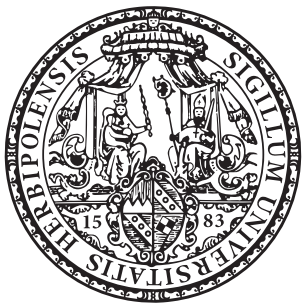


Julius-Maximilians-Universität Würzburg
Chair of Computer Science VII
Robotics and Telematics

Dissertation

Detecting Changes and Finding Collisions in 3D Point Clouds

Data Structures and Algorithms for Post-Processing Large Datasets

submitted to the Faculty of
Mathematics/Computer Science
of the University of Würzburg
in fulfillment of the requirements for the degree of
Doctor Rerum Naturalium (Dr. rer. nat.)
by

Johannes Schauer Marin Rodrigues

2020-05-22

Supervisor and first reviewer:	Prof. Dr. Andreas Nüchter
Second reviewer:	Prof. Dr. Alexander Reiterer

Contents

List of Tables

List of Figures

Abstract

Affordable prices for 3D laser range finders and mature software solutions for registering multiple point clouds in a common coordinate system paved the way for new areas of application for 3D point clouds. Nowadays we see 3D laser scanners being used not only by digital surveying experts but also by law enforcement officials, construction workers or archaeologists. Whether the purpose is digitizing factory production lines, preserving historic sites as digital heritage or recording environments for gaming or virtual reality applications – it is hard to imagine a scenario in which the final point cloud must also contain the points of "moving" objects like factory workers, pedestrians, cars or flocks of birds. For most post-processing tasks, moving objects are undesirable not least because moving objects will appear in scans multiple times (see Figure 1) or are distorted due to their motion relative to the scanner rotation (see Figure 2).

The main contributions of this work are two postprocessing steps for already registered 3D point clouds. The first method is a new change detection approach based on a voxel grid which allows partitioning the input points into static and dynamic points using explicit change detection (see Figure 3) and subsequently remove the latter for a "cleaned" point cloud (see Figure 4). The second method uses this cleaned point cloud as input for detecting collisions between points of the environment point cloud and a point cloud of a model that is moved through the scene.

Our approach on explicit change detection is compared to the state of the art using multiple datasets including the popular KITTI dataset. We show how our solution achieves similar or better F_1 scores than an existing solution while at the same time being faster.

To detect collisions we do not produce a mesh but approximate the raw point cloud data by spheres or cylindrical volumes. We show how our data structures allow efficient nearest neighbor queries that make our CPU-only approach comparable to a massively-parallel algorithm running on a GPU.

The utilized algorithms and data structures are discussed in detail. All our software is freely available for download under the terms of the GNU General Public license. Most of the datasets used in this thesis are freely available as well. We provide shell scripts that allow one to directly reproduce the quantitative results shown in this thesis for easy verification of our findings.

Figure 1: The same car multiple times from the KITTI dataset due to the same car being present in multiple scans at different positions in a mobile mapping scenario

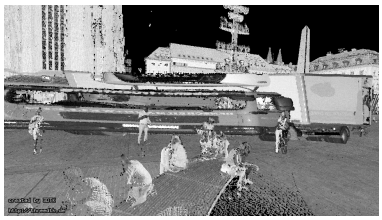


Figure 2: A "stretched" van from the Würzburg dataset due to the vehicle moving in the direction of the scanner rotation in a terrestrial mapping scenario

Figure 3: Non-static points are identified (magenta)...

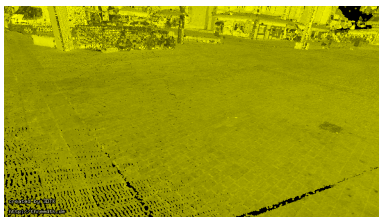


Figure 4: ...and removed without artifacts

> "Remember kids, the only difference between screwing around and science is writing it down."
>
> — Alexander Jason

1
Introduction

The work presented in this thesis started when we visited one of the leading automakers in Europe and were given a simple task: "We want to produce a new car model but we do not know whether the body will fit through our existing factory. Can you help us figure that out?"

We were surprised to learn that high tech companies like the one we visited do not possess digital models of their factories. Instead, production plants grow organically over the decades and figuring out whether a factory is compatible with a given new car model involves cardboard cutouts, styrofoam and lots of measurement tape. Needless to say, production needs to be halted while measurements are being taken manually by the workers, so money is lost during that downtime.

Automotive manufacturing lines proved to be a complex playground. While most of the time the car body moves along a straight line on a series of rails, it also gets turned upside down, rotated, gets put into elevator shafts and through narrow tunnels. Especially within turns or in the constricted environment of an elevator, a 2D stencil is not sufficient to adequately check whether the car body will indeed fit and whether there is enough safety distance around it.

Our solution was to create a three dimensional map of the production line and clearing it of moving objects like employees working along the production line. We then send a virtual model of the car body through that map along the same trajectory that the real car body would take and compute whether there are any collisions or whether safety margins are satisfied.

In this thesis we describe two important building blocks for this approach:

- removing dynamic objects from point clouds and
- computing collisions and penetration depths between two 3D point clouds.

Removing non-static (or dynamic) parts of a point cloud by itself has multiple applications:

- in indoor offices for intrusion detection or workspace planning,

- inside a factory or at industrial sites for industry 4.0 applications,
- at a mining site to monitor progress and watch for hazards,
- for city master planning and documentation purposes,
- at historical sites for archaeology and digital preservation purposes,
- and for gaming and virtual reality applications.

Likewise, computing collisions is important part

- of mobile robotics,
- planning of factory production lines,
- robotic manipulation tasks, as well as for
- logistical planning, like transportation of large goods along highways or railways.

All of this work is done with raw point clouds as input and without any surface approximation or model extraction or tracking of individual objects over time. Working with the raw point clouds as they are acquired by 3D laser range finders has the advantage that we do not take into account the inaccuracies of methods that turn the point cloud into a mesh or any other representation that tries to approximate surfaces or classify and fit objects. We show that using raw point clouds for dynamic object removal and collision detection can be done in a fast and reliable way.

1.1 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 and Woo[1]
- an approach that doesn't classify whole voxels as dynamic but only subsets of points in a voxel, achieving sub-voxel accuracy
- a spherical quadtree data structure for nearest neighbor search and point cloud reduction
- two collision detection methods (kd-CD and kd-CD-simple) to find collisions of a single arbitrary (and deformable) point cloud (the model) with a static environment
- two methods to calculate penetration depth of the model with the environment (kd-PD and kd-PD-fast)
- a highly optimized *k*-d tree implementation and query functions to perform collision detection

[1] Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10

All contributions of this thesis are released as part of *3DTK – The 3D Toolkit*[2]. 3DTK is released as free software under the terms of the GNU General Public License and can be downloaded either from sourceforge.net[3] or from a git mirror on github.com[4]. The tools build and run on Linux, Windows as well as on MacOS.

[2] `http://threedtk.de`

[3] `https://sourceforge.net/projects/slam6d/`

[4] `https://github.com/3DTK/3DTK`

Together with this thesis we release two shell scripts that can be run on a GNU/Linux machine to fully reproduce the F_1 scores we present in the quantitative results in section 4.10.1. The first script was already released together with our publication in the ISPRS Journal of Photogrammetry and Remote Sensing[5] while the other script extends upon the first with an evaluation of the KITTI dataset. The scripts can be retrieved via the following links, respectively:

[5] Schauer, J. and Nüchter, A. (2018a). Removing non-static objects from 3d laser scan data. *ISPRS Journal of Photogrammetry and Remote Sensing (JPRS)*, 143:15–38

- For datasets sim, lab, carpark, lecturehall: `https://robotik.informatik.uni-wuerzburg.de/telematics/download/isprs2018/`
- For KITTI datasets: `https://robotik.informatik.uni-wuerzburg.de/telematics/download/kitti2020/`

The scripts will download and compile our software as well as a competing solution, download the necessary datasets and finally run both approaches on each dataset, producing the F_1 scores found in the tables later in this thesis. Our datasets are published in the Robotic 3D Scan Repository at `http://kos.informatik.uni-osnabrueck.de/3Dscans/`.

1.2 *About this book*

This thesis was prepared using the LaTeX Tufte style. The Tufte style is named after Edward Tufte and used in his books[6]. The style is known for its extensive use of sidenotes and tight integration of graphics with text. Usually, looking up a citation would require to find the citation reference in the bibliography at the end of the work and finding figures and footnotes would require moving the eye away from the current paragraph of text to the top or the bottom of the page, respectively. We chose this style so that the reader is able to read through the main text of this thesis while at the same time be able to inspect the accompanying information without loosing the current visual focus. Since figures will usually be found directly next to the text referencing it, they will neither break the text flow nor will the reader get distracted by having to search for the right figure number.

[6] Tufte, E. R. (2006). *Beautiful evidence*. Graphis Press

This work is heavily based on and extends on previous publications from the author of this thesis. Previous work on change detection was published in the ISPRS Journal of Photogrammetry and Remote Sensing[7] and a summary of the work appeared in the IEEE Robotics and Automation Letters[8]. We expand on the these two publications with more in-depth explanations of the utilized algorithms and data structures like the spherical quadtree, a more thorough evaluation of the proposed approach by also running it on the KITTI dataset, as well as with a method to reduce the runtime of the voxel traversal for explicit change detection by several orders of magnitude without a significant change in the final F_1 score. The first publication on the topic of change detection was in

[7] Schauer, J. and Nüchter, A. (2018a). Removing non-static objects from 3d laser scan data. *ISPRS Journal of Photogrammetry and Remote Sensing (JPRS)*, 143:15–38

[8] Schauer, J. and Nüchter, A. (2018b). The Peopleremover — Removing Dynamic Objects From 3-D Point Cloud Data by Traversing a Voxel Occupancy Grid. *IEEE Robotics and Automation Letters (RAL)*, 3(3):1679–1686

the proceedings of the 13th IEEE International Conference on Control and Automation 2017[9]. While later work focused on input from terrestrial scans, this publication evaluated the change detection method on data from mobile mapping in an automotive production line environment.

Our work on collision detection started with a publication from the proceedings of the Photogrammetric Computer Vision conference in 2014[10] and focused on a dataset of a train tunnel and a train wagon moving through it. That work was further refined for a longer publication in the journal of Advanced Engineering Informatics in 2015[11]. That work also expanded on the underlying *k*-d tree data structure as the backbone for efficient collision computations on 3D point cloud data. Finally, we compared our work on collision detection with an implementation that relied on massively parallel computations on the GPU in a collaboration with colleagues from the Institute of Mathematical Machines in Warsaw in a publication that appeared in the proceedings of the 4th IFAC Symposium on Telematics Applications in 2016[12].

Unless otherwise specified, all runtime results in this thesis were carried out on a desktop PC with an Intel Xeon e5-2630 v3 processor with 8 physical cores with 2.4 GHz each and 32 GB of RAM. The operating system was Debian 10 Buster with Linux kernel 4.19.0 and GCC 8.3.0.

Unless otherwise specified, a left-handed coordinate system is used throughout this thesis with the y-axis pointing upwards, the z-axis pointing foward and the x-axis pointing to the right.

1.3 Outline

The remainder of this work is split into five parts. In the part that follows we discuss general data structures and algorithms for working with 3D point clouds. Specifically we outline how our *k*-d tree works and what techniques are used to improve its performance. We use a sphere quadtree as a search tree which is used to search a 3D point cloud for nearest angular neighbors. Lastly, we explain what data structure we chose to bin 3D points into a voxel grid without using memory for unoccupied volumes. The following part then explains the datasets that were used throughout the remainder of the thesis. We acquired some datasets specifically with change detection and collision detection in mind while others were taken from existing research. The largest part of the thesis handles our novel change detection algorithm, its performance improvements and techniques for avoiding false positives and generate qualitatively and quantitatively satisfying results. The next part shows our approach to collision detection and penetration depth computation and compares its runtime to a massively parallel implementation running on a GPU. In the last part we draw our conclusions and give an outlook on future developments.

[9] Schauer, J. and Nüchter, A. (2017). Digitizing automotive production lines without interrupting assembly operations through an automatic voxel-based removal of moving objects. In *Control & Automation (ICCA), 2017 13th IEEE International Conference on*, pages 701–706. IEEE

[10] Schauer, J. and Nüchter, A. (2014). Efficient point cloud collision detection and analysis in a tunnel environment using kinematic laser scanning and kd tree search. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 40(3):289

[11] Schauer, J. and Nüchter, A. (2015). Collision detection between point clouds using an efficient *k*-d tree implementation. *Journal Advanced Engineering Informatics (JAdvEI)*, 29(3):440–458

[12] Schauer, J., Bedkowski, J., Majek, K., and Nüchter, A. (2016). Performance comparison between state-of-the-art point-cloud based collision detection approaches on the CPU and GPU. In *Proceedings of the 4th IFAC Symposium on Telematics Applications (TA '13)*, volume 49, pages 54–59, Porto Alegre, Brazil

2
Data structures for point cloud processing

In this section we describe the data structures that lay the foundation for the change detection and collision detection algorithms explained in later part of this thesis. Managing 3D point clouds is a challenging task due to the sheer number of points in common datasets which easily exceed several hundred million or billions of points[1]. Without the use of specialized data structures finding points sharing similar properties, easily becomes an $O(n^2)$ operation on a dataset with n points.

In the following sections we present our implementation of two different tree data structures: the k-d tree and the spherical quadtree as well as a voxel grid implementation as a spatial hashing data structure. In general, we use tree data structures to quickly find spatial neighbors and a voxel grid as a representation of an occupancy grid.

[1] see datasets El Teniente, Wolfsburg or campus in later sections for examples

2.1 k-d tree

2.1.1 Introduction

In this section our highly-optimized k-d tree implementation is presented. It is implemented in 3DTK[2] in C++. It currently implements multiple search functions, can be parameterized to be used with 3D point data of different precision and container type, allows one to present search results as pointers, array indices or as 3D coordinate data and allows parallel execution through OpenMP. Its correctness has been verified by a test suite which combines brute force implementations of the search functions (test all points for satisfaction of the search criterea) against the result of a search in the k-d tree.

[2] Nüchter, A., Elseberg, J., Schneider, P., and Paulus, D. (2010). Study of parameterizations for the rigid body transformations of the scan registration problem. *Computer Vision and Image Understanding*, 114(8):963 – 980

The general operation of the search functions will be presented by using the function `fixedRangeSearch` as an example. The function is implemented in the class `KDTreeIndexed`. It sets up the `KDParams` structure with the search parameters and then calls the recursive function `_FixedRangeSearch` (notice the leading underscore) implemented in `KDTreeImpl`. The function `_FixedRangeSearch` in turn implements the actual search operations.

