. 21. Each displayed measurement represents the accumulated F_1-scores for all shifts along all three coordinate axis with only the chosen axis fixed. The measurements "wrap around" as the value achieved for an offset of 0 are equal to the ones achieved for an offset of 1.0.

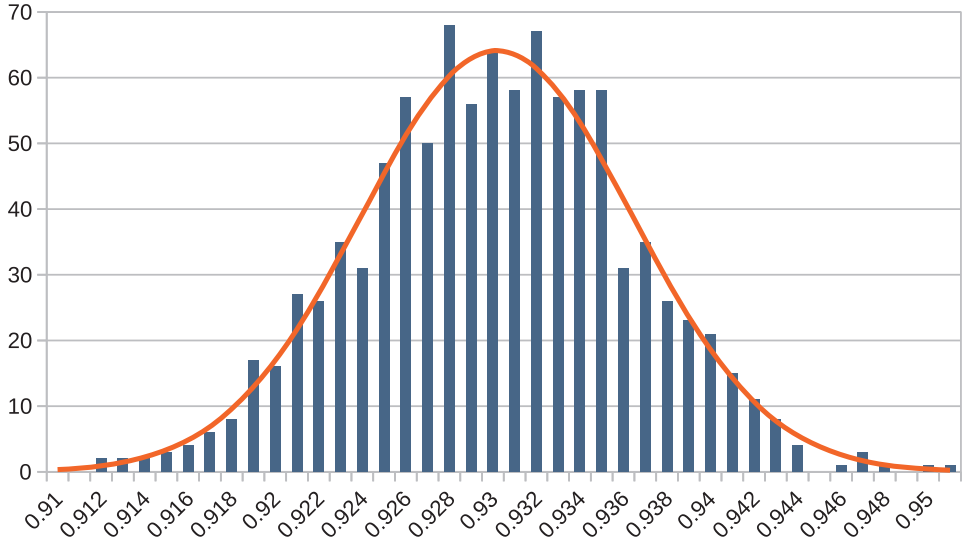


Fig. 20. Histogram of F_1-scores for 1000 permutations of rotations of the input data around all three coordinate axes. The x-axis shows the F_1-scores. The y-axis shows the number of values falling into bins of 0.001 in width. A gaussian is fitted through the measurements.

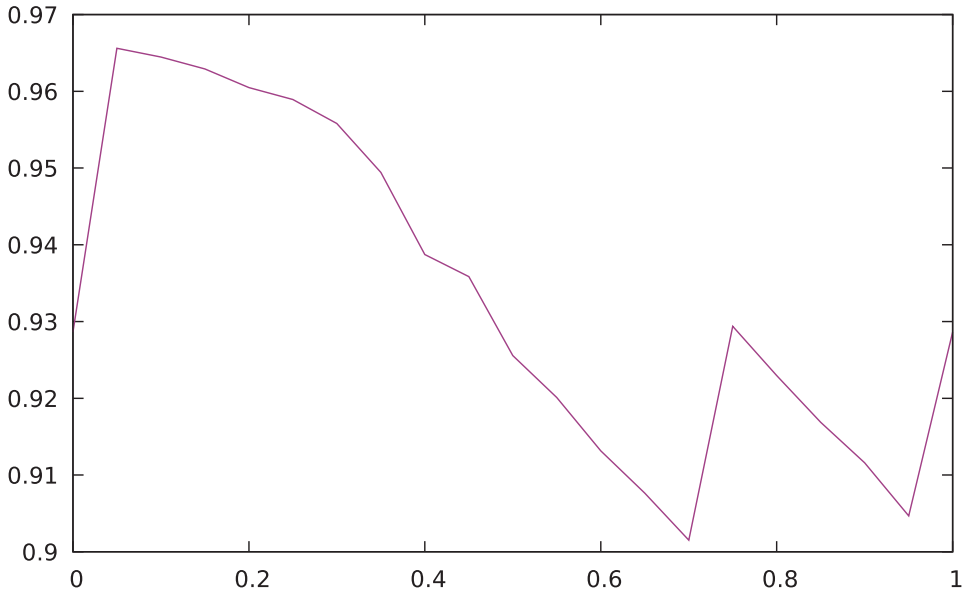


Fig. 21. F_1-scores achieved by translating the input along the axis perpendicular to the plane on which the moving cubes are placed. The x-axis shows the offset along the axis. The y-axis the achieved F_1-score. Measurements are connected with a line for visual clarity.

9.2. Qualitative assessment

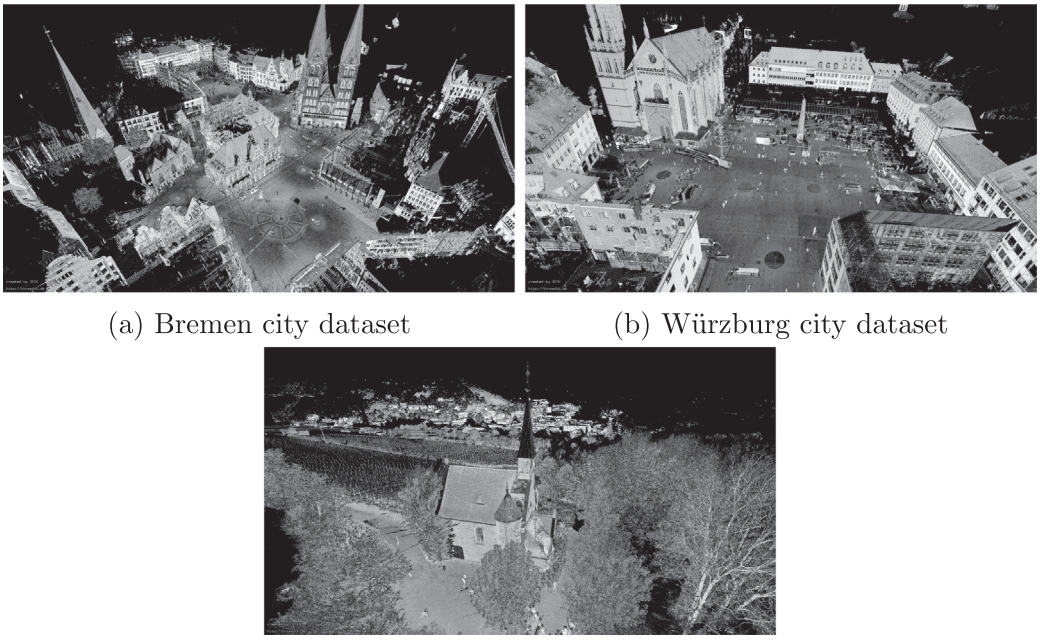
For qualitative analysis we are using the three datasets from Fig. 22. The column "normals" displays the percentage of points for which surface normal computation as part of finding the shadowed points was required. As detailed in Section 5, the computations have to be carried out for only a very small fraction of all input points.

In contrasts to the datasets we used for quantitative analysis, these datasets do not come with any ground truth labeling of points, classifying them whether they are indeed static or dynamic. Thus, without being able to identify false positives and false negatives, F_1-scores cannot be computed.

All datasets were measured using a Riegl VZ-400 laser scanner. The grayscale values represent the measured reflectance. We processed them using a voxel size of 10 cm, a minimum cluster size of 40 voxels and with sub-voxel accuracy enabled. Raw data concerning the number of points and number of scans of each dataset is found in Table 3. The runtimes in the last column of the table were gathered on an Intel Xeon e5-2630 v3 Desktop system with 8 physical cores with 2.4 GHz each and as many threads as there were scans in the dataset. All datasets

Table 3
Overview of the datasets used for qualitative analysis.

name	#points	#scans	normals(%)	t(s)
Bremen city	215652387	13	0.222	2939
Würzburg city	86585411	6	0.21	4967
Randersacker	194754633	11	0.010	1344



(a) Bremen city dataset (b) Würzburg city dataset

(c) Randersacker dataset

Fig. 22. Overview of the datasets used for qualitative analysis.

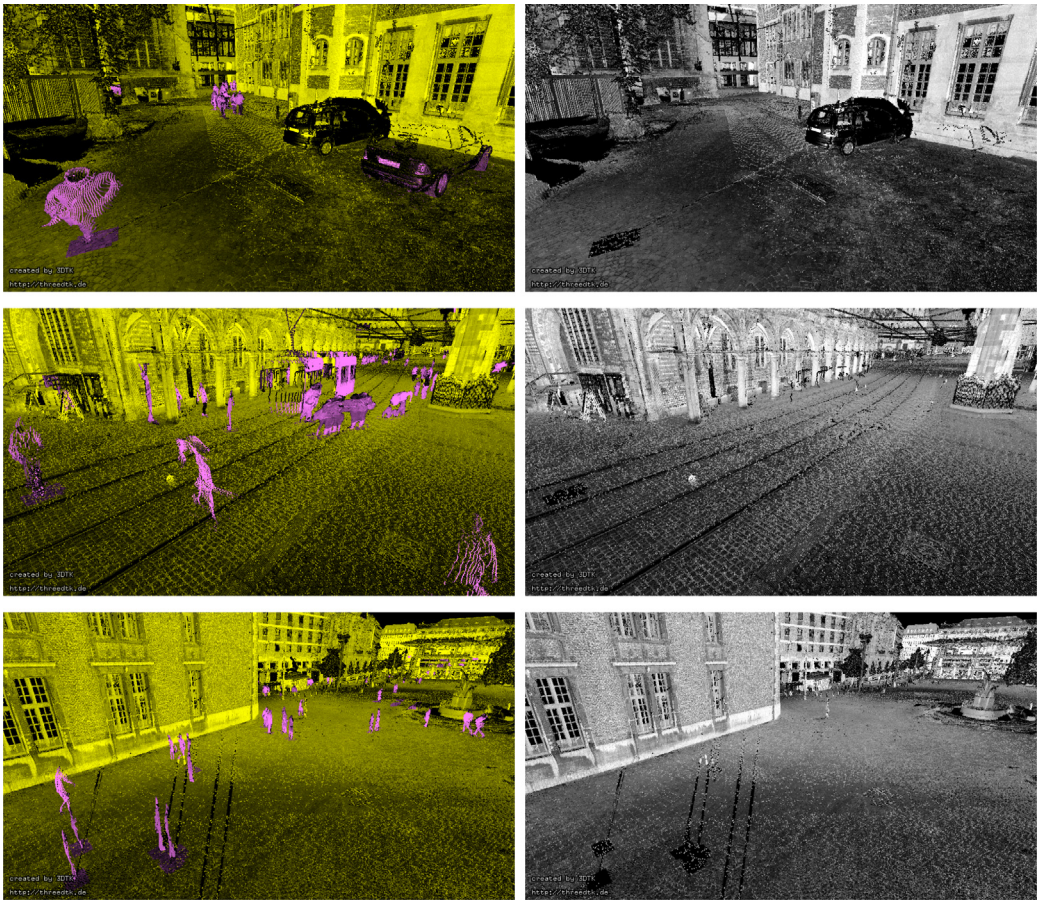


Fig. 23. Bremen city dataset: Each row shows the same camera position. Left column: Static points in yellow and dynamic points in magenta. Right column: scene with only static points. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

were registered using slam6D from *3DTK – The 3D Toolkit*.[7]