For an overview, consider Figure 2.1. Boxes are in UML, relationships (arrows) are not. `KDTreeImpl` is templated by `KDtreeIndexed`

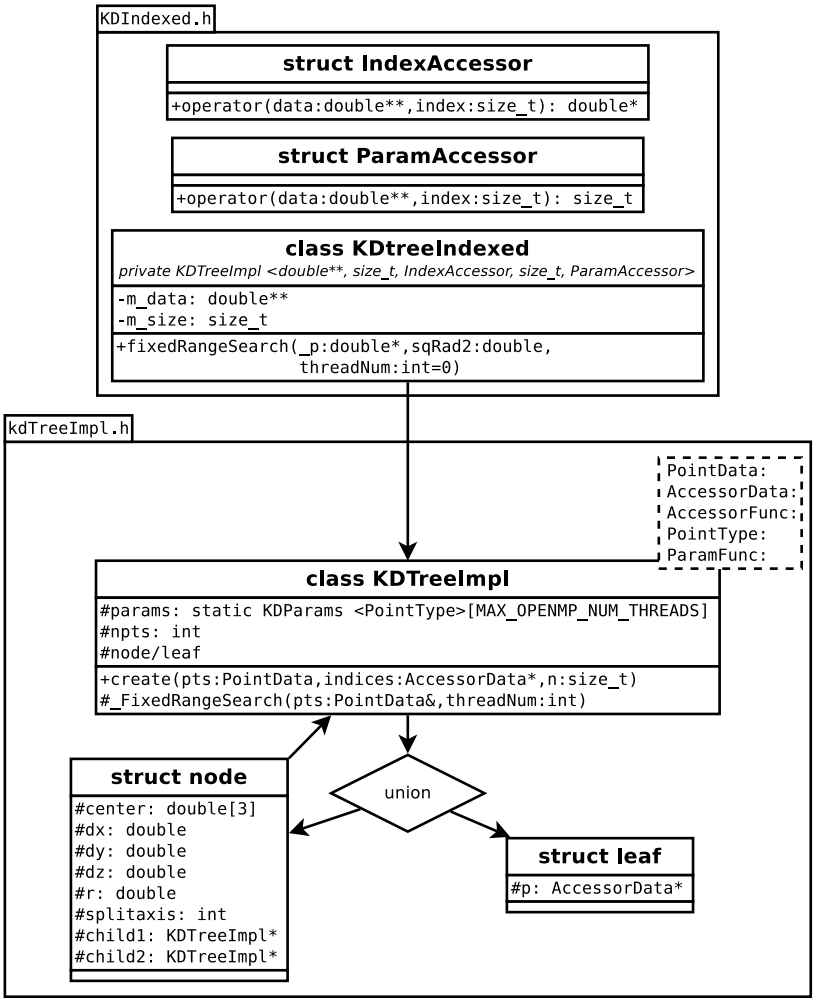


Figure 2.1: (mostly) UML diagram illustrating the relationship between different C++ classes in our k-d tree implementation

with the parameters listed in the comment. The `node` and `leaf` structure are a C++ `union`. The `params` member of `KDTreeImpl` is static. UML packages are used to indicate file membership and to group for readability. Only the `fixedRangeSearch` search function and its recursive counterpart `_FixedRangeSearch` are listed for brevity.

The template class `KDTreeImpl` provides the implementation of search functions and at the same time represents an inner node or a leaf node of the *k*-d tree. Multiple classes instantiate `KDTreeImpl`, one of them being `KDtreeIndexed` which is of particular use for the collision detection method in this thesis. The classes and functions seen in Figure 2.1 will be explained in more detail in the following sub-sections.

2.1.2 *Related work*

k-d trees[3] are a special kind of binary space partitioning trees. To construct a *k*-d tree, a set of points is recursively divided into two child nodes with their associated bounding boxes. In contrast to tree structures with a regular spatial partitioning like octrees, the repeated computation of bounding boxes adjusts the spatial search volumes to the underlying point cloud.

The *k*-d tree implementation in this work bears similarities to R+-trees[4] insofar it recalculates a new bounding box for each child node. In contrast to R+-trees, the *k*-d tree implementation presented here does not make efforts to create a balanced tree. In a publication by Elseberg et al.[5] our *k*-d tree implementation was benchmarked against three nearest-neighbor search libraries based on the *k*-d tree data structure: ANN[6], libnabo[7] and FLANN[8] and came out amongst the fastest implementations.

[3] short for *k*-dimensional tree

[4] Sellis, T., Roussopoulos, N., and Faloutsos, C. (1987). The r+-tree: A dynamic index for multi-dimensional objects

[5] Elseberg, J., Magnenat, S., Siegwart, R., and Nüchter, A. (2012). Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1):2–12

[6] Mount, D. M. and Arya, S. (2010). Ann: a library for approximate nearest neighbor searching, 2005

[7] Magnenat, S. (2014). libnabo

[8] Muja, M. and Lowe, D. G. (2012). Flann - fast library for approximate nearest neighbors

2.1.3 *Tree data structure*

Below listing shows an excerpt from the template class `KDtreeImpl`. Each instance of the class represents an inner or leaf node in the *k*-d tree.

```
template<class PointData, class AccessorData, class AccessorFunc,
class PointType, class ParamFunc> class KDtreeImpl {
  public:
    void create(PointData, AccessorData *, int);
  protected:
    static KDParams<PointType> params[MAX_OPENMP_NUM_THREADS];
    int npts; // equal zero for inner nodes, otherwise leaf
    union {
      struct {
        double center[3];
        double dx, dy, dz;
        int splitaxis;
        KDtree* child1, *child2;
```

```
14        } node;
15        struct { AccessorData* p; } leaf ;
16      };
17      void _FixedRangeSearch(const PointData&, int);
18  };
19  template<class T> class KDParams {
20    public:
21     double maxdist_d2;
22     double *p;
23     vector<T> range_neighbors;
24  }
```

The public `create` function in line 4 recursively creates a *k*-d tree by splitting the points it received as an argument into two, creating two new instances of `KDtreeImpl` and calling their `create` function with one of the new point sets, respectively. The inner working of the `create` function is explained in section 2.1.4.

The static member `params` in line 6 is set once for every new search in the *k*-d tree. It avoids having to pass the search parameters for each recursive function call and thus reduces the size of required operations on the stack. As it is a static member, it will only be stored in memory once, i.e., hardware cache friendly. The `KDParams` class in this shortened excerpt stores the point around which to search `p`, the squared search radius `maxdist_d2` and the search result vector `range_neighbors`. Since it is possible to carry out searches in the same *k*-d tree in parallel, an array of size `MAX_OPENMP_NUM_THREADS` exists.

The member `npts` in line 7 stores the number of points this node contains. If this value is non-zero, the node is a leaf node. Otherwise, the node is an inner node.

Depending on the node type, a union structure in line 8 stores data about the node. Inner nodes store their center coordinate (line 10), the node size (line 11), the coordinate axis by which the node is split (line 12) and pointers to the two children the node is split into (line 13). Leaf nodes store a pointer `p` to an array representing the contained points (line 15).

2.1.4 *Building the k-d tree*

A *k*-d tree is created by instantiating `KDtreeImpl` and calling its `create` method with the points one wants to fill the *k*-d tree with. The `create` method will then recursively instantiate new `KDtreeImpl` child nodes until all points are distributed into leaf nodes. The `create` method is shown as an abbreviated excerpt in the following listing and is explained in more detail further below.

```
1  KDtreeImpl::create(PointData pts, AccessorData *indices, int n) {
2    if (n > 0 && n <= 10) { // Leaf nodes, copy data
3      npts = n;
4      leaf.p = new AccessorData[n];
```

```
5       for (int i = 0; i < n; ++i) leaf.p[i] = indices[i];
6       return;
7     }
8     npts = 0; // inner node
9     // finding bounding box
10    // node.center, node.dx, node.dy, node.dz
11    [...]
12    // calculate longest axis
13    if (node.dx > node.dy)
14      if (node.dx > node.dz) node.splitaxis = 0;
15      else node.splitaxis = 2;
16    else
17      if (node.dy > node.dz) node.splitaxis = 1;
18      else node.splitaxis = 2;
19    // distributing data to fields left and right for the
20    // following nodes according to splitval
21    double splitval = node.center[node.splitaxis];
22    AccessorData *left;
23    [...]
24    // creation of subtrees
25    node.child1 = new KDtreeImpl();
26    node.child1.create(pts, indices, left-indices);
27    node.child2 = new KDtreeImpl();
28    node.child2.create(pts, left, n-(left-indices));
29  }
```

The first check in line 2 decides whether the current node is an intermediate node or a leaf node. If the number of points passed to the `create` function is less than or equal to 10 then this node will become a leaf node storing all points it is given and recursion stops. Otherwise the node is an inner node. This is recorded in the `npts` member in line 8. The number 10 is chosen as the bucket size because of run-time evaluations done by Nüchter et al[9] (see Figure 5 in that paper).

The clipped lines 9-11 calculate an axis aligned bounding box for the points the function is given. The bounding box is represented as its center point and its half length, width and height. Thus, the values `node.dx`, `node.dy` and `node.dz` store the distance from the center to the sides of the bounding box.

The axis by which to split the bounding box into two is found in lines 12-18. The split is done by determining the longest axis and splitting the bounding box in half by that axis. The choice of splitting along the longest axis over any other axis is made to create bounding boxes with volumes that more closely resemble a cube than thin slices. This is because during range search, less neighbor nodes will be intersecting a search radius if bounding, the higher the ratio between the bounding box volume and the sum of bounding box sides.

Instead of splitting the box by half its size, other choices are

[9] Nuchter, A., Surmann, H., Lingemann, K., Hertzberg, J., and Thrun, S. (2004). 6d slam with an application in autonomous mine mapping. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 2, pages 1998–2003. IEEE

possible. By investing more computational resources, the mean or the median coordinate values of the points inside the bounding box along the split axis can be computed. The median is the best choice for creating a balanced tree, as it will split each node such that the same number of points (plus-minus one) is contained in each child node. But computing the median is most computationally expensive as it requires to sort the coordinate values in $O(n \log n)$. Splitting by the mean can result in child nodes for which better-fitting bounding boxes can be computed because the node gets split by a value which halves the points by their spatial values. Tests with real-life data have shown that the theoretical advantages of more computationally expensive choices of the splitting axis show little benefit over just splitting the bounding box by half its size. An explanation is that re-computing the axis aligned bounding box is already sufficient for adjusting the tree geometry to the underlying data.

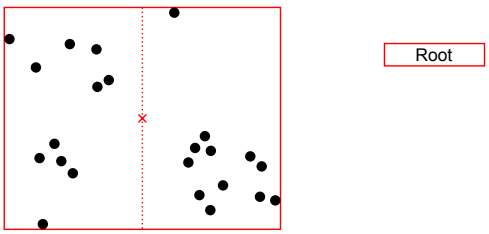
Lines 19-23 partition the points the `create` function is given. To reduce the amount of required copies, the original array with points is reused and split into half. Only points which happened to be on the wrong side are swapped with wrong points on the other side. On average this halves the amount of required copy operations. In the end, `indices` will point to the left hand side half of the original array while `left` will point to the right hand side half of the array. The last lines 24-28 instantiate two new `KDtreeImpl` objects and call their `create` function with the respective, sorted half of the original input data.

2.1.5 k-d tree layout

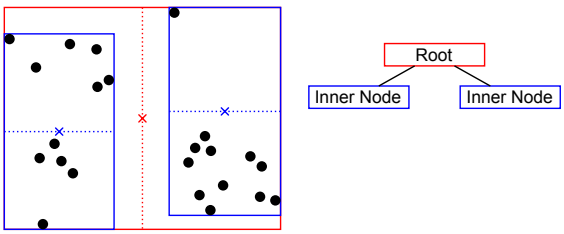
The `create` function explained in section 2.1.4 will result in a partitioning of the input points as shown in Figure 2.2 which shows a simplified two-dimensional representation of the input points and the resulting tree structure in memory.

In each sub-figure, 23 points are represented by black circles and the bounding boxes by solid colored lines. Crosses and dotted lines represent the bounding box centers and signify the split-axis of the 2-dimensional *k*-d tree. The letters identify the created groups of points per leaf node. The tree representation of the created 2D *k*-d tree can be seen on the right of each Figure. The color of the solid boxes corresponds to the bounding boxes in the left Figure. Boxes with dotted outlines are leaf nodes. The names of the leaf nodes correspond to the letters in the left Figure.

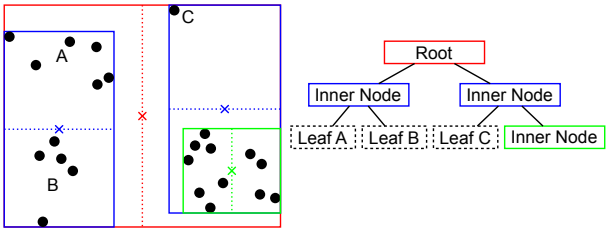
In contrast to a classical *k*-d tree, the search volume of child nodes is reduced by recalculating a bounding box for the enclosed points. This technique is similar to how R-trees operate and helps to create a tighter boundary for the enclosed points which in turn results in performance improvements during look-ups. This is because restricting the bounding volume of child nodes to a new bounding rectangle allows the algorithm to abort a search quickly



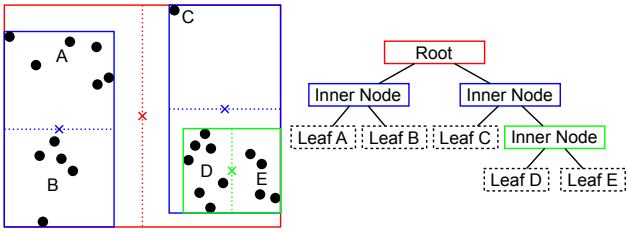
(a) Bounding box around all points (red)



(b) Two bounding boxes around the partitioned points (blue)



(c) One the right node has more than 10 points and gets divided again (green).



(d) Last division creating leaf nodes D and E.

Figure 2.2: Example of creating a 2D k-d tree

instead of having to search the k-d tree until leaf nodes are reached and inspected.

Considering Figure 2.2, the `create` function is first called with all 23 points as an argument. Since $23 > 10$, a new inner node will be created by calculating the node center and its bounding box (in red). The bounding box is wider than it is tall so that the points will be partitioned by a vertical axis across the bounding box center. Two new `KDtreeImpl` instances are created for each side and get passed 11 and 12 points, respectively. Since both values are greater than 10 again, new inner nodes will be created with their bounding boxes shown in blue. The following iteration will then result in two leaf nodes on the left hand side (6 points in the upper region and 5 points in the lower region) and one leaf node on the right hand side with one point. One last iteration over the remaining 11 points on the right hand side will create two last child nodes. Leaf nodes do not require a bounding box because when they are encountered during a k-d tree search, all the points they contain are checked and no further recursion has to be done.

2.1.6 *Searching the k-d tree*

Spacial searches in point clouds are parameterized by two properties: the location (where to search for results) and the subject (what to return). The following five search areas are implemented by 3DTK:

a. radius r around a point P_1
b. radius r around an infinite line defined by a point P_1 and a direction vector v
c. radius r around an infinite ray defined by P_1 and v
d. radius r along a finite line segment defined by points P_1 and P_2

and
e. inside an axis aligned bounding box defined by P_1 and P_2 as the corners with minimum and maximum coordinate values, respectively

Additional search volumes that can be added in the future would be oriented bounding boxes, cylinders or general polytopes. In most volumes, it is possible to perform searches for the following result types:

1. the point closest to P_1
2. the k points closest to P_1
3. all points within the search volume
4. the point closest to the given line, ray or line segment
5. the k closest points to the given line, ray or line segment

After eliminating the inapplicable combinations, one ends up with 19 meaningful search functions. A full list is omitted for brevity. For example, the common nearest neighbor search is searching for the closest point to P_1 (1) in a radius r around a point P_1 (a). For the collision detection method presented in this thesis, the following four functions are needed:

- `FindClosest`: closest point to a coordinate: (a) and (1)
- `fixedRangeSearch` all points around a coordinate: (a) and (3)
- `segmentSearch_1NearestPoint` closest point to P_1 in a line segment: (d) and (1)
- `segmentSearch_all` all points around a line segment: (d) and (3)

2.1.7 *_fixedRangeSearch*

All recursive search functions are divided into the following three functional parts. Firstly, the node is checked whether it is an inner node or a leaf node. If it is a leaf node, then all points the node contains are checked for satisfiability of the search criteria and the function returns. The second part is reached if the node is an inner node and thus the first part did not cause the function to return. In that case, a check is done whether the node can possibly contain parts of the result. If not, then the function returns. Otherwise, thirdly, the search recurses into one or both child nodes.

```
1  void KDtreeImpl::_FixedRangeSearch(const PointData& pts,
2  int threadNum) {
3    AccessorFunc point; ParamFunc pointparam;
4    if (npts) { // node is leaf
5      for (int i = 0; i < npts; i++) {
6        double myd2 = Dist2(params[threadNum].p,
7                            point(pts, leaf.p[i]));
8        if (myd2 < params[threadNum].maxdist_d2)
9          params[threadNum].range_neighbors.push_back(
10           pointparam(pts, leaf.p[i]));
```

```
11      }
12      return;
13    }
14    // quick test whether subtree has to be searched
15    double approx_dist_bbox =
16      max(max(fabs(params[threadNum].p[0]-node.center[0])-node.dx,
17      fabs(params[threadNum].p[1]-node.center[1])-node.dy),
18      fabs(params[threadNum].p[2]-node.center[2])-node.dz);
19    if (approx_dist_bbox >= 0 && sqr(approx_dist_bbox)
20       >= params[threadNum].maxdist_d2) return;
21    // recursive case
22    double myd = node.center[node.splitaxis]
23                 - params[threadNum].p[node.splitaxis];
24    if (myd >= 0.0f) {
25      node.child1->_FixedRangeSearch(pts, threadNum);
26      if (sqr(myd) < params[threadNum].maxdist_d2)
27        node.child2->_FixedRangeSearch(pts, threadNum);
28    } else {
29      node.child2->_FixedRangeSearch(pts, threadNum);
30      if (sqr(myd) < params[threadNum].maxdist_d2)
31        node.child1->_FixedRangeSearch(pts, threadNum);
32    }
33  }
```

Above listing shows the function `_FixedRangeSearch` as implemented in the `KDtreeImpl` class. It fills the result vector in the `KDParams` static member with all points in the k-d tree which lie around a certain squared radius `maxdist_d2` around a point `p`.