We collected the "Würzburg city" dataset specifically for evaluation with our change detection algorithm. Thus, we chose a time of day where the market place was moderately crowded such that enough change is present. Since the quality of our approach is highly sensitive to the quality with which the dataset is registered, we took care to choose very small epsilon and high iteration numbers to achieve the best registration possible for the dataset.

The "Bremen city" dataset is presented to show how our algorithm can be directly applied on datasets that were recorded without our approach in mind. The dataset was recorded in February 2010, seven years before work on our change detection approach started. Furthermore, the dataset shows some small registration errors which we will use to show how they effect the result of our approach in Section 10.

Finally, the "Randersacker" dataset was included because it mainly consists of foliage and other greenery. While normal vectors are easily computed on most surfaces in an urban environment like "Würzburg city" and "Bremen city", we wanted to include a dataset with only few flat surfaces to show how our method performs in them.

Figs. 23 and 24 display the results for the "Bremen city" and "Würzburg city" datasets, respectively. The left-hand-side column shows the original scan partitioned into static (yellow) and dynamic (magenta) points. The right-hand-side column shows the dataset without the points that were identified as dynamic. Since the "Bremen city" dataset was recorded without our algorithm in mind, it was measured very early on a Sunday morning to include as few pedestrians as possible. Thus, it includes considerably less moving objects compared to the "Würzburg city" dataset where we took care to pick a time where the scan area was moderately crowded. Our approach reliably identifies pedestrians, cars, trams and an opened door. Due to our approach to subvoxel-accuracy, no false negatives remain on the ground. After removal of the dynamic objects, no holes are created on the ground.

The results from the "Randersacker" dataset are shown in Fig. 25. Since only few moving objects were present at the time when the dataset was taken, we only present the partitioned rendering with static points in yellow and dynamic points in magenta. Similarly to the urban datasets, moving objects were correctly classified. Both images show how foliage is not classified as dynamic even though in both renderings, the trees were measured by multiple scans. Thus, our algorithm is not only appropriate for urban environments but also for scenes with few flat surfaces.

9.3. Performance

The algorithms that we benchmarked are implemented in C++. For simplicity we use a `std::unordered_map<struct voxel, std::set<size_t>>` data structure to store which voxel coordinate contains points from which scan slice. We tested our algorithm on an Intel Xeon e5-2630 v3 Desktop system with 8 physical cores with 2.4 GHz each.

We conducted a similar experiment to find the dependency of the algorithm runtime from the number of input points. We randomly sampled the first scan of the "lecturehall" dataset to obtain input point clouds ranging from 1 million up to 22 million points and then executed our method on each of the resulting point clouds. The results are shown

[7] http://threedtk.de.

Fig. 24. Würzburg city dataset: Each row shows the same camera position. Left column: Static points in yellow and dynamic points in magenta. Right column: scene with only static points. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in Fig. 26 and again indicate a linear relationship. This makes sense because for the voxel traversal we access voxels in the grid using $O(1)$ operations on a hash map.

For a fair comparison we run everything single threaded even though the method by Underwood et al. has a slight advantage because processes are run connected by a UNIX pipe and thus they are partly executed in parallel. The inputs to both approaches are pointclouds in ASCII text format. Since the algorithms by Underwood require multiple executions of `points-detect-change`, we convert the input into binary format for faster load times.

The runtime measurements shown in Table 4 were obtained by timing the full execution pipeline. To speed up the approach by Underwood et al. we converted the original ASCII point cloud data files into their binary format. As the method by Underwood et al. is only able to compare pairs of scans, the runtime results for the "sim", "lab" and "carpark" datasets are not very meaningful. Our method easily outperforms theirs in terms of runtime because we apply their method on all possible combination of scan pairs, leading to $\frac{N(N-1)}{2}$ comparisons for N scans. For a fairer comparison we recorded the "lecturehall" dataset. It only consists of two scans and thus allows to directly compare one run of the Underwood et al. method with one run of our approach. As listed in Table 4, both approaches require a similar amount of time.

To also give evidence for our claim that the method by Underwood et al. performs slower for the purpose of "scan cleaning" on datasets

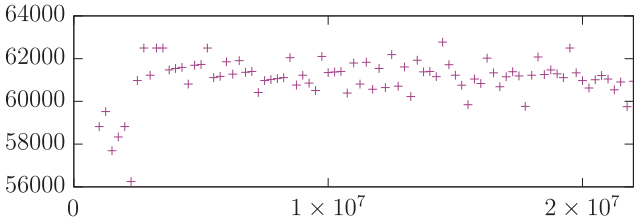


Fig. 26. The x-axis shows the number of points passed to the algorithm. The y-axis shows the number of points that the algorithm is able to process per second.

Table 4
Runtimes of our method versus the method by Underwood

dataset	Underwood	3DTK	
name	$t(s)$	normals(%)	$t(s)$
sim	25	8.03	**6**
lab	405	0.02	**29**
carpark	34	0.23	**23**
lecturehall	837	0.003	**687**
campus	12.8 days	0.16	**13.1 h**
würzburg	7961	0.21	**4967**

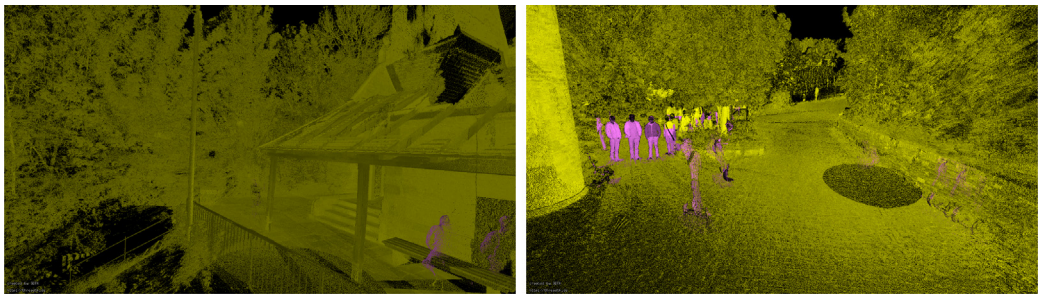


Fig. 25. Randersacker dataset containing lots of foliage. Scans show static points in yellow and dynamic ponits in magenta. Despite not offering a clear surface, foliage is not removed. **Left:** People sitting on a bench in the lower right **Right:** Segways in the foreground and students in the background. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

with many scans, we used the "campus" dataset. Fig. 18f shows the dataset from above. That dataset consists of 146 scans with 15 Million points per scan on average for a total of 2.2 Billion points for the whole dataset. Comparing all possible scan pairs of this dataset would lead to 10585 comparisons. But since it doesn't make sense to compare scans that do not overlap in their observed volume we used a heuristic to discard all scan pairs that do not share a sufficiently large observed volume. Our heuristic uses the voxel datastructure that was already generated by the "peopleremover" to find those scans pairs. This heuristic under-approximates because ideally we are not only interested in the scans that measure points in a shared volume but also in the scan pairs where the free volume observed by one scan intersects with the measured points by the other. But even with this conservative heuristic, there exist 8372 scan pairs (79% of all possible scan pairs) in this dataset that share at least one 10 cm voxel with each other. This is explained by the large open spaces in the dataset. To further reduce the number of scan pairs that we choose for comparison with the algorithm by Underwood et al. we also discard all pairs that share less than 1000 voxel with each other. This leaves 3456 scan pairs to compare. Since the "campus" dataset does not contain any labels of dynamic objects, we reused the parameters that worked best for the "lecturehall" dataset. The results shown for the "campus" dataset in Table 2 indicate, that the algorithm by Underwood et al. performs an order of magnitude slower in this task compared to our solution. The number of compared scan pairs could be further reduced but for the purpose of "scan cleaning", the fewer comparisons are made, the more false negatives will be introduced in situations where a volume is seen as occupied by most scans and only seen as free by a few.

Our approach allows trading solution quality for runtime. For example, if the "lecturehall" dataset were processed with a voxel size of 17.5 cm instead of 10 cm as shown in Table 2, then the F_1-score would only slightly decrease from 0.96 to 0.95 but computation time would be cut by 18% down to 567 s.

The voxel traversal algorithm is very well suited for multithreading. Not only the voxel traversal can be run concurrently but also other parts of the execution pipeline can be run concurrently. While it is possible to introduce even more parallelism, our current implementation is able to handle scans in parallel for computing the maximum traversal ranges through the occupancy grid as well as during the voxel traversal phase. We didn't introduce parallelism in the other parts as they only require very little runtime in practice (filling the occupancy grid, clustering and sub-voxel accuracy) or are heavily I/O bound (loading input from files and storing the results).

As our benchmark system has eight physical cores we tested with one to eight threads in parallel and recorded the runtimes of each part of the algorithm. We used the first eight scans of the "Bremen city" dataset as input, with a voxel size of 10, a minimum cluster size of 40 and with subvoxel accuracy enabled.

The results are shown in Fig. 27. The phases of the algorithm for which runtimes have separately been timed coincide with the enumeration from Section 3. The runtimes for "maxranges" and "voxel traversal" do not scale completely linearly with the number of threads because of overhead in critical sections when the results are joined and because different scans take a different amount of time, leading to situations where only one CPU is still active near the end of each phase. Using datastructures that minimize the time spent in critical sections as well as adding parallelization to other parts of the algorithm is future work. The runtimes of "clustering" and "sub-voxel" are not visible in the barchart as each of them takes less than 3 s on the given dataset.

Since it's the main variable of our algorithm, we show the dependency of the runtime on the chosen voxel size. We used the first scan of the "Bremen city" dataset and executed our algorithm on it with varying voxel size, a minimum cluster size of 40 and with subvoxel accuracy enabled. Since only a single scan was processed, only one thread was used. The results are shown in Fig. 28. As expected, a larger voxel size results in faster execution as less voxels have to be traversed. The only part of the algorithm with a runtime dependent on the voxel size is the voxel traversal itself. It can be seen how the runtime scales inverse proportional to the voxel size.