The parameterized functions of type `IndexAccessor` and `ParamAccessor` in line 3 are used to return coordinate data or data of the type stored in the results vector for each point in the leaf node, respectively. They do not pose a performance overhead as they are inlined by the compiler.

In case the node is found to be a leaf in line 4, all points in the leaf are checked whether their squared distance `myd2` to P is less than r.[10] If they do, then they are appended to the result vector.

After all points in the leaf node have been checked, the function returns. If the node is not a leaf node but an inner node, then the next part from line 15-20 checks whether further recursion into the child nodes of this node is required. The check whether to abort will be outlined in the next subsection 2.1.8.

The last part of each search function in lines 22-32 recurses into the child nodes. First, a check for the point's position relative to the split axis of the current node (as calculated in line 22) decides which child node to recurse first. Whether or not the other child node is recursed into as well depends on whether the bounding cube of the search radius around P can possibly extend into the other child as well or not.

[10] Throughout its codebase 3DTK stores and compares squared distances where possible, to avoid expensive computation of the square root. The `Dist2` function starts computing the euclidean distance between two points, but then does not compute the square root, resulting in the squared distance between two points.

2.1.8 Quick check whether to abort

A heuristic was developed that allows a quick check whether or not to continue searching further down the current branch of the k-d tree. Lines 15-18 in the last listing implement this check in C++. This code compiles to only 16 SSE2 instructions and requires no branching operations like a trivial check otherwise would.

The algorithm works by calculating a value d_P which is then compared to the search radius to decide whether or not to abort the search in the k-d tree. In the following formula, P is the three dimensional coordinate of the point around which the search is to be done. The current node of the k-d tree is parameterized by its center coordinate C and its axis aligned bounding box size $2d_x$, $2d_y$ and $2d_z$.

$$d_P = \max(|P_x - C_x| - d_x, |P_y - C_y| - d_y, |P_z - C_z| - d_z)$$

Suppose the six sides of the node's axis aligned bounding box form six axis aligned planes: each plane being the infinite extension of the six sides of the node's bounding box, respectively. Opposing sides of the node's bounding box form pairs of parallel planes. Three of these plane pairs are created, one pair along each dimension. Then the distance of P to the closest plane of each pair of planes is found. If P is between a pair of planes, then its distance is represented as a negative value. Then the maximum distance of the resulting three distance values is taken (one for each dimension). If the maximum value d_P is negative, then all three coordinate values of P must lie inside the current node's bounding box and the search has to recurse into one or both child nodes. If the maximum value is positive and larger than the search distance, then the current node cannot contain any results and the function returns without recursing deeper into the tree.

Figure 2.3 shows a two-dimensional overview of all possible locations a circular search radius (green) can have relative to the axis aligned bounding rectangle (yellow) of a two-dimensional k-d tree, ignoring rotations and mirroring. Each column represents a different horizontal position of the search radius relative to the bounding rectangle while each row represents a different vertical position. The lower-right triangle is faded out because it mirrors the upper left triangle along the diagonal. The black and red lines represent the positive and negative, respectively, distance from the search radius to the linear extension of the closest side of the bounding rectangle. The dark and light blue cells mark those positions in which parts of the search radius are found to lie in the bounding rectangle. In these cases, the search is not aborted as the search results might lie within the bounding box. In the other cases (cells with a white background) the search is aborted. The dark blue cell (b2) marks the case where this conclusion might lead to a false positive.

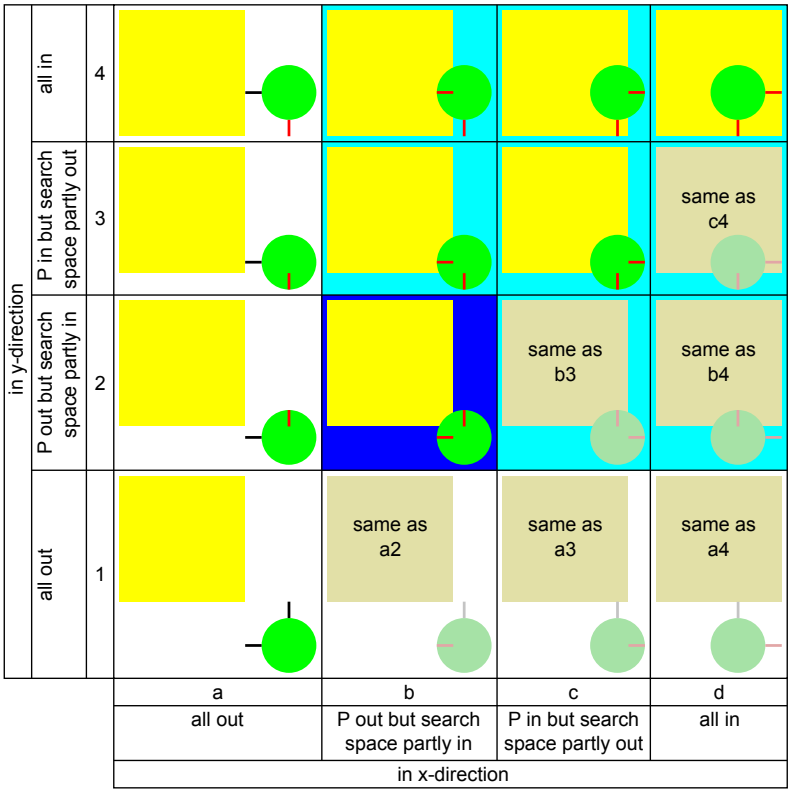


Figure 2.3: Two-dimensional overview of all possible locations a circular search radius (green) can have relative to the axis aligned bounding rectangle (yellow).

Figure 2.4 also visualizes the point where this check is not precise and generates a false positive. This is a close-up of cell b2 with a dark blue background in Figure 2.3). It shows the search radius (green) in a position which visualizes the false positive which will find the search radius to be intersecting with the axis aligned bounding rectangle (yellow) while there is no intersection in practice. Furthermore it shows the center of the bounding rectangle C, its size d_x and d_y, the center of the search radius P and its radius r as well as the linear extensions of the sides of the bounding rectangle X_1, X_2, Y_1 and Y_2. The distance e_x calculates as $|P_x - C_x| - d_x - r$. Since the result is negative, the line is colored in red. Similarly, e_y is calculated as $|P_y - C_y| - d_y - r$.

Since only the bounding cube of the search radius r around P is concerned, it can happen that both bounding cubes intersect while the actual search sphere does not intersect. In this case, the check will not abort the recursion even though no result can possibly be found in the current node in this situation. This inexactness is not a problem for values of r which are of similar order of magnitude as leaf node sizes in the search area. In that case, the overhead of searching for matching points in the few leaf nodes that are wrongly classified is far less than the overhead that is created by a more expensive but exact check which requires branching. A similar enhancement to sphere/box intersection checks by replacing branching with the max operator is shown in [11].

A slower but exact check would be of the form (in 2D):

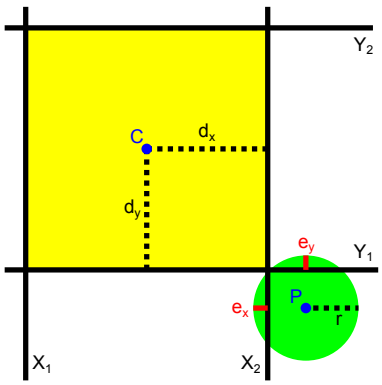


Figure 2.4: Close-up of cell b2 in Figure 2.3

[11] Larsson, T., Akenine-Möller, T., and Lengyel, E. (2007). On faster sphere-box overlap testing. *journal of graphics, gpu, and game tools*, 12(1):3–8

```
def doesCubeIntersectSphere(c1x, c2x, c1y, c2y, sx, sy, r):
    ds = sqr(r)
    if sx < c1x:
        ds -= sqr(sx - c1x)
    elif sx > c2x:
        ds -= sqr(sx - c2x)
    if sy < c1y:
        ds -= sqr(sy - c1y)
    elif sy > c2y:
        ds -= sqr(sy - c2y)
    return ds > 0
```

If the search radius r grows bigger, then it might be worth to add a second, more exact check after the quick inexact check. This is done for our k-d tree search functions around line segments. While inexact, checking whether parts of a node's bounding sphere intersect with the line segment's bounding sphere first, before doing an exact check, increased the runtime by two to three orders of magnitude. It is up to further research whether it is worthwhile to develop a more clever method which is able to decide for the best check to abort in each situation.

2.1.9 Subclassing the k-d tree

While the class `KDtreeImpl` contains the algorithms to build and search a k-d tree, it needs to be subclassed by a class that specifies the parameters of `KDtreeImpl`, provides a frontend for the search functions and which fills the parameter container `KDParams` with the correct values.

Parametrization of the `KDTreeImpl` class allows one to access coordinate data of different precision and container type through the `PointData` parameter. `AccessorData` allows different ways to access this data (through indices or pointers) while the `AccessorFunc` allows different ways of retrieving coordinate data with double precision from an array of `PointData` elements through an index given by the `AccessorData` type. The `PointType` parameter also governs how point data is stored in the shared parameter container `KDParams`. The `ParamAccessor` returns data of type `PointType` from the `PointData` type data array, given an index of type `AccessorData`.

This type of parameterization allows different use cases for the k-d tree. Originally, coordinate data was stored as pointers to three-tuple double arrays. This variant stores the data in the indices array, therefore using the identity function for `AccessorFunc` and `ParamFunc` and using `Void` as the `PointData` parameter. Later, support for the `DataXYZ` type was added which stores point data and attributes in a struct.

2.1.10 *An indexing k-d tree*

```
struct IndexAccessor {
  inline double *operator() (double** data, size_t index) {
    return data[index];
  }
};
struct ParamAccessor {
  inline size_t operator() (double** data, size_t index) {
    return index;
  }
};
class KDtreeIndexed : private KDTreeImpl<double**,
size_t, IndexAccessor, size_t, ParamAccessor> {
  public:  vector<size_t> fixedRangeSearch(double *, double);
  private: double **m_data;
}
```

For collision detection as explained in chapter 5, we make use of the indexing functionality of `KDtreeImpl`. Data and indices are passed to the *k*-d tree during creation and the search functions return individual indices or vectors of indices. This is useful to quickly calculate a partitioning of the points into colliding and non-colliding points without having to perform pointer arithmetic and relying on a certain layout of the point data in memory. Returning the indices of a range search allows one to quickly update boolean collision values in a second vector. As `IndexAccessor` and `ParamAccessor` are inlined by the compiler, they do not lead to a performance degradation.

```
KDtreeIndexed::KDtreeIndexed(double **pts, size_t n) {
  m_data = pts;
  create(pts, prepareTempIndices(n), n);
}
vector<size_t> KDtreeIndexed::FixedRangeSearch(double *p,
double maxdist2, int threadNum) {
  params[threadNum].maxdist_d2 = maxdist2;
  params[threadNum].p = p;
  _FixedRangeSearch(m_data, threadNum);
  vector<size_t> result;
  for (auto it : params[threadNum].range_neighbors) {
##pragma omp critical
    result.push_back(*it);
  }
  return result;
}
```

The constructor of `KDtreeIndexed` (line 1) simply creates the underlying *k*-d tree by supplying it with the given point values and an indexing array (line 3). The function `FixedRangeSearch` fills the `KDParams` structure with info about the desired point P and search

radius r in lines 7 and 8 and then calls the recursive search function that is implemented by `KDtreeImpl` in line 9. The search function saves its result in the `KDParams` structure, so they are copied to the final result vector in lines 10-14.

2.2 Sphere Quadtree

2.2.1 Introduction

As will be discussed in chapter 4, our voxel based change detection approach requires looking up all angular neighbors of a given query point to compute point shadows. Angular neighbors are those points which are seen within a maximum angular distance away from the initial query point. This means that even points that are far away in euclidean space can become angular neighbors. Or in other words: when all points of a single scan are projected on a unit sphere, then the angular neighbors of a given point are all points that lie on the sphere cap with a given angle and the query point at the apex of the cap.

A spherical quadtree is similar to a 3D octree. It produces a (nearly) regular partitioning of the input points and provides a data structure that can be efficiently traversed in $O(n \log n)$ to find angular range neighbors. In contrast to an octree, a spherical quadtree is two-dimensional and – as the name implies – is a tree where each node has four children and not eight. Similar to an octree, we use the spherical quadtree to search for neighbors and to perform point reduction – but both in terms of the points as projected on a unit sphere surface and not in euclidean space.

To create a spherical quadtree, 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 computations 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.

2.2.2 Related work

An alternative to using a tree data structure for angular neighbor lookup are binning approaches. One method that is also used for efficient point cloud reduction is to project all points into rectangular range images using common projections like mercator, conic, pannini or equirectangular. The downside of these projections is their behaviour at the poles where all pixels of the bottommost and topmost row are angular neighbors. It would be ideal, if it were possible to find a regular partitioning of a sphere surface with shapes of equal size and number of neighbors. Unfortunately the Euler characteristic of the sphere[12] makes it impossible to have

[12] The number of vertices V plus the number of faces F minus the number of edges E of polyhedra is equal to 2 for a sphere: $V - E + F = 2$

more than five such partitionings: the five platonic solids.

To partition a sphere surface into more than the 20 sides of an icosahedron, approaches use a cube, a tetrahedron, an octahedron or an icosahedron as a base and continue subdividing their sides in a regular manner. The triangular faces of a tetrahedron, octahedron and icosahedron are projected to the surface of the unit sphere, each of its three edges get halved and the three new vertices create four new triangles. Similarly, the faces of a cube are projected to the sphere surface and divided into four new faces with half the side-length each. Subdivisions of the pentagon faces of a dodecahedron are not known to the author.

Using the octahedron and icosahedron as a starting polyhedron seems to be the most popular choice in literature. The result of applying these recursive subdivisions of the unit sphere surface is also known as hierarchical triangular meshes[13] or sphere quadtrees[14]. That data structure has so far mostly been used for Geographic Information Systems to model features on top of the earth surface[15] or in astronomy to map objects in the sky[16]. 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 $O(\log n)$ or as means to perform point reduction in angular space. We create one sphere quadtree per scan with its center at the origin of the scanner-local coordinate system.

[13] Szalay, A. S., Gray, J., Fekete, G., Kunszt, P. Z., Kukol, P., and Thakar, A. (2007). Indexing the sphere with the hierarchical triangular mesh. *arXiv preprint cs/0701164*

[14] Fekete, G. (1990). Rendering and managing spherical data with sphere quadtrees. In *Proceedings of the 1st Conference on Visualization'90*, pages 176–186. IEEE Computer Society Press

[15] Goodchild, M. F. and Shiren, Y. (1992). A hierarchical spatial data structure for global geographic information systems. *CVGIP: Graphical Models and Image Processing*, 54(1):31–44

[16] Budavári, T., Szalay, A. S., and Fekete, G. (2010). Searchable sky coverage of astronomical observations: Footprints and exposures. *Publications of the Astronomical Society of the Pacific*, 122(897):1375

2.2.3 *Implementation*

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 quadtree to insert it. We achieve this by arranging the eight quadtrees in an array of length eight in an ordering such that the index of the quadtree 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 more costly geometric computations when a new point is inserted or when the data structure is queried for angular neighbors. Specifically, for an icosahedron, multiple triple product[17] computations have to be carried out to determine into which of the 20 sides of the icosahedron a given point is projected to.