Lastly, we also investigated whether our approach can be leveraged for achieving better point cloud registration results. Our idea was, that dynamic objects may have a negative impact on how well two scans can be matched. To evaluate our hypothesis, we used the Würzburg city dataset as it contained the highest number of dynamic points (2.65% of all occupied voxels are marked dynamic). We executed our algorithm with a voxel size of 10, a minimum cluster size of 40 and with subvoxel accuracy. We then registered the resulting cleaned scans again using slam6D from 3DTK using the same parameters as we used for the initial registration of the dataset. The results we achieved indicate no significant change in the scan registration. The differences in translation were not larger than 0.01 mm in any direction and the differences in angular orientation generally below 0.001° but never larger than 0.05°.

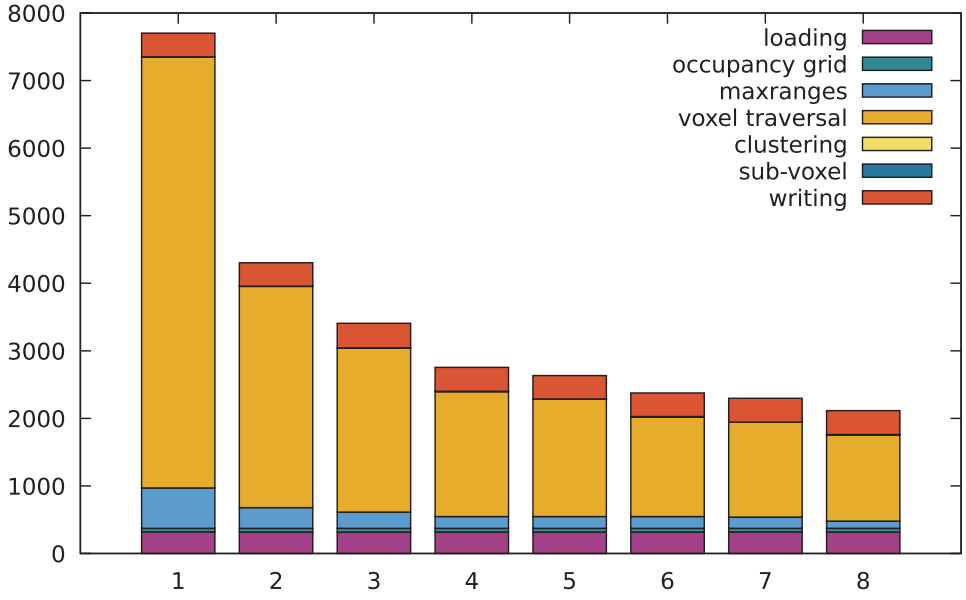


Fig. 27. Runtime of our algorithm in seconds (y-axis) depending on the used number of threads (x-axis).

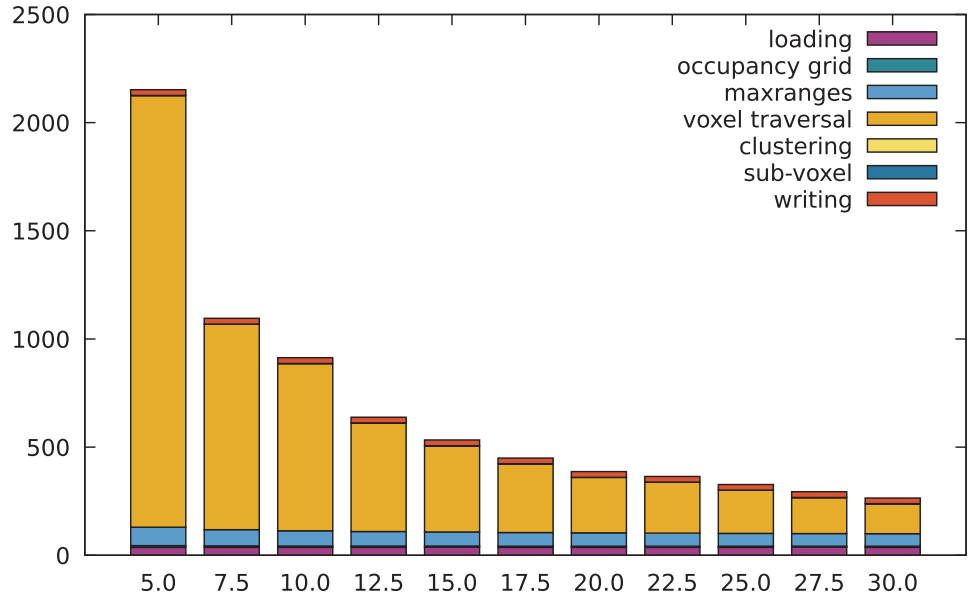


Fig. 28. Runtime of our algorithm in seconds (y-axis) depending on the voxel size (x-axis).

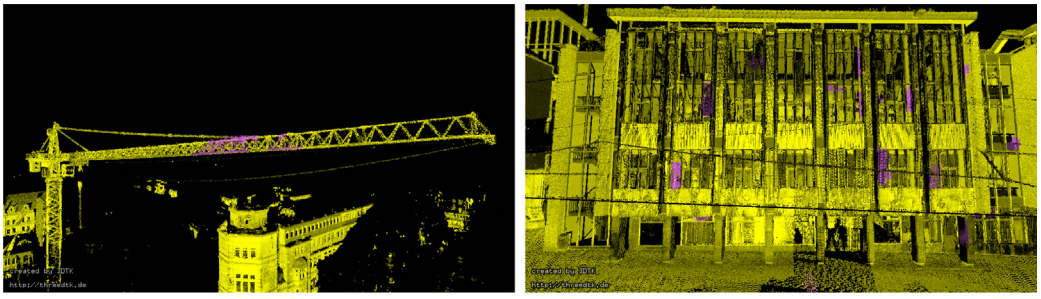
(a) (b)

Fig. 29. Slight registration errors at the church towers lead to incorrectly aligned surfaces. The outer surface is thus marked as dynamic. **Left:** Towers of Bremen Cathedral (St. Petri Dom zu Bremen) **Right:** Tower of Church of Our Lady (Kirche Unser Lieben Frauen).