A given triangle is subdivided into four new triangles by taking the center points of its three edges for three new vertices. These

[17] The triple product between three vectors a, b and c is computed as $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$. If a and b are triangle vertices, then the sign of the triple product can be used to determine whether a point c lies inside or outside the triangle.

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.

Since the same point on the unit sphere will be shared by multiple triangles, either adjacent or on multiple depths of the tree, surface coordinates are not stored directly in the tree nodes. Instead, two mappings are maintained by the algorithm in form of a hash table. One hash table maps vertex indices to normalized locations on the surface of the unit sphere. The other hash table maps pairs of indices to the index of the normalized average between these two vertices (their middle). The former mapping allows the algorithm to re-use vertex position and saves memory by only storing their index instead of the full vertex coordinate for every triangle. The latter mapping allows the algorithm to skip repeated computation of edge centers for adjacent triangles.

The initial mapping of vertex indices to normalized vertex coordinates is filled with the six vertices that make up an octahedron. To allow trivial binning of new points into the right sub-tree starting at each side of the octahedron, the octahedron is aligned with the coordinate axis with the following six vertices: (-1,0,0), (1,0,0), (0,-1,0), (0,1,0), (0,0,-1) and (0,0,1). Indices 0 to 5 of the vertex mapping will map to these initial vertices, respectively. Any further entry of this hash table is added by the function computing new middle points between two vertices. That function will retrieve the middle point from the second hash table, if present or compute a new middle point and add a new mapping to each of the hash tables.

The order of the initial six vertices in the last paragraph was chosen that way, because it allows for a simple computation of the vertices that make up the triangles on the eight sides of the octahedron:

```
std::vector<std::array<size_t, 3>> mainvertices;
for (int x : {-1, 1}) {
    for (int y : {-1, 1}) {
        for (int z : {-1, 1}) {
            size_t v1 = x < 0 ? 0 : 1;
            size_t v2 = y < 0 ? 2 : 3;
            size_t v3 = z < 0 ? 4 : 5;
            if (((x > 0) ^ (y > 0) ^ (z > 0)) == false) {
                std::swap(v1, v3);
            }
            mainvertices.push_back({v1, v2, v3});
        }
    }
}
```

As a result, the vector `mainvertices` will contain eight entries. Each entry is a three-tuple of indices 0 to 5. With the ordering of

the initial six vertices that was specified, each index will refer to the right vertex such that eight triangles of an octahedron are formed. The `std::swap` instruction makes sure that the vertices of the current triangle are given in the same order (clockwise). This is necessary to make the normal vector always point away from the center of the unit sphere into which the octahedron is inscribed.

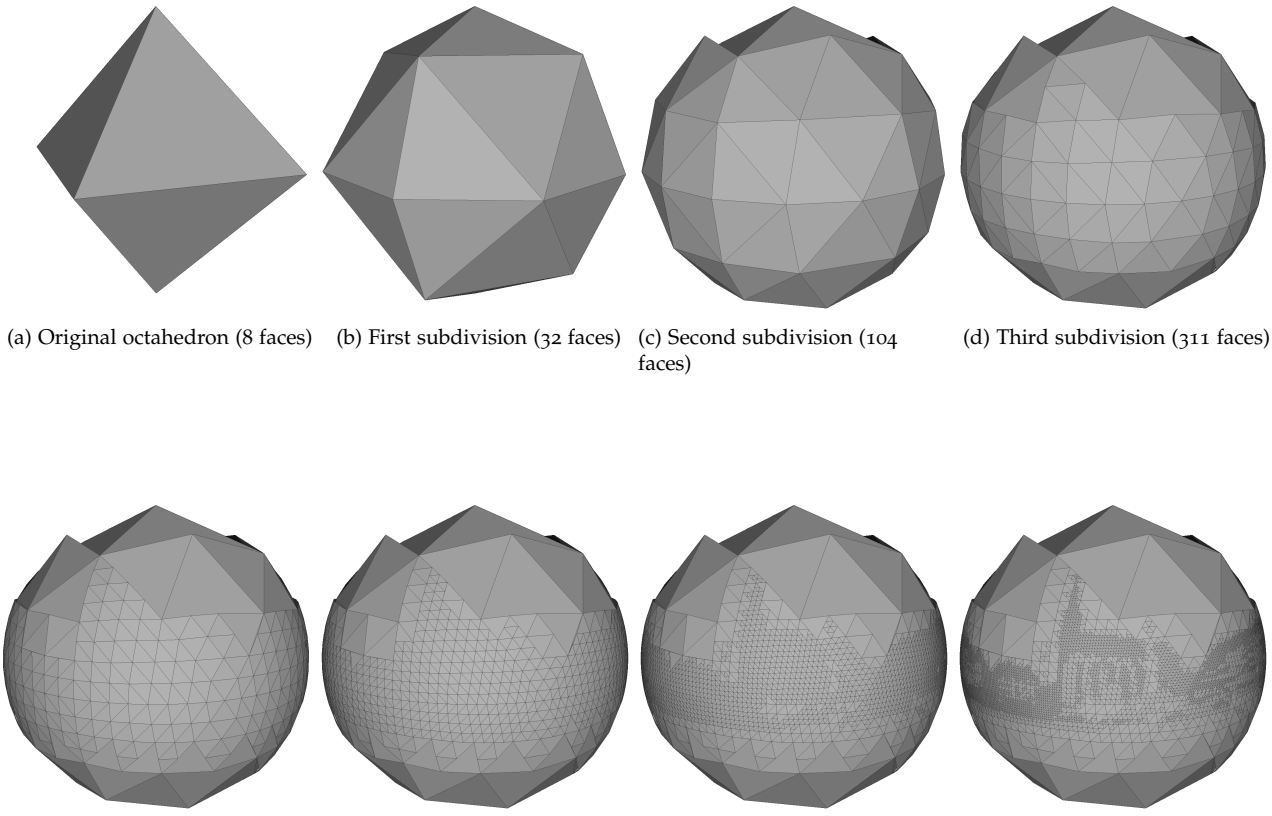
Each of the sides of the octahedron starts a new search tree. The search tree consists of nodes which are either intermediate nodes or leaf nodes. Creating a new node requires the three vertices that make up the new triangle as well as the list of points that fall into that triangle on the unit sphere. The vertices are only used to create new child nodes and sort the given points into each, accordingly. The vertices that were used to create a node are not permanently stored. Instead, each node stores the center as well as the radius of the circumcircle of the triangle formed by its three vertices. These two values can then later be used when searching the tree to efficiently decide whether a given point can possibly lie inside a node or not.

A node is made a leaf node without any further children if either less than a certain maximum number of points per node are to be sorted in it or if the circumcircle of the triangle is below a certain minimum threshold. The second criteria makes sure not to recurse infinitely in case the input data contains the exact same point more often than the maximum number of points per node.

If a node is not a leaf node, then four child nodes are created by computing (or re-using from the precomputed hash mapping) the three middle points of the three edges of the triangle formed by the three vertices that were given to the node. The three new vertices together with the three original vertices form four new triangles.

Naively, figuring out into which of the four new triangle any given point belongs would require up to twelve computations of the triple product between two vertices making up one of the new edges and the new point. By determining for which triangle the triple product is greater or equal to zero for all three sides, the right child node to sort the given point in can be found.

The number of necessary computations can be reduced to at most three computations of the triple product by exploiting the fact that the points that are to be sorted must already lie inside the original outer triangle that forms the current node. With that knowledge, only the triple product of the given point and the three new edges has to be checked. If the result is positive for the triple product with any of the three edges, then the point belongs to the triangle sharing that edge and one of the original outer vertices, respectively. If the result is not positive for any of the edges, then the point must lie in the new center triangle. 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) 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)

Figure 2.5: Example spherical quad tree using a scan of the Würzburg dataset.

A visualization of how the points from a terrestrial scan are inserted into the data structure is shown in Figures 2.5. Each subfigure shows one additional recursion step. 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.

Figure 2.6 shows a visualization of the nodes of a sphere quadtree from a terrestrial scan with 13769 faces. The reflectance values of that scan are mapped as a texture on a perfect sphere in the same orientation in Figure 2.7. The final data structure contains 580,000 points. Figure 2.7 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 Figure 2.6 allows one to clearly recognize the shape of the scanned buildings shown in Figure 2.7. 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 quadtrees. 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.

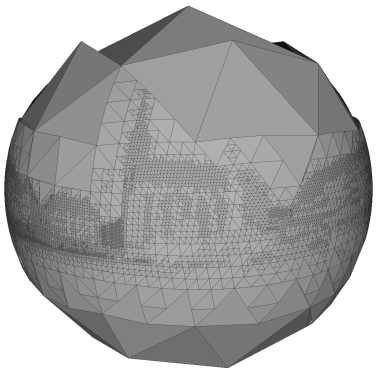


Figure 2.6: Final spherical quad tree of a scan from the Würzburg dataset

Figure 2.7: The scan from Figure 2.6 as reflectance image on a perfect sphere surface in the same orientation.

2.2.4 *Search tree*

Similar to the *k*-d tree, the spherical quad tree is used for range searches. While the *k*-d tree is used for nearest neighbor searches around a given coordinate in 3D, the spherical quad tree is used to find all neighbors within a certain angular radius around a given coordinate in 2D on a unit sphere surface. In section 2.1.8 we discussed how our *k*-d tree implementation uses a heuristic with only few false positives to perform quick intersection tests between the axis aligned bounding box around the node contents and the search radius. Performing an accurate intersection test between a non-euclidean triangle on a sphere surface and a search radius (a sphere cap) involves comparing the center of the search radius with the planes through the sphere center and all sides of the triangle, respectively, and then comparing their angular distances. To avoid the necessary computations to carry out this precise check, we approximate each spherical quad tree node by the circumcircle of the triangle from which the node was created. As a result, only the angular difference between the center of that circumsphere and the center of the search radius has to be checked against the sum of circumcircle radius and search radius. This heuristic also allows us to discard the triangle that was used to create a spherical quad tree

node because we will only use its circumcircle for angular range searches.

In case of the spherical quad tree, node boundaries are non-euclidean triangles on a sphere surface and the search volume is the area of a sphere cap. In the worst case, the triangle is "flat" or close to being flat, so either a degenerate triangle, an obtuse triangle with one very large angle or an acute triangle with one very short and two very long sides. The area of all of these triangles is small compared to the area of their circumcircles. The best case on a sphere surface is a triangle whose vertices lie exactly on a plane intersecting with the sphere center. In that case the triangle area is equal to the area of its circumcircle. In two dimensions, the best triangle is an equilateral triangle. The two dimensional case is important here, because as we subdivide triangles into smaller and smaller triangles, they also more and more approximate triangles on a plane in the same way as small distances on the earth's surface can be approximated using euclidean geometry instead of using great-circle distances.

In Figure 2.8 an octahedron was recursively subdivided, building a complete spherical quad tree up to a depth of 10, which results in $8 \cdot 4^{10} = 8388608$ child nodes. In each depth of the subdivision, the ratio between triangle area on the unit sphere surface and circumcircle area (or sphere cap area) was computed and the minimum and maximum values for this ratio can be seen in the Figure.

The values can be understood as follows. For a depth of zero, the triangles of the original octahedron are projected on the unit sphere surface. Thus, their area is equal to one eighth of the unit sphere ($r = 1.0$) surface area.

$$A_{triangle0} = \frac{4\pi r^2}{8} = \frac{1}{2}\pi \tag{2.1}$$

Its circumcircle area is the area of the sphere cap created by a plane cutting the unit sphere at all three vertices of the triangle. The sphere cap surface area can be computed from the sphere radius $r = 1.0$ and the angle θ between the the rays from the center of the sphere to the apex of the cap and the edge of the disk forming the base of the cap.

$$A_{cap0} = 2\pi r^2 (1 - \cos\theta) \tag{2.2}$$

The angle θ can be obtained by computing the circumradius of the equilateral triangles with side length $s = \sqrt{1^2 + 1^2} = \sqrt{2}$ that form the octahedron:

$$r_{tri} = \frac{s}{\sqrt{3}} = \sqrt{\frac{2}{3}} \tag{2.3}$$

The angle theta is the angle of the right triangle formed by the center of the circumcircle, one of the edges of the triangle and the center of the unit sphere. The angle θ is thus asin (r_{tri}), plugging

Figure 2.8: The minimum and maximum ratio between triangle area and its circumcircle area on a octahedron, subdivided up to a certain depth.

all of this into equation 2.2 with unit sphere radius $r = 1.0$ and by using the identity $\cos \operatorname{asin} x = \sqrt{1 - x^2}$ we get:

$$A_{cap0} = 2\pi \left(1 - \cos \operatorname{asin} \sqrt{\frac{2}{3}}\right) \tag{2.4}$$

$$= 2\pi \left(1 - \sqrt{1 - \frac{2}{3}}\right) \tag{2.5}$$

$$= 2\pi \left(1 - \frac{1}{\sqrt{3}}\right) \tag{2.6}$$

Finally, computing the ratio between $A_{triangle0}$ and A_{cap0} yields:

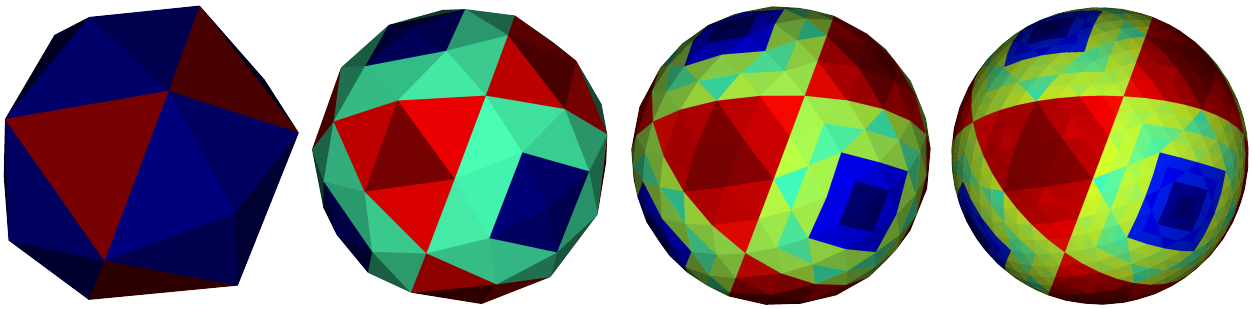
$$A_{ratio0} = \frac{\frac{1}{2}\pi}{2\pi \left(1 - \frac{1}{\sqrt{3}}\right)} \tag{2.7}$$

$$= \frac{1}{4 - \frac{4}{\sqrt{3}}} \tag{2.8}$$

$$= 0.5915063509461096\ldots \tag{2.9}$$

The value of A_{ratio0} is precisely the value at depth zero in Figure 2.8.

As depth increases, one can see that the minimum and maximum values seem to converge towords 0.318310 and 0.413497, respectively. These values can be explained by looking at a rendering of the subdivided unit sphere.



(a) depth 1 (32 faces) (b) depth 2 (128 faces) (c) depth 3 (512 faces) (d) depth 4 (2048 faces)

Figure 2.9: Subdivided octahedron in four different depths with colors indicating the ratio between each triangle area and its circumcircle area. The color scales are different because the respective minimum and maximum areas differ too much. Blue indicates a low ratio (bad) and red a high ratio (good).