10. Limitations

Our approach suffers from some limitations. False positives are introduced in the following situations:

- Incorrectly computed normal vectors lead to wrong shadowing information and thus to some lines of sight traversed through the voxel grid longer than they should've been traversed, resulting in voxels marked as see-through which are actually not. This situation easily occurs in either very noisy scans or for parts of a point cloud that doesn't have any clear normal vector like fences, wires, meshes and foliage. An example is shown in Fig. 30a.
- Solid objects that the laser beam can pass through due to their optical properties like glass will be marked as dynamic because they are seen as see-through. An example is shown in Fig. 30b.
- Surfaces with high reflectivity will result in points being seen in places "behind" the mirror and thus result in voxels being marked as see-through that lie in a direct line of sight between them and the sensor. An example is shown in Fig. 31. Work as in Koch et al. (2017) can mitigate this effect.



(a) (b)

Fig. 30. Examples for false positives **Left:** False positives due to wrongly computed normal vectors along the boom of a crane **Right:** False positives on a facade due to transparent windows.

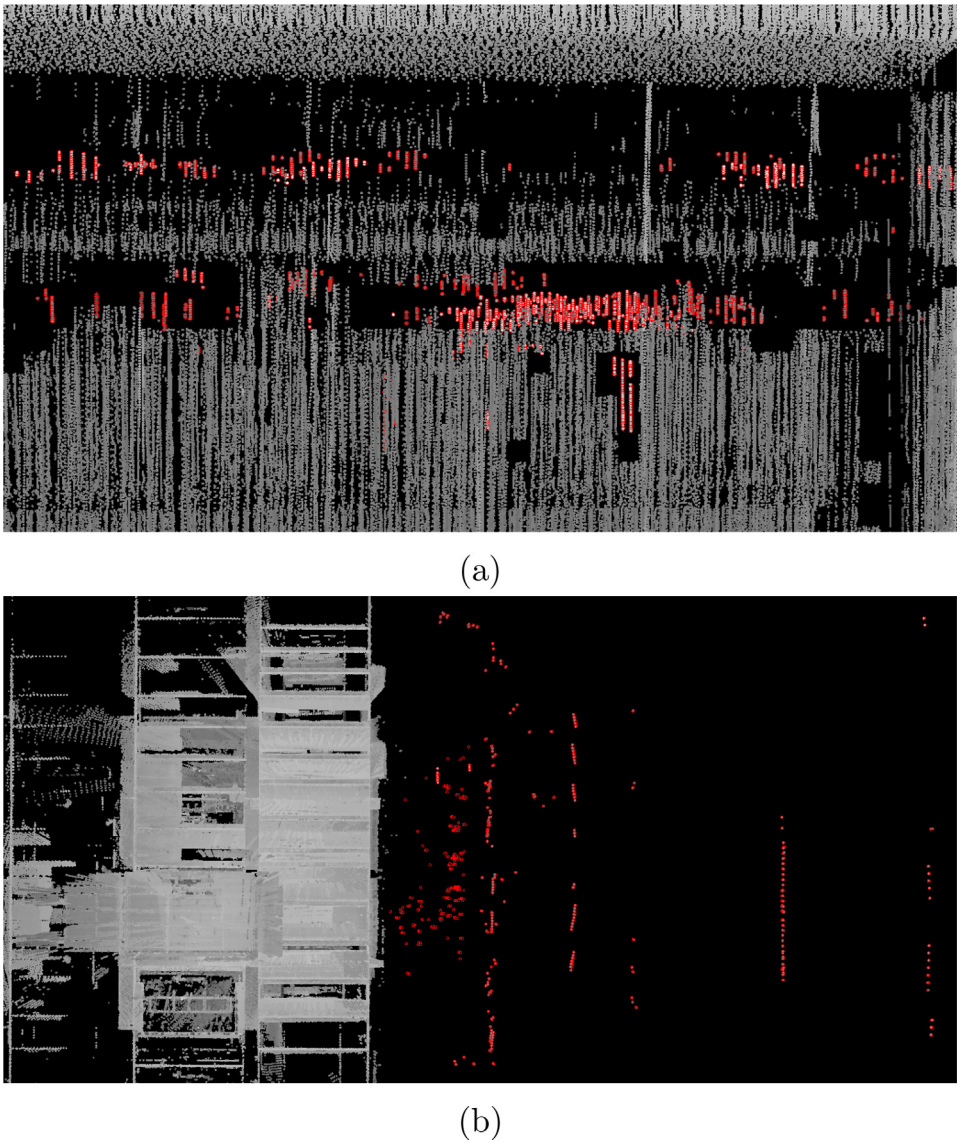


Fig. 31. The high reflectivity of surfaces commonly found in factory environments poses a great challenge. The wall in the top figure has holes because of reflected points "behind" the wall (in red). These points are false as the wall is solid and the area on the right in the bottom figure should be empty. **Top:** "Holes" in the wall created by points (in red) recorded behind the wall. The points are not exactly aligned with the holes due to parallax. **Bottom:** Top-view of the scene, showing the same points behind the wall (in red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- If our approach to subvoxel-accuracy is used, some false positives will be introduced as was shown in Section 8.
- If scans are not precisely aligned "double walls" or similar effects are created where there should only be a single wall. In these situations the wall in front of the other will wrongly be marked as "see through". An example is shown in Fig. 29.