Figure 2.9 shows the original octahedron subdivided up to depth 1, 2, 3 and 4. Each triangle is colored to indicate its ratio with its circumcircle area. Blue colors indicate low ratios and red high ratios. As one can see, the blue triangles center around the six vertices that created the original octahedron and approximate right triangles with two equal length sides. Red triangles surround the center of the sides of the original octahedron and approximate equilateral triangles. As triangles become smaller and smaller relative to

the sphere itself, euclidean geometry can be used to approximate their properties. The circumcircle of right triangles has its midpoint at the center of the hypotenuse. The ratio between its area (with two of its sides a and b being of equal length) and the area of its circumcircle (through its hypotenuse c) can be computed as:

$$\frac{A_{rectTri}}{A_{cirumRectTri}} = \frac{\frac{1}{2}ab}{\frac{1}{4}\pi c^2} \tag{2.10}$$

$$= \frac{\frac{1}{2}a^2}{\frac{1}{4}\pi\sqrt{a^2+b^2}^2} \tag{2.11}$$

$$= \frac{\frac{1}{2}a^2}{\frac{1}{2}\pi a^2} = \frac{1}{\pi} \tag{2.12}$$

$$= 0.3183098861837907\cdots \tag{2.13}$$

Similarly, for a equilateral triangle with sides of length a, the ratio computes as:

$$\frac{A_{eqTri}}{A_{cirumEqTri}} = \frac{\frac{\sqrt{3}}{4}a^2}{\pi\left(\frac{a}{\sqrt{3}}\right)^2} \tag{2.14}$$

$$= \frac{\sqrt{3}a^2}{4\pi\frac{a^2}{3}} \tag{2.15}$$

$$= \frac{3\sqrt{3}}{4\pi} \tag{2.16}$$

$$= 0.41349667156634407\cdots \tag{2.17}$$

The resulting values from equations 2.13 and 2.17 are the same values to which the minimum value and the maximum value in Figure 2.8 converge to. Since the triangle subdivision as proposed here produces four similar triangles in every step of the iteration, it can be concluded that the minimum and maximum ratios for the spherical quadtree subdivision will indeed remain those shown even for higher recursion depths.

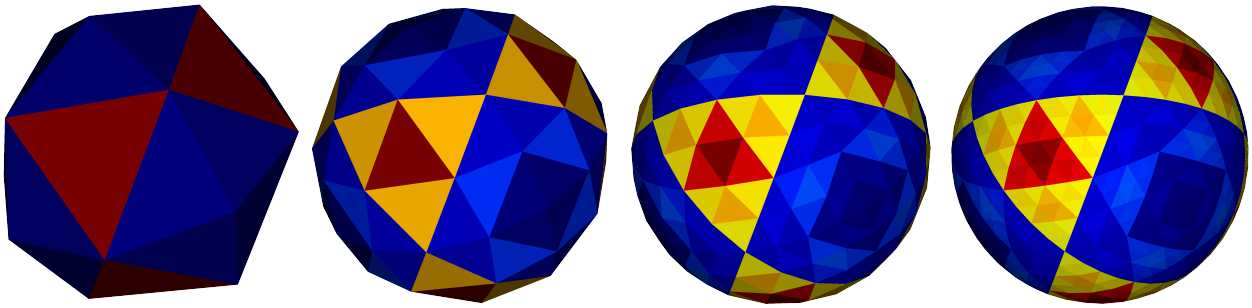
2.2.5 Point reduction

The spherical quad tree can also be used as a method for reducing the number of points in a point cloud. This operation has a similar effect as point cloud reduction using range images with the advantage, that points will end up evenly distributed across the unit sphere surface. Range image reduction on the other end suffers from differing point densities toward the poles depending on the projection method that was used to project the points to the range image rectangle. Point reduction using the spherical quad tree works similarly to point reduction using the octree. The parameters of octree reduction are the final octree node size and the points per node. The octree is then traversed up to the given threshold and points are randomly returned to fulfill the requested density criteria. Similarly, for point reduction using a spherical quadtree, the

parameters are the surface area of a spherical cap A_{cap} given by its angular diameter and the maximum number of points k that shall be on that surface area. The tree is then recursively traversed either down to the child nodes or to the node whose circumcircle area is smaller than the area of the requested sphere cap. The number of points n that the node actually will return are computed using the area covered by the current node on the unit sphere surface A_{tri}:

$$n = k\frac{A_{cap}}{A_{tri}} \tag{2.18}$$

The n points will then be randomly selected from N and returned. If N contains less than n points, all points in N are returned. The area A_{tri} could be computed on-the-fly from the circumcircle radius but is stored in each node to save computation time.



(a) depth 1 (32 faces) (b) depth 2 (128 faces) (c) depth 3 (512 faces) (d) depth 4 (2048 faces)

Figure 2.10: Subdivided octahedron in four different depths with colors indicating the triangle area. The color scales are different because the respective minimum and maximum areas differ too much. Blue indicates the minimum area and red the maximum area.

Figure 2.10 shows the differences in triangle area across the subdivided sphere for different subdivision depths. As explained in section 2.2.4, the triangles towards the vertices of the original octahedron approximate right triangles while those in the center of the original faces approximate equilateral triangles. Since the range of minimum and maximum area of triangles across the subdivided sphere differs greatly for each subsequent subdivision step, the color scales are different for each sub-figure in Figure 2.10.

Table 2.1: Minimum and maximum triangle areas for different recursion depths

Depth	#faces	minimum area	maximum area
1	32	0.861700	1.152986
2	128	0.222379	0.359341
3	512	0.056059	0.095932
4	2048	0.014044	0.024401

Depth	#faces	minimum area	maximum area
5	8192	0.003513	0.006127

As shown by table 2.1 the minimum and maximum area per triangle can be nearly twice each other's value. Nevertheless, the overall point density across the whole unit sphere surface will be equal after point reduction because the size of each triangle is taken into account when computing the number of points n that remain per node.

2.3 Voxel Grid

2.3.1 Introduction

Regular grid data structures like voxel grids possess several desirable properties in contrast to irregular space partitioning data structures such as k-d trees. Their regular nature allows efficient access of each voxel in $O(1)$ instead of having to search a tree data structure in $O(n \log(n))$. Similarly, inserting new data into an existing grid, can be achieved in $O(1)$ simply because adjacent grid cells do not have to be changed at all. Inserting data into a tree requires expensive rebalancing operations. The disadvantage of regular grids over k-d trees is, that they need the grid size as an input parameter where k-d trees find their optimal partitioning by finding minimal bounding boxes as needed.

This section first gives a short overview of several different grid data structures that are used in literature to represent geometries from point clouds. We then detail our approach which is a regular occupancy grid with metadata information.

2.3.2 Related work

The concept of occupancy grids for map building my autonomous mobile robots was first introduced by Hans Moravec and Alberto Elfes in their much cited publications between 1985 and 1992.[18] The authors equipped a mobile robot with a sonar ring and build a 2D occupancy grid map. Each grid cell contains the probability for it being occupied and the probability for it being empty. Another type of occupancy grid map are fuzzy maps which do not store the probability mass function but the possibility degree as a fuzzy set. This concept was first shown in a paper by Oriolo et al.[19] and has been extended by Guadarrama et al. who build fuzzy grid maps modeled by rules from linguistics. In contrast to probability maps or fuzzy maps, their solution does not require a sensor model and is still able to generate robust maps of the environment.[20]

Another popular, voxel-based approach for robotic mapping ist he cumulative weighed signed distance function, or SDF, which was proposed in by Curless and Levoy[21] as a method to reconstruct a 3D mesh from range images. At its core, the SDF method

[18] Moravec, H. and Elfes, A. (1985). High resolution maps from wide angle sonar. In *Proceedings. 1985 IEEE international conference on robotics and automation*, volume 2, pages 116–121. IEEE

[19] Oriolo, G., Ulivi, G., and Vendittelli, M. (1997). Fuzzy maps: a new tool for mobile robot perception and planning. *Journal of Robotic Systems*, 14(3):179–197

[20] Guadarrama, S. and Ruiz-Mayor, A. (2010). Approximate robotic mapping from sonar data by modeling perceptions with antonyms. *Information Sciences*, 180(21):4164–4188

[21] Curless, B. and Levoy, M. (1996). A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312

uses a voxel grid where each voxel essentially stores the distance between the voxel center and the closes measured surface. These distances together then allow to compute an implicit surface of the measured object. The method allows one to encode measurement uncertainties by using weights and provides an update function for incremental and independent integration of additional measurements. While Curless and Levoy take measurements using a sensor triangulating between a line laser and a CCD camera, the method became popular 15 years later when Newcombe et al.[22] used the method together with Microsoft's low-cost RGB-D camera Kinect, providing excellent real-time 3D reconstruction and tracking functionality using commodity hardware. Their approach is known as KinectFusion.

[22] Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohi, P., Shotton, J., Hodges, S., and Fitzgibbon, A. (2011). Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136. IEEE

The approach we use for our occupancy grid, also does not require a sensor model and does not compute or store probabilities. Instead, we store a binary map where each voxel either contains points or not and where later, via voxel traversel, voxels are either see-through or not.

2.3.3 *Implementation*

Our own implementation of a voxel occupancy grid heavily relies on C++ Standard Template Library (STL) containers[23] and Boost[24]. At the heart of it, is the definition of the voxel itself as a container holding its three signed coordinates[25].

[23] `http://www.cplusplus.com/reference/stl/`

[24] `https://www.boost.org/`

[25] Testing showed that an `std::tuple` would be just as fast and require the same amount of system memory. The author decided against its use due to the long-winded form of accessing members via `std::get<T>(v)`

```
1  struct voxel{
2      mutable ssize_t x;
3      mutable ssize_t y;
4      mutable ssize_t z;
5
6      voxel(const ssize_t X, const ssize_t Y, const ssize_t Z)
7          : x(X), y(Y), z(Z) {}
8
9      voxel(const struct voxel &other)
10         : x(other.x), y(other.y), z(other.z) {}
11
12     bool operator<(const struct voxel& rhs) const
13     {
14         if (x != rhs.x) {
15             return x < rhs.x;
16         }
17         if (y != rhs.y) {
18             return y < rhs.y;
19         }
20         return z < rhs.z;
21     }
22
23     bool operator==(const struct voxel& rhs) const
24     {
```

```
25          return x == rhs.x && y == rhs.y && z == rhs.z;
26      }
27
28      bool operator!=(const struct voxel& rhs) const
29      {
30          return x != rhs.x || y != rhs.y || z != rhs.z;
31      }
32  };
```

Two constructors allow instantiating new voxels from an existing voxel or from three voxel coordinates. Operator implementations allow sorting and testing for equality. The vertices of each voxel together form a cubic lattice where the voxel coordinate (X, Y, Z) is derived from the cartesian coordinate (x, y, z) of the vertex with the lowest coordinate value, i.e. the vertex in negative direction. The final voxel coordinate is computed by dividing the vector (x, y, z) value by the voxel size V and round down to the nearest integer towards minus infinity:

$$X = \lfloor x/V \rfloor \tag{2.19}$$
$$Y = \lfloor y/V \rfloor \tag{2.20}$$
$$Z = \lfloor z/V \rfloor \tag{2.21}$$

The same relationship is used to associate input points (p_1, p_y, p_z) into their corresponding voxels:

$$X = \lfloor p_x/V \rfloor \tag{2.22}$$
$$Y = \lfloor p_y/V \rfloor \tag{2.23}$$
$$Z = \lfloor p_z/V \rfloor \tag{2.24}$$

While this relationship between cartesian coordinate and voxel coordinate is certainly trivial to express in math, doing the same in a computer language like C++ proves to be tricky because integer rounding will round towards zero and not towards minus infinity. This is because languages like C don't "round" but just truncate the result of the division to the integer part[26]. More formally, computing $q = \frac{a}{b}$ with remainder r using integer division ($q \in \mathbb{Z}$) in nearly all languages satisfies the following conditions:

[26] The truncation of a positive number is $\lfloor x \rfloor$ while the truncation of a negative number is $\lceil x \rceil$

$$a = qb + r \tag{2.25}$$
$$|r| = |b| \tag{2.26}$$

In C/C++ where the quotient is defined by truncation, the remainder r would have the same sign as the dividend a and the quotient q gets truncated towards zero:

$$a = b\text{trunc}\left(\frac{a}{b}\right) + r \tag{2.27}$$

The result of this can be seen in Figure 2.11. One can see, that the values from $-b$ until b all get truncated towards zero. This effect is undesirable as it would map all cartesian coordinate values from $-V$ to V to the voxel with coordinate $(X, Y, Z) = (0, 0, 0)$.

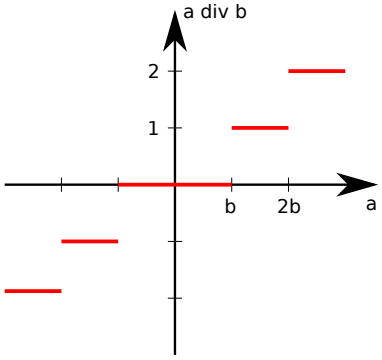


Figure 2.11: div function based on truncation for $b > 0$

To solve this problem, Donald Knuth defines *floored division* in *Art of Computer Programming*[27]:

$$a = b \left\lfloor \frac{a}{b} \right\rfloor + r \tag{2.28}$$

The downside of this definition is, that the remainder will be negative if b is negative which has undesirable consequences for the voxel traversal method. Instead, what we want is Euclidean integer division as formulated by Boute et al.[28] like this:

$$q = \operatorname{sgn}(n) \left\lfloor \frac{a}{|b|} \right\rfloor \tag{2.29}$$

where sgn is the sign function

$$\operatorname{sgn}(x) := \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{2.30}$$

Equation 2.27 will then look like this for euclidean integer division:

$$a = |b| \left\lfloor \frac{a}{|b|} \right\rfloor + r \tag{2.31}$$

A graphical representation of what this yields can be seen in Figure 2.12.

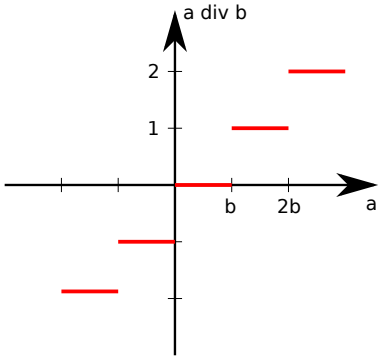


Figure 2.12: div function based on euclidean division for $b > 0$

Since C++ implements integer division by truncation towards zero, a division function was created that implements euclidean division. From Figure 2.11 and 2.12 it can be observed, that to turn the former into the latter, the quotient has to be decremented by one for $b < 0$.

```
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;
}
```

[27] Knuth, D. E. (2011). *The Art of Computer Programming*. Addison-Wesley Professional

[28] Boute, R. T. (1992). The euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(2):127–144

The function `fmod(a, b)` is defined in POSIX.1-2001 and the more recent POSIX.1-2008 and computes the floating point remainder of dividing a by b. Specifically, the return value is $a - qb$ where q is the truncated integer division quotion of a/b, truncated to the next full integer toward zero. It is the floating point function for the integer-only `%` operand[29].

[29] In contrast to popular believe the `%` operator in C does not compute the modulo but the remainder

Since `fmod` also truncates toward zero, we also have to adjust its output in cases where the output is negative due to a sign difference between a and b. The following modulo function is not needed for addressing voxels but for efficient voxel traversal as shown in section 4.4.5. We present it here due to its similarity to the integer division function above.

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

We store the voxel grid inside an `std::unordered_map`, using the `struct voxel` as the key. To allow this, the voxel type needs to be hashable. Since performance is not our prime concern, standard boost facilities are used to map the three dimensional vector of the voxel coordinate to a one-dimensional hash value.

```
namespace std
{
        std::size_t hash<struct voxel>::operator()(struct voxel const& t) const
        {
            std::size_t seed = 0;
            boost::hash_combine(seed, t.x);
            boost::hash_combine(seed, t.y);
            boost::hash_combine(seed, t.z);
            return seed;
        }
};
```

Since the result proved to be fast enough for our purposes, no performance evaluation was carried out. Yet, the implementation of the boost `hash_combine` function is similar to the spatial hashing function presented by Teschner et al.[30]. Specifically, at its core, `hash_combine` computes on 32 bit platforms:

[30] Teschner, M., Heidelberger, B., Müller, M., Pomerantes, D., and Gross, M. H. (2003b). Optimized spatial hashing for collision detection of deformable objects. In *VMV*, volume 3, pages 47–54

```
seed ^= value + 0x9e3779b9 + (seed << 6) + (seed >> 2);
```

Whereas the solution proposed by Teschner et al. computes:

$$hash(x,y,z) = (73856093x \oplus 19349663y \oplus 83492791z) \mod n \tag{2.32}$$

With $\oplus$ being the xor operator, n the hash table size and 73856093, 19349663 and 83492791 some large prime numbers. Despite their disadvantages (e.g. zeroes hash to zeroes), the hash function seems to be good enough for practical applications.