In turn, false negatives occur during the following circumstances:

- At the boundaries between static and dynamic parts of the scan, some artifacts will be left depending on the voxel size and alignment. This effect can be reduced using our algorithm for subvoxel-accuracy which was explained in Section 8.
- Incorrectly computed normal vectors leading to wrong shadowing information can result in a traversal distance toward a point being cut off too early and thus miss traversing voxels that should be seen as free.
- If the line of sight from a second scan never intersects with a voxel that the former scan measured, then that voxel will never be marked as dynamic. This situation occurs through occlusion by otherwise dynamic points, by the scanner placements or in volumes where the point density is very low, as it typically is the case the further objects are away from the sensor.

In summary, apart from these properties, the quality of our results has similar limitations as competing methods and is highest in situations where the measurement noise is low, scans can be correctly registered and there are no transparent or reflecting objects in the scene.

11. Future work

So far we use the C++ standard library functionality like `std::set` and `std::unorderded_map` to build the voxel grid. We started with this simple approach to be able to show that good results can be achieved even when discretizing the input data with a regular occupancy grid. The bottleneck of our algorithm is the walk through the voxel grid and most time during its traversal is spent looking up grid cell information from the occupancy grid. Using an

`std::unordered_map` already gives us better runtimes than competing approaches but we assume that we can further increase performance by using datastructures which are directly designed for fast traversal through an occupancy grid. Examples for such implementations are Octomap (Hornung et al., 2013) as well as our own Octree implementation (Elseberg et al., 2013) which both offer an octree data structures which implement ray tracing capabilities. Another approach would be to replace the voxel grid by a sparse voxel DAG (Kämpe et al., 2013) which is specifically optimized to facilitate fast ray tracing through it.

Since the only data structure that must remain in memory is the occupancy grid, and since the memory requirement of that grid is several orders of magnitude less than the raw point cloud data, especially when using techniques like sparse voxel DAGs, it becomes feasible to process point clouds which would otherwise not fit into memory by loading scans on demand. For each step of the algorithm, only the points of a single scan are required and thus it becomes possible to immediately remove data from memory after it has been processed.

Lastly, many laser scanners are able to return more than one echo. Typically, structures that result in multiple laser echos are edges, fences, power lines or vegetation (Elseberg et al., 2011). These are also all the structures which typically do not provide good normal vectors. Information about multiple echos could be used to make our algorithm more robust against situations in which normal vectors cannot be computed.

12. Conclusion

We presented an approach specifically tailored to remove dynamic portions of 3D point cloud data. Our solution is suitable for scan slices from mobile mapping as well as for terrestrial scan data. We show experimental evidence that our approach compares favourably in quality to an existing solution for scan pairs. In terms of runtime our method is superior as it compares arbitrarily many scans with linear complexity.

Appendix A

Listing 1 and Listing 2.

```
ssize_t div(double a, double b)
{
        ssize_t q = a/b;
        double r = fmod(a,b);
        if ((r != 0) && ((r < 0) != (b < 0))) {
                q -= 1;
        }
        return q;
}
```

Listing 1. Integer division in C/C++ rounding to the next smallest integer.

```
double mod(double a, double b)
{
        double r = fmod(a, b);
        if (r!=0 && ((r<0) != (b<0))) {
                r += b;
        }
        return r;
}
```

Listing 2. Floating point modulo operation in C/C++.

References

Amanatides, J., Woo, A., et al., 1987. A fast voxel traversal algorithm for ray tracing. In: Eurographics, vol. 87. pp. 3–10.

Andreasson, H., Magnusson, M., Lilienthal, A., 2007. Has somethong changed here? Autonomous difference detection for security patrol robots. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2007. IROS 2007. IEEE, pp. 3429–3435.

Asvadi, A., Peixoto, P., Nunes, U., 2016a. Two-stage static/dynamic environment modeling using voxel representation. In: Robot 2015: Second Iberian Robotics Conference. Springer, pp. 465–476.

Asvadi, A., Premebida, C., Peixoto, P., Nunes, U., 2016b. 3d lidar-based static and moving obstacle detection in driving environments: an approach based on voxels and multi-region ground planes. Robot. Auton. Syst. 83, 299–311.

Blanco-Claraco, J., 2014. Mobile robot programming toolkit (MRPT). URL < https://www.mrpt.org > .

Budavári, T., Szalay, A.S., Fekete, G., 2010. Searchable sky coverage of astronomical

observations: footprints and exposures. Publ. Astron. Soc. Pac. 122 (897), 1375.
Drews-Jr, P., Núñez, P., Rocha, R.P., Campos, M., Dias, J., 2013. Novelty detection and segmentation based on gaussian mixture models: a case study in 3d robotic laser mapping. Robot. Auton. Syst. 61 (12), 1696–1709.
Elseberg, J., Borrmann, D., Nüchter, A., 2011. Full wave analysis in 3d laser scans for vegetation detection in urban environments. In: 2011 XXIII International Symposium on Information, Communication and Automation Technologies (ICAT). IEEE, pp. 1–7.
Elseberg, J., Borrmann, D., Nüchter, A., 2013. One billion points in the cloud–an octree for efficient processing of 3d laser scans. ISPRS J. Photogramm. Remote Sens. 76, 76–88.
Fekete, G., 1990. Rendering and managing spherical data with sphere quadtrees. In: Proceedings of the 1st Conference on Visualization'90. IEEE Computer Society Press, pp. 176–186.
Gálai, B., Benedek, C., 2017. Change detection in urban streets by a real time lidar scanner and MLS reference data. In: International Conference Image Analysis and Recognition. Springer, pp. 210–220.
Goodchild, M.F., Shiren, Y., 1992. A hierarchical spatial data structure for global geographic information systems. CVGIP: Graph. Models Image Process. 54 (1), 31–44.
Herbert, M., Caillas, C., Krotkov, E., Kweon, I.S., Kanade, T., 1989. Terrain mapping for a roving planetary explorer. In: Proceedings. 1989 IEEE International Conference on Robotics and Automation, 1989. IEEE, pp. 997–1002.
Hermann, A., Drews, F., Bauer, J., Klemm, S., Roennau, A., Dillmann, R., 2014. Unified GPU voxel collision detection for mobile manipulation planning. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, pp. 4154–4160.
Hornung, A., Wurm, K.M., Bennewitz, M., Stachniss, C., Burgard, W., 2013. Octomap: an efficient probabilistic 3d mapping framework based on octrees. Auton. Robots 34 (3), 189–206.
Kämpe, V., Sintorn, E., Assarsson, U., 2013. High resolution sparse voxel dags. ACM Trans. Graph. (TOG) 32 (4), 101.
Koch, R., May, S., Murmann, P., Nüchter, A., 2017. Identification of transparent and specular reflective material in laser scans to discriminate affected measurements for faultless robotic slam. Robot. Auton. Syst. 87, 296–312.
Kruger, J., Westermann, R., 2003. Acceleration techniques for GPU-based volume rendering. In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03). IEEE Computer Society, pp. 38.
Liu, K., Boehm, J., Alis, C., 2016. Change detection of mobile lidar data using cloud computing. In: International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences-ISPRS Archives, vol. 41. International Society of Photogrammetry and Remote Sensing (ISPRS), pp. 309–313.
Núñez, P., Drews, P., Bandera, A., Rocha, R., Campos, M., Dias, J., 2010. Change detection in 3d environments based on gaussian mixture model and robust structural matching for autonomous robotic applications. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, pp. 2633–2638.
Pfaff, P., Triebel, R., Burgard, W., 2007. An efficient extension to elevation maps for outdoor terrain mapping and loop closing. Int. J. Robot. Res. 26 (2), 217–230.
Qin, R., Tian, J., Reinartz, P., 2016. 3d change detection–approaches and applications. ISPRS J. Photogramm. Remote Sens. 122, 41–56.
Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., Strasser, W., 2003. Smart hardware-accelerated volume rendering. In: VisSym, vol. 3. Citeseer, pp. 231–238.
Ruixu Liu, V.K.A., 2017. 3d indoor scene reconstruction and change detection for robotic sensing and navigation. URL < https://doi.org/10.1117/12.2262831 > .
Rusu, R.B., Cousins, S., 2011. 3d is here: point cloud library (pcl). In: 2011 IEEE International Conference on Robotics and Automation (ICRA). IEEE, pp. 1–4.
Schauer, J., Nüchter, A., 2017. Digitizing automotive production lines without interrupting assembly operations through an automatic voxel-based removal of moving objects. In: 2017 13th IEEE International Conference on Control & Automation (ICCA). IEEE, pp. 701–706.
Szalay, A.S., Gray, J., Fekete, G., Kunszt, P.Z., Kukol, P., Thakar, A., 2007. Indexing the sphere with the hierarchical triangular mesh. Available from: < cs/0701164 > .
Triebel, R., Pfaff, P., Burgard, W., 2006. Multi-level surface maps for outdoor terrain mapping and loop closing. In: 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, pp. 2276–2282.
Turk, M.J., Smith, B.D., Oishi, J.S., Skory, S., Skillman, S.W., Abel, T., Norman, M.L., 2011. yt: a multi-code analysis toolkit for astrophysical simulation data. Astrophys. J. Suppl. Ser. 192, 9.
Underwood, J.P., Gillsjö, D., Bailey, T., Vlaskine, V., 2013. Explicit 3d change detection using ray-tracing in spherical coordinates. In: 2013 IEEE International Conference on Robotics and Automation (ICRA). IEEE, pp. 4735–4741.
Vieira, A.W., Drews, P.L., Campos, M.F., 2014. Spatial density patterns for efficient change detection in 3d environment for autonomous surveillance robots. IEEE Trans. Autom. Sci. Eng. 11 (3), 766–774.
Weinlich, A., Keck, B., Scherl, H., Kowarschik, M., Hornegger, J., 2008. Comparison of high-speed ray casting on GPU using CUDA and OpenGL. In: Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing, vol. 1. pp. 25–30.
Xiao, W., Vallet, B., Brédif, M., Paparoditis, N., 2015. Street environment change detection from mobile laser scanning point clouds. ISPRS J. Photogramm. Remote Sens. 107, 38–49.
Xiao, W., Vallet, B., Paparoditis, N., 2013. Change detection in 3d point clouds acquired by a mobile mapping system. ISPRS Ann. Photogramm. Remote Sens. Spatial Informat. Sci. 1 (2), 331–336.