2.4 Summary

This section presented three data structures for working with large 3D point clouds: a k-d tree, a sphere quadtree and a voxel grid. Each of the three data structures address a different problem domain. While k-d trees excel at finding nearest neighbors in euclidean space, the sphere quad tree does the same for angular neighbors. Lastly, a hash-based voxel grid provides a space-efficient representation for an occupancy grid and will later be used for efficient ray traversal.

3
Datasets

Throughout this work a number of datasets will be used. In this section we present the datasets, how they were acquired and list their properties. Many of our datasets are made publicly availble in the Robotic 3D Scan Repository at `http://kos.informatik.uni-osnabrueck.de/3Dscans/`. Some datasets are provided elsewhere and will be indicated as such.

3.1 *Bremen, Randersacker, Würzburg, campus, lecturehall*

These five datasets were collected by our research group using a Riegl VZ-400 laser scanner and registered using slam6d. All but the lecturehall dataset are used for qualitative analysis and runtime benchmarks for our approach to change detection. The lecturehall dataset was specially created to serve as input for a quantitative assessment of our change detection algorithm. All datasets were collected using terrestrial scanning, that is, in a stop-and-go fashion with the Riegl scanner mounted on a tripod.

The field of view of the scanner is 360° horizontally and 100° vertically. The scans have an angular resolution of 0.04° for a native scan resolution of 9000 times 2500 points, for a maximum of 22.5 million points per scan. Since all five datasets, except for the lecturehall dataset are outdoor datasets, the sky takes up a large part of the scan without any measured points and thus typically the datasets contain between 14 to 17 million points per scan. The exception is the indoor dataset lecturehall with 22.3 million points per scan.

Table 3.1: Overview of the datasets obtained using the Riegl VZ-400

name	#points	#scans
Bremen city	215652387	13
Randersacker	194754633	11
Würzburg city	86585411	6
campus	2227455077	146
lecturehall	44574647	1

We collected the "Würzburg city" dataset specifically for qualitative evaluation of 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. Figure 3.1 shows the dataset with points colored by reflectance using the "Jet" color map. Figure 3.2 shows a bird-eye view with reflectivity coloring in a gray scale.

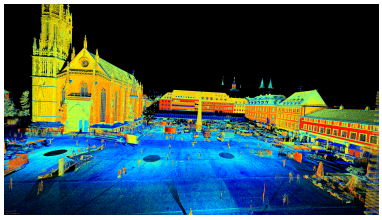


Figure 3.1: Würzburg City Dataset

Figure 3.2: Würzburg City 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 4.11. The dataset was acquired on a Sunday morning with only few people outside. Markers were used to register the dataset.

Figure 3.3: Bremen City Dataset

The "Randersacker" dataset was included because it mainly consists of foliage and other greenery while our other datasets show urban environments with a lot of clear flat surfaces. 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.

Figure 3.4: Randersacker Dataset

The "campus" dataset was acquired on the campus area of Jacobs University Bremen. Since the dataset covers nearly the whole campus, it is the largest dataset with more than 2 billion points in it. Similarly to the "Bremen city" dataset, the "campus" dataset was registered using reflective markers.

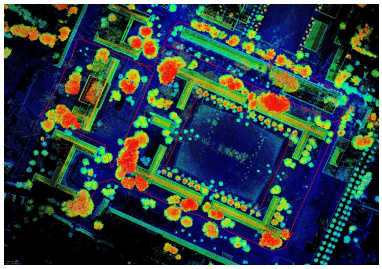


Figure 3.5: Dataset campus

The final dataset lecturehall is shown in Figure 3.6 and is a small lecture hall at the Physics department of Würzburg university, scanned from two vantage points by a Riegl VZ-400 laser scanner. One of the scans has two people in it, holding a blanket, while the other does not. The points in this dataset were manually labeled as dynamic or static to provide a ground truth for the change detection algorithm.

Figure 3.6: Dataset lecturehall without people in it

3.2 *Underwood (sim, lab, carpark)*

To quantify their approach to change detection, Underwood et al.[1] provide three datasets[2] with ground truth annotations, indicating whether each point is dynamic or static.

[1] Underwood, J. P., Gillsjö, D., Bailey, T., and Vlaskine, V. (2013). Explicit 3d change detection using ray-tracing in spherical coordinates. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4735–4741. IEEE

Table 3.2: Overview of the datasets by Underwood et al. and their properties

name	#points	#scans
sim	387838	8
lab	5815910	12
carpark	1965017	4

Table 3.2 shows the total number of points and the number of scans in each dataset. The sim dataset in Figure 3.7 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 Figure 3.8 and Figure 3.9 is from a robot moving through a cluttered lab environment with small boxes ($14 \times 17 \times 40$ cm) being present or not at multiple locations. The dataset was recorded using a Velodyne HDL64ES2 laser scanner at the Australian Centre for FieldRobotics (ACFR).

The third dataset carpark from Figure 3.10 consists of four stationary scans in a carpark environment where a car was moved into different positions for each scan. Like the lab dataset, it was recorded with a Velodyne laser scanner and ground truth information was added manually and reflected points were removed. The dataset was wrongly aligned, so we registered the scans again using the ICP implementation from `slam6D` before passing the points to each algorithm.

3.3 *KITTI*

The KITTI dataset[3] is a popular dataset for computer vision benchmarks from the Karlsruhe Institute of Technology. They equip a standard station wagon (see Figure 3.11) with two high-resolution color and grayscale video cameras, a Velodyne laser scanner and a GPS localization system with an IMU. The focus of the dataset lies on object recognition, object tracking and visual odometry or laser-based SLAM. For these purposes, the dataset comes with extensive ground truth annotations in form of 2D and 3D bounding boxes for the 2D camera images and the 3D point cloud from the laser scanner, respectively. But while objects like cars, vans, trucks, pedestrians, trams and cyclists are correctly annotated, this does not yet supply a fitting ground truth for the evaluation of our approach to change detection because many of these objects can also be stationary, like parked cars or sitting people.

[2] `http://www.acfr.usyd.edu.au/papers/icra13-underwood-changedetection.shtml`

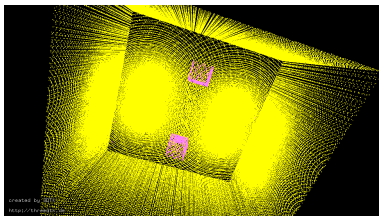


Figure 3.7: Dataset sim

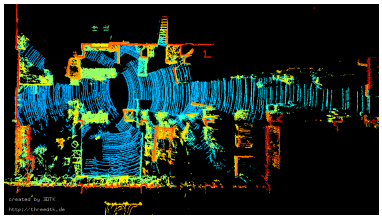


Figure 3.8: Dataset lab seen from above

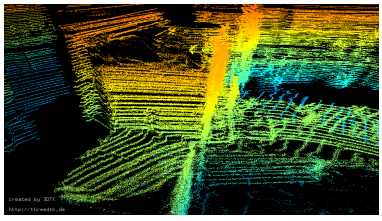


Figure 3.9: Dataset lab showing noise

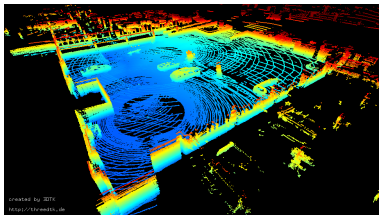


Figure 3.10: Dataset carpark

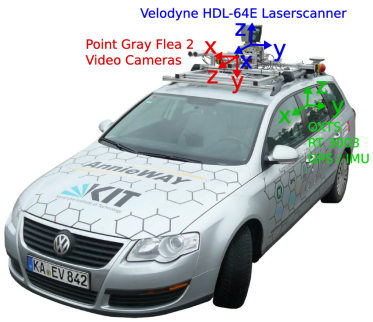


Figure 3.11: KITTI setup by Geiger et al.

[3] Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*

To still use the KITTI dataset for evaluation, we make use of the third party annotations provided by the authors of MODNet[4] in form of the MoSeg-KITTI Motion Segmentation or KITTI MoSeg dataset. That dataset provides binary masks for 1300 images from different scenes from the KITTI dataset. This data was further expanded by the authors of FuseMODNet[5] to binary masks for 12919 images. The masks apply to the images captured by the left camera on the KITTI measurement setup. White pixels signify moving objects and black pixels mark static objects in the captured image. The data was collected for datasets from the "city", "residential" and "road" categories as they can be downloaded from `http://www.cvlibs.net/datasets/kitti/raw_data.php`.

Since change detection approaches are highly sensitive to registration errors, the KITTI scans were registered using slam6d. For this purpose, the scan locations were transformed from WGS-84 coordinates into ECEF coordinates, centered at position of the first scan. This transformation is not only useful because most software requires cartesian coordinates as input but also because the 32 bit floating point datatype is not precise enough to store large values as they are common in WGS-84 coordinates to a precision necessary for change detection. Attempting to handle WGS-84 positions as 32 bit floats will commonly result in a loss of precision in the range of several decimeters.[6]

As the provided binary masks only apply to the image recorded by the left camera and not to the point cloud itself, we use the calibration information between camera and velodyne scanner to segment the point cloud into points that are static, points that are not static and points that lie outside the field of view of the camera.

The dataset properties can be seen in table 3.3. The first column shows the ID of the scene in the nomenclature used by the KITTI website. The second column shows the number of scans in that scene. Each scan consists of a 3D point cloud as captured by the Velodyne laser scanner, 2D images from the cameras and GPS and IMU information. Measurements were taken at a rate of 10 Hz. The third column shows the total number of points in the scene. On average each scan consts of 120.000 points. The fourth column shows the remaining number of points in the scene after the 2D masks have been applied. On average, 15.9% of the recorded points remain from the original point cloud. The fifth column shows how many points in the dataset were marked as dynamic. The sixth and last column shows how much percent of the points in column four were marked as dynamic.

Table 3.3: Properties of the KITTI dataset

ID	#scans	#points	#in view	#dynamic	%dynamic
1	108	13178862	2061966	0	0
2	77	9385524	1460458	509	0.03
5	154	18746749	3024753	71914	2.37

[4] Siam, M., Mahgoub, H., Zahran, M., Yogamani, S., Jagersand, M., and El-Sallab, A. (2017). Modnet: Moving object detection network with motion and appearance for autonomous driving. *arXiv preprint arXiv:1709.04821*

[5] Rashed, H., Ramzy, M., Vaquero, V., El Sallab, A., Sistu, G., and Yogamani, S. (2019). Fusemodnet: Real-time camera and lidar based moving object detection for robust low-light autonomous driving. In *The IEEE International Conference on Computer Vision (ICCV) Workshops*

[6] Karlsruhe is situated at a latitude of 49.00921°. The next 32 bit floating point number that represents that value is 49.0092086792. The next 32 bit floating point number afterwards is 49.0092124939, making a difference of 0.0000038147° between the two. At this latitude, each degree corresponds to 111210 m. This means that the distance between two representable poses in 32 bit floating point is 42 cm at that latitude.

ID	#scans	#points	#in view	#dynamic	%dynamic
9	443	51397378	8324420	122534	1.47
11	233	26224153	4227742	137310	3.24
13	144	17347579	2671305	88580	3.31
14	314	37846323	5834016	145539	2.49
15	297	35610272	5432985	125708	2.31
17	114	13327230	2156064	64739	3.00
18	270	31955563	5108984	185341	3.62
19	481	59808295	9545508	122182	1.27
20	86	10543977	1696864	1836	0.10
22	800	97010620	15486296	91027	0.58
23	474	55250697	8873180	2175	0.02
27	188	23071194	3632449	25949	0.71
28	430	52927931	8365825	41805	0.49
29	430	50903961	8377487	140199	1.67
32	390	46049509	7108348	128066	1.80
35	131	15296575	2532189	526	0.02
36	803	94239228	14857440	257062	1.73
39	395	47036670	7438221	45823	0.61
46	125	15247051	2449682	52303	2.13
48	22	2603914	412239	26381	6.39
51	438	50608329	7873791	444033	5.63
52	78	8749723	1298085	0	0
56	294	35967724	5657548	91119	1.61
57	361	41745641	6131022	207255	3.38
59	373	43835683	6911311	206065	2.98
60	78	8414744	1514989	4081	0.26
61	703	86608211	13849465	0	0
64	570	69835646	11094105	8758	0.07
70	420	52455735	8333931	35235	0.42
79	100	12520385	2033490	0	0
84	383	45327855	7271586	58012	0.79
86	706	88547628	14158549	296	0
87	729	91361891	14542479	0	0
91	340	41662792	6681938	544	0
93	433	50949973	8130679	0	0

3.4 *El Teniente, Hannover, Wolfsburg, Traintunnel*

The datasets “El Teniente”, “Hannover”, “Wolfsburg”, “Traintunnel” and “Trainwagon” were used for collision detection. Each dataset is comprised of the pointcloud of the model, the pointcloud of the environment and the 6 DOF trajectory that the model takes through the environment. The trajectory is a sequence of transformation matrices describing the rotation and translation (but not scaling or shearing) of the model at each step.

The first column in table 3.4 shows the name of the dataset, the second column shows the number of points in the environment, the

third column the number of points in the model, the fourth column the number of discrete positions along the trajectory and the fifth and sixth column the number of colliding points on the CPU and GPU, respectively.

Table 3.4: Overview of the used datasets and their properties

Name	#Environment	#Model	#Trajectory
El Teniente	806183400	100000	17795
Hannover	55872714	214489	17234
Wolfsburg	350109065	434700	398999
Train Tunnel	18919000	28622	19392

3.4.1 *El Teniente*

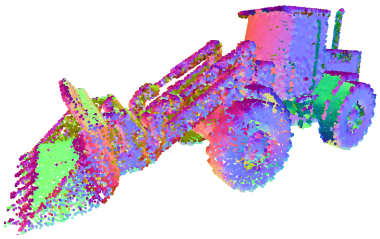


Figure 3.12: Point cloud of a front loader colored by surface normal

The "El Teniente" dataset was collected by Leung et al.[7] in a stop-and-go fashion in the El Teniente underground copper mine in Chile with the Riegl VZ-400.[8] The trajectory follows the path along which the scanner was moved in a closed loop. Due to the stop-and-go scanning method, the point density is highest around the 44 positions where a scan was carried out. The individual terrestrial scans were registered using markers and the batch optimization method by Borrmann et al.[9] A synthetic point cloud of a 3D model of a front loader was moved through this dataset. The point cloud of the front loader was acquired by sampling the surface of a CAD model such that the final point cloud contained 100000 points. The trajectory was produced by fitting a B-Spline through the 44 scanning positions. The model was rotated as if it was driving along the trajectory. Due to the synthetic nature of the model as well as the trajectory, more collisions are produced than in a real-world scenario. The dataset can be downloaded from `http://dataset.amtc.cl/`.

Figure 3.13: Husky A200 robot with Riegl VZ-400 inside the mine by Leung et al.

[7] Leung, K., Lühr, D., Houshiar, H., Inostroza, F., Borrmann, D., Adams, M., Nüchter, A., and Ruiz del Solar, J. (2017). Chilean underground mine dataset. *The International Journal of Robotics Research*, 36(1):16–23

[8] A fly-through video can be found on YouTube: `https://youtu.be/ZjxKzYmhlP4`

[9] Borrmann, D., Elseberg, J., Lingemann, K., Nüchter, A., and Hertzberg, J. (2008). Globally consistent 3d mapping with scan matching. *Robotics and Autonomous Systems*, 56(2):130–142

[10] Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014a). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2(5):117

3.4.2 *Hannover and Wolfsburg*

The "Hannover" and "Wolfsburg" datasets were collected in production facilities of the automotive company Volkswagen in their factories in Hannover and Wolfsburg, respectively. Both datasets were collected using continuous laser scanning with a FARO Focus3D sensor on a mobile platform which was moved on the production conveyor[10]. The characteristics of these two point clouds are very similar because of the similar environment and the similar scanning methods. The main difference is, that the "Wolfsburg" dataset is much larger and produced the longest run times in our test due to its size, the high sampling rate along its trajectory as well as using a model with twice the amount of points compared to

the "Hannover" dataset. We used pointclouds extracted from actual CAD models of the Volkswagen Crafter and Tiguan car bodies for the "Hannover" and "Wolfsburg" datasets, respectively. The trajectory was retrieved from the scanner positions and transformed such that a realistic simulation of the movement of the car body along the production line is achieved.

3.4.3 *Traintunnel and Trainwagon*

The dataset called "Train Tunnel" was recorded by the company TopScan GmbH using mobile mapping from an Optech Lynx Mobile Mapper laser scanner mounted on a van on a train wagon[11]. Figure 3.14 shows the setup.

[11] Schauer, J. and Nüchter, A. (2014). Efficient point cloud collision detection and analysis in a tunnel environment using kinematic laser scanning and kd tree search. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 40(3):289

Figure 3.14: The Optech Lynx Mobile Mapper on the back of a train wagon.

TopScan also provided the trajectory data to us which is comprised of 23274 positions over a distance of 1144 *m*. The dataset contains a tunnel environment as well as an open outdoor environment before and after the tunnel. Datasets of this kind can also be acquired using the sensor skid system presented by Elseberg et al.[12]

The model moved through the environment was a manually scanned train wagon which we moved along a trajectory that allowed us to simulate a bogie size of 20 *m* of that train wagon, leading to collisions that could not have been detected with a structure gauge based method.

[12] Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014b). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5:117–122

To retrieve a point cloud of a suitable model to move through the environment, the train wagon that is seen in Figure 3.15 was manually scanned using a RIEGL VZ-400 laser scanner (see Figure 3.16). Seven scans were taken from all sides of the wagon and registered using 3DTK's SLAM implementation .

Figure 3.15: A photo of the scanned train wagon with a bogie distance of 20 m.

The train wagon is manually extracted from the resulting registered point cloud by using 3DTK's `show` application . As the train wheels are still part of the wagon, they will always result in an expected collision with the rails themselves.

As calibration data of the precise location of the scanner relative to the environment is missing, our results can only serve a demonstration purpose of our methods (see Figure 3.17). The final point cloud of the wagon contained 2.5 million points.

Figure 3.16: The Riegl VZ-400 laser scanner set up next to the train wagon.

The trajectory provided to the authors included orientation information in three degrees of freedom as well. Since a train wagon is mounted on two bogies and since the origin of the coordinate system of the train is located in its center, using this trajectory directly would mean that the wagon would rotate around its own center along the trajectory. This produces wrong results since instead, the bogies of the train have to remain on the tracks while the center follows accordingly. A new trajectory is calculated from the

original trajectory by assuming a bogie distance of 20 m and moving the train wagon such that the center of both bogies is always on the original trajectory. Since this operation requires the original trajectory to be a continuous function and not a sampled trajectory, a spline is fitted across all points of the trajectory with a sum of squared residuals over all the spline's control points of 10 m. This amounts to the spline only a few millimeter away on average from the original trajectory. The FITPACK library[13] is used to calculate the spline. The result of this computation also adjusted the yaw and pitch of the trajectory.

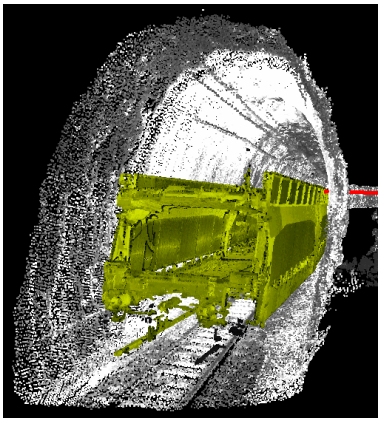


Figure 3.17: Aligned train wagon (yellow) inside the tunnel environment (gray) and trajectory (red).

[13] Dierckx, P. (1993). *Curve and surface fitting with splines*. Oxford University Press, Inc

4
Change detection

For the purpose of visualization and further post-processing of 3D point cloud data, it is 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.

4.1 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 datasets.

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

4.1.1 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 sonar, RADAR or RGB-D systems or point clouds from stereo vision. Any input which allows associating every 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 Figure 4.1 where pedestrians in the foreground and cars in the background are classified as non-static and subsequently removed in Figure 4.2.

Figure 4.1: Non-static points are identified (magenta)...

Figure 4.2: ...and removed without artifacts

4.2 *Related work*

Our solution falls into the realm of change detection[2] 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[3]. Or the solution shown in by Liu et al.[4] 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 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 Under-

[2] Qin, R., Tian, J., and Reinartz, P. (2016). 3d change detection–approaches and applications. *ISPRS Journal of Photogrammetry and Remote Sensing*, 122:41–56

[3] Vieira, A. W., Drews, P. L., and Campos, M. F. (2014). Spatial density patterns for efficient change detection in 3d environment for autonomous surveillance robots. *IEEE Transactions on Automation Science and Engineering*, 11(3):766–774

[4] Liu, K., Boehm, J., and 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*, volume 41, pages 309–313. International Society of Photogrammetry and Remote Sensing (ISPRS)

wood et al.[5] 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 et al. [6] 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. [7] 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.[8] 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[9] also use the same algorithm as we do by Amanatides and Woo[10] to cast rays. But instead of voxels they use an octree 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 by Ruixu et al.[11]

Besides voxels and octrees other data structures to store occupation information in are elevation maps[12], multi-level surface

[5] Underwood, J. P., Gillsjö, D., Bailey, T., and Vlaskine, V. (2013). Explicit 3d change detection using ray-tracing in spherical coordinates. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4735–4741. IEEE

[6] Gálai, B. and 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*, pages 210–220. Springer

[7] Xiao, W., Vallet, B., and Paparoditis, N. (2013). Change detection in 3d point clouds acquired by a mobile mapping system. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 1(2):331–336; and Xiao, W., Vallet, B., Brédif, M., and Paparoditis, N. (2015). Street environment change detection from mobile laser scanning point clouds. *ISPRS Journal of Photogrammetry and Remote Sensing*, 107:38–49

[8] Asvadi, A., Peixoto, P., and Nunes, U. (2016a). Two-stage static/dynamic environment modeling using voxel representation. In *Robot 2015: Second Iberian Robotics Conference*, pages 465–476. Springer; and Asvadi, A., Premebida, C., Peixoto, P., and Nunes, U. (2016b). 3d lidar-based static and moving obstacle detection in driving environments: An approach based on voxels and multi-region ground planes. *Robotics and Autonomous Systems*, 83:299–311

[9] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206

[10] Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10

[11] Ruixu Liu, V. K. A. (2017). 3d indoor scene reconstruction and change detection for robotic sensing and navigation

[12] Herbert, M., Caillas, C., Krotkov, E., Kweon, I. S., and Kanade, T. (1989). Terrain mapping for a roving planetary explorer. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 997–1002. IEEE; and Pfaff, P., Triebel, R., and Burgard, W. (2007). An efficient extension to elevation maps for outdoor terrain mapping and loop closing. *The International Journal of Robotics Research*, 26(2):217–230

maps[13], and Gaussian Mixture Models[14]. Existing methods that require computation of free volume for robotic path planning are known to use a 3D Bresenham ray casting kernel[15] carrying out the ray casting in many parallel threads on the GPU. GPU-based ray casting techniques were first shown by Roettger et al.[16] and Kruger et al.[17] and are today often implemented using OpenGL and CUDA[18].

4.3 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 (voxel)**.** 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

[13] Triebel, R., Pfaff, P., and Burgard, W. (2006). Multi-level surface maps for outdoor terrain mapping and loop closing. In *2006 IEEE/RSJ international conference on intelligent robots and systems*, pages 2276–2282. IEEE

[14] Andreasson, H., Magnusson, M., and Lilienthal, A. (2007). Has somethong changed here? autonomous difference detection for security patrol robots. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 3429–3435. IEEE; Núñez, P., Drews, P., Bandera, A., Rocha, R., Campos, M., and Dias, J. (2010). Change detection in 3d environments based on gaussian mixture model and robust structural matching for autonomous robotic applications. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2633–2638. IEEE; and Drews-Jr, P., Núñez, P., Rocha, R. P., Campos, M., and Dias, J. (2013). Novelty detection and segmentation based on gaussian mixture models: A case study in 3d robotic laser mapping. *Robotics and Autonomous Systems*, 61(12):1696–1709

[15] Hermann, A., Drews, F., Bauer, J., Klemm, S., Roennau, A., and Dillmann, R. (2014a). Unified gpu voxel collision detection for mobile manipulation planning. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4154–4160. IEEE

[16] Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., and Strasser, W. (2003). Smart hardware-accelerated volume rendering. In *VisSym*, volume 3, pages 231–238. Citeseer

[17] Kruger, J. and Westermann, R. (2003). Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society

[18] Weinlich, A., Keck, B., Scherl, H., Kowarschik, M., and 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*, volume 1, pages 25–30

even only once. No heuristic about object movement is required.

Figure 4.3 displays the general idea of our algorithm in a two-dimensional 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. The circular object in areas C1 and C2 is dynamic and only seen by the first scan (Figure 4.3). Since the second scan (Figure 4.4) 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.

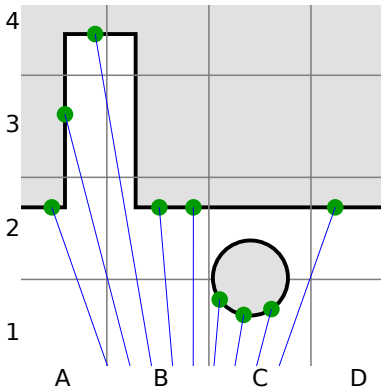


Figure 4.3: The scene as scanned from a center position (ray origin not part of the Figure). The scanner measures the green points.

Figure 4.4 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 Figure 4.3. Still, they are not removed because these areas are never marked as see-through by other scans (for example the second scan in Figure 4.4). The same holds for the red points in area C2. They are only seen by the second scan in Figure 4.4 because the circular moving object in C1 and C2 occludes the points during the first scan in Figure 4.3. Still, the points remain classified as static because their containing areas are never marked as see-through.

Our algorithm goes through the following stages:

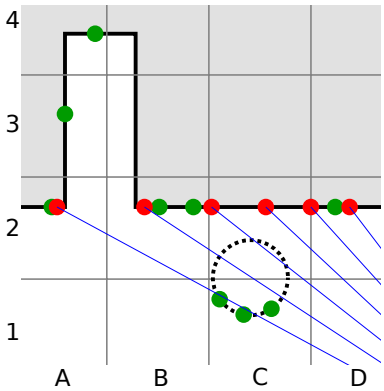


Figure 4.4: The scene as scanned from a position to the right. The circular object in areas C1 and C2 moved away and its former position is marked with dotted lines. The scanner measures the red points.

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. Optionally, computing a reduced set of rays to traverse through the voxel grid
5. Traversing lines of sight through the voxel grid for each scanner location to each measured point by that scan, identifying see-through voxels
6. Clustering for false positive noise removal
7. Removal of false negatives through our approach to sub-voxel accuracy
8. 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 Figure 4.4, 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 Figure 4.3 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 4.8.

4.4 *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 and Woo.[19] 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[20], MRPT[21], PCL[22], yt[23]) supports any of these properties and thus we detail our approach here.

We empirically verified the correctness of our algorithm by comparing it with a brute-force implementation which enumerates all voxels intersecting with a given line by simply checking *all* voxels in the grid for a possible intersection. We generate a synthetic corpus of line segments to check against by going through all permutations of x, y and z coordinates for the start and endpoint of the line segment from a fixed list of input coordinate values. We arbitrarily chose the values -5, -1, 0, 1, and 5 as well as the next smaller and next bigger floating point number around each of them. We then test all these line segments against voxel grids with voxel sizes of 0.1, 1, 2, 5 and 10. All of this together results in more than 24 million different test cases that we check our algorithm against.

[19] Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10

[20] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206

[21] Blanco-Claraco, J. (2014). Mobile robot programming toolkit (mrpt)

[22] Rusu, R. B. and Cousins, S. (2011). 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4. IEEE

[23] Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., and Norman, M. L. (2011). yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement Series*, 192:9

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.

4.4.1 *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 $t \geq 0$. 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.

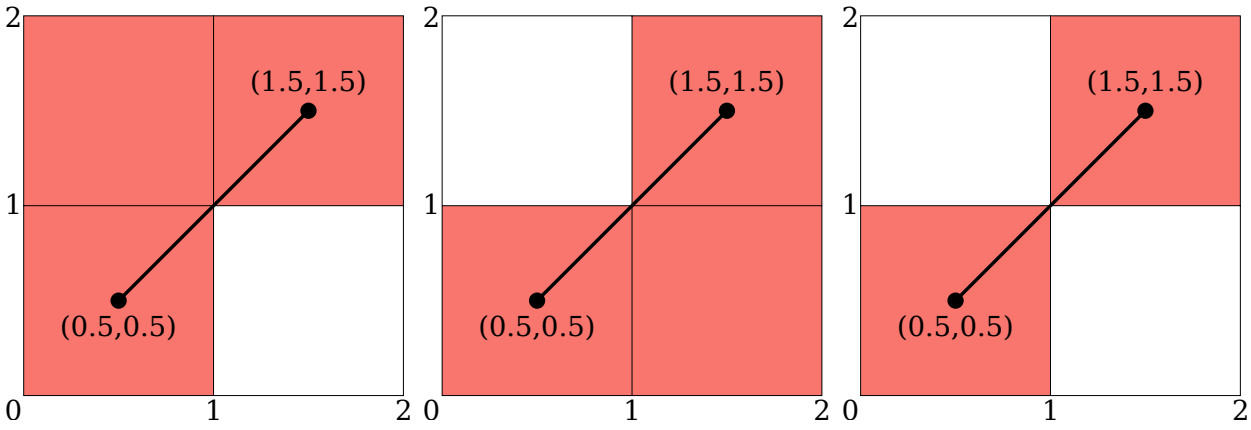
To the best of our knowledge, neither the original publication nor any other publication since then explains in more detail how these variables are set up exactly. We found though, that many of the current limitations of existing implementations suffer from wrongly handled corner-cases in how these variables are set up. Thus, in this section we also describe how to initialize the variables in detail.

4.4.2 *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 (line)**.** 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.



(a) If the traversal algorithm checks the vertical dimension first, then voxel (0,1) is added to the result.

(b) If the traversal algorithm checks the horizontal dimension first, then voxel (1,0) is added to the result.

(c) Our algorithm only traverses voxel (0,0) and voxel (1,1).

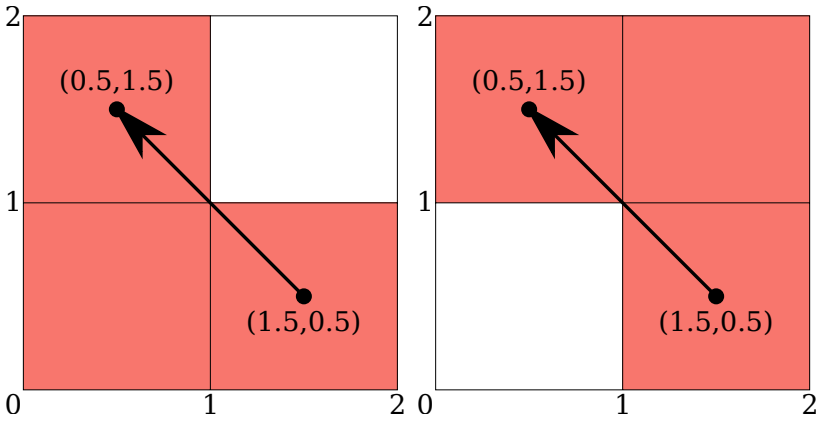
Figure 4.5: Two-dimensional example of the voxel traversal problem

Figure 4.5 shows a 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.

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 Figures 4.5, 4.6 and 4.7. 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.

4.4.3 *Avoiding accumulation of floating point errors*

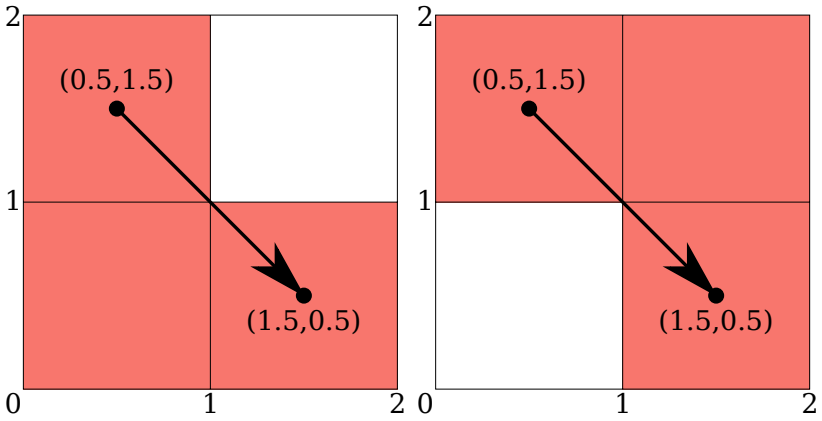
The original algorithm by Amanatides and Woo increments the `tMax` variables by the corresponding `tDelta` value in each loop iteration. Since both variables are floating point values, this accumulates an error over time, especially once `tMax` becomes several



(a) 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)

(b) Our algorithm correctly identifies voxel (1,1) as part of the solution.

Figure 4.6: A line is traversed from (1.5,0.5) to (0.5,1.5). The arrow indicates the direction.



(a) 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)

(b) Our algorithm correctly identifies voxel (1,1) as part of the solution.

Figure 4.7: A line is traversed from (0.5,1.5) to (1.5,0.5). The arrow indicates the direction.

orders of magnitude larger than `tDelta`. Additionally, when the ray is nearly parallel to the coordinate axis, small floating point errors will lead to 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 the algorithm 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 to step next.

4.4.4 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.

4.4.5 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 co-

ordinate 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 section 2.3.3.

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** $i \leftarrow 0, 2$ **do**
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** $step[i] = -1 \wedge tMax[i] = tDelta[i] \wedge startvoxel[i] \neq endvoxel[i]$ **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 section 2.3.3.

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 4.4.4 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 ← (0,0,0)
32:  tMaxStart ← tMax
33:  loop
34:      stepped ← (false, false, false)
35:      minVal ← MIN(tMax)
36:      for i ← 0,2 do
37:          if minVal = tMax[i] then
38:              mult[i] ← mult[i] + 1
39:              curvoxel[i] ← startvoxel[i] + mult[i] · step[i]
40:              tMax[i] ← tMaxStart[i] + mult[i] · tDelta[i]
41:              stepped[i] ← true
42:      if ((stepped[0] ∧ stepped[1])                    then
           ∨ (stepped[0] ∧ stepped[2])
           ∨ (stepped[1] ∧ stepped[2]))
           ∧ (step[0] = 1 ∨ step[1] = 1 ∨ step[2] = 1)
           ∧ (step[0] = −1 ∨ step[1] = −1 ∨ step[2] = −1)
43:          addvoxel ← curvoxel
44:          for i ← 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] ← addvovel[i] + 1
50:                  else if mult[i] > maxMult[i] then
51:                      return
52:          if addvoxel ≠ curvoxel then
53:              VISITOR(addvoxel)
54:      for i ← 0,2 do
55:          if stepped[i] ∧ 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 4.4.3, 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 4.4.2, 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.

4.4.6 Reusing already computed paths

After having shown how the list of voxels intersecting with a line segment is computed, we evaluated the possibility of re-using that list for all target points that share the same path through the voxel grid. We were looking at two approaches. Either partition the input points in a way such that each set of points shares the same path through the voxel grid and thus the path only has to be computed once. Or create a hash which allows one to look up the path toward the current query point from a cache. For any such approach to succeed the following conditions have to be fulfilled:

- There must be significantly more points per voxel than there are unique paths from the ray origin to that voxel. Only then can enough points share the same path and thus avoid recomputation of the path such that the additional memory requirement for storing any computed path is justified.
- Memory requirements must stay within practical limits
- (optional) The cached results from one ray origin (scanner locations) should be reusable for different origins

Unfortunately one has to abandon this idea because none of the requirements was met:

- The number of unique paths to a target voxel increases with its distance from the ray origin, thus making it less and less likely that two target points share the same voxel path
- Memory requirements grow with cubic complexity with increasing distance of the target point from the ray origin
- Stored paths from one ray origin cannot be re-used for other ray origins

Until we find theoretical groundwork that lets us come to these conclusions, we motivate it by using an empirical approach instead. The following variables influence the path that a ray traverses through a regular axis-aligned voxel grid:

- The ray origin relative to the voxel grid boundaries
- The ray direction

- The radius up to which the ray is traversed through the voxel grid
- The voxel grid size

Since we have no reason to suspect to observe fundamentally different results for different ray origins, we arbitrarily chose the coordinate center as the ray origin in our experiments. As far as unique traversed paths through the voxel grid go, the traversal radius and voxel grid size are dependent variables. We thus fix the voxel grid size to a value of 10. The remaining free variables in our experiment are search radius along the ray and the ray direction.

To show empirical evidence for our conclusions, we focus on the relationship between the number of unique voxel paths per search radius. To compute this number, we iterate through radii from 1 to 270 and for each search radius, traverse the voxel grid in over 20 million directions. We record for each search radius how many unique paths were found. The 20 million directions were chosen by recursively subdividing the faces of an icosahedron up to a depth of 10 and using the unit vector to the resulting vertices as direction for the samples.

The two illustrations in Figure 4.8 and 4.9 show the result of this algorithm with a radius of 50 and an aforementioned voxel size of 10. Since the ray origin is at the coordinate origin and thus at a voxel vertex, the result is symmetric in each octant of the resulting sphere. Thus, the figure only shows one of these octants. The others are symmetric to the one shown.

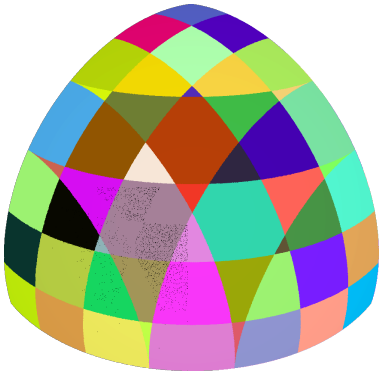


Figure 4.8: Visualization of which part of a sphere surface falls into which voxel

Figure 4.8 visualizes the unique voxels that intersect the sphere surface. Each unique voxel intersection of the surface is represented by a unique color. Voxel paths are only unique if they share the same list of voxels. From this definition we conclude that points falling into different surface voxels cannot share the same path. Thus, for a given radius, there exist at least as many voxel paths as there are surface voxels. The pattern is symmetric for each axis-aligned sphere octant and thus only one of the octants with 52 colored patches is displayed.

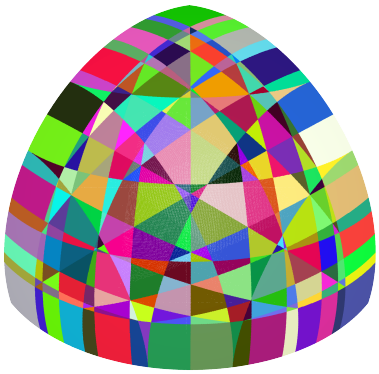


Figure 4.9: Visualization of the unique paths through a regular axis-aligned voxel grid

Figure 4.9 then further subdivides the patches from the surface voxels seen in Figure 4.8. Each unique color signifies a unique path through the voxel grid. One can observe surprising symmetries along multiple axis on the sphere surface. One can also observe how surface voxels get split by differing amounts depending on their location. The pattern is symmetric for each axis-aligned sphere octant and thus only one of the octants with 285 colored patches is displayed.

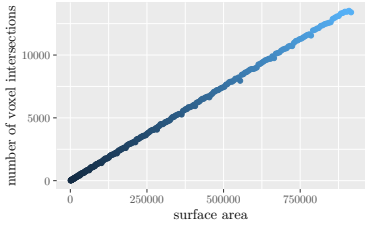


Figure 4.10: Number of voxels in a regular voxel grid intersecting the surface of a sphere per sphere surface area

Figures 4.10 and 4.11 extend the results from Figures 4.8 and 4.9 to 270 different radii. Figure 4.10 shows the number of voxels in a regular voxel grid intersecting the surface of a sphere per sphere

surface area.270 data points were calculated for every integer radius from 1 to 270 (resulting in a decreasing density of data points with increasing surface area). A linear dependency is apparent between the two values.

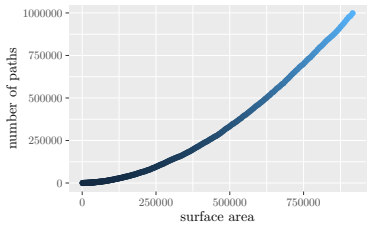


Figure 4.11: number of possible unique paths through a voxel grid

Figure 4.11 shows the number of possible unique paths through a voxel grid of size 10 from the sphere center to the sphere surface per sphere surface area. 270 data points were calculated for every integer radius from 1 to 270 (resulting in a decreasing density of data points with increasing surface area). Since the number of surface voxels is linear dependent on the sphere surface area, we conclude that the number of unique paths to a target voxel increases with its distance from the ray origin.

Put into practical terms: if one repeats the computation for a larger radius, then one finds that with a voxel size of 10 *cm* there exist more than 6 million unique paths to the 46 thousand surface voxels that are at least 5 *m* away. This leads to an average of 140 unique paths per surface voxel. This in turn means that to increase the probability of two points with a distance from the scanner of 5 *m* or more to share a path through the voxel grid, the scan must contain significantly more than 6 million points. While a voxel size of 10 *cm* might already be a big voxel size, the situation worsens with smaller voxel sizes. At the same time, one usually is interested in targets further away than 5 *m*. With common high quality terrestrial laser scans containing in the order of magnitude of 10 million points, it is easy for a scan to contain less points per voxel than there are possible unique paths to that voxel on average.

We thus conclude, that even if there existed a way to precompute the points with a common path through the voxel grid at zero cost, there is little point in doing so because in real world scenarios with voxel sizes of 10 *cm* or less and distances of 5 *m* or more, it becomes increasingly unlikely for two points to share a common path through the voxel grid and thus being able to benefit from such a computation.

We conclude this section with also briefly touching upon the other two reasons why this approach does not fulfill the aforementioned conditions:

- We have shown that the number of paths per radius grows faster than the associated surface area. We know that the surface area of a sphere grows with the square of the radius. The memory requirements to store a path grow linearly with the radius. Thus, the space requirements for storing all required paths grow cubic with the search radius.
- We conducted our results with a fixed ray origin at the coordinate center. The voxel walk is similar to the Bresenham line-drawing algorithm in that its result depends on the sub-voxel starting point of the ray. Thus, the computations made for one ray origin cannot be re-used for others as it is unlikely for two

scanner positions to share the same position relative to the voxel grid boundaries.

4.5 *Scan slices from Mobile mapping*

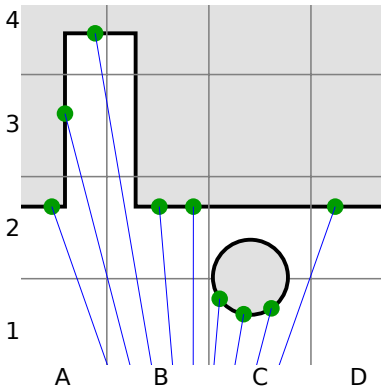


Figure 4.12: The scene as scanned from a center position (ray origin not part of the Figure). The scanner measures the green points.

Before one can apply the voxel walk algorithm to find free voxels for the mobile mapping scenario one problem remains. The scan slice that resulted in measurement of the green point in area A4 of Figure 4.12 illustrates this. Clearly, the line of sight from the scanner towards this point (in blue) passes through area B2. Still, that area should not be marked as free. We solve the problem that partly occupied voxels impose by looking at the neighbor slices of the current scan slice. If the line of sight passes through voxels that also contain points from scan slices adjacent to the scan slice of the current target point, then these voxels are not marked as free and the voxel-walk aborts early without marking the any further voxels as free.

The idea here is, that while we are tracing lines of sight towards individual points, we also always take a sliding window of their neighborhood into account. This technique is similar to the one presented by Hornung et al. [24] to avoid removal of points when scanning surfaces with a small incident angle of the laser beam. The size of the window must be large enough such that the central slice does not share any voxels that we are interested in with the most outer slices. The starting voxel of most rays will be shared by many slices, but having marked the starting voxel as free poses no problem as the space that the scanner was moved through was free to begin with. At the same time, the window size must not be too large. The window must not contain slices that record points of the same object from two completely different scanner rotations. To satisfy both constraints, one input to our algorithm is the number of slices that the scanner records in one full rotation. The window size is then chosen as half that size. This ensures that the "field of view" is large enough to consider adjacent partly occupied voxels and that it is not so large as to have the same object twice in a single "field of view". An additional assumption here is also that the scanner is never turned quickly enough against its own orientation such that this constraint is violated. In current practical setups in factory environments, this is not likely to happen.

We presented the results of this approach in a publication that was accepted at ICCA 2017[25]. In this thesis we present a generalized algorithm which does not require subsequent scans to be spatially close to each other and is free of the constraint of scans to be processed in their temporal order and also does not impose any restrictions on knowing sensor configuration settings like the number of slices per rotation. It is thus capable of processing scans coming from terrestrial mapping.

[24] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206

[25] Schauer, J. and Nüchter, A. (2017). Digitizing automotive production lines without interrupting assembly operations through an automatic voxel-based removal of moving objects. In *Control & Automation (ICCA), 2017 13th IEEE International Conference on*, pages 701–706. IEEE

4.6 *Panorama scans from Terrestrial mapping*

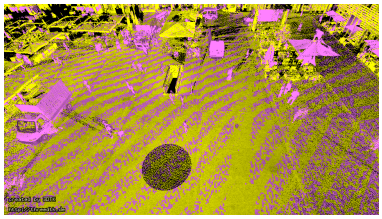


Figure 4.13: Artifacts of false positives on the ground using a naive approach

Attempting to apply the same technique from mobile mapping to terrestrial scans will result in unexpected false-positives as they are visualized in Figure 4.13. The figure shows the result of a naive approach to detecting free voxels by directly using the method from mobile mapping. Dynamic points are marked in magenta. There are several "stripes" of false positives on the ground. This section explains where this effect comes from and how we prevent it by computing "point shadows".

The effect shown in Figure 4.13 is created from the alignment of the ground relative to the voxel grid together with an effect shown in Figure 4.14. 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. The figure shows a scene where, 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.

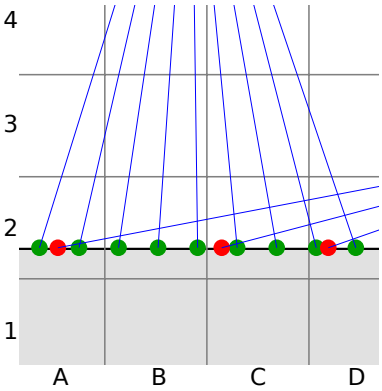


Figure 4.14: False positives variant 1

From that Figures 4.13 and 4.14 it seems apparent that the problem is only due to the small incident angle but as Figure 4.15 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. Figure 4.15 shows a scene where, 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.

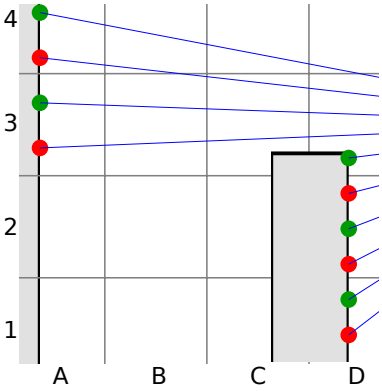


Figure 4.15: False positives variant 2

A solution implies not traversing the lines of sight towards the red points in A2 and A3 in Figure 4.14 and 4.15 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 Figure 4.14 but not the problem shown in Figure 4.15. Instead, the offset in Figure 4.15 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 Figure 4.14 and 4.15 we do not simply clip the rays toward the shadowed points by the distance of the point casting the shadow from 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.

Figure 4.16 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.[26] 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.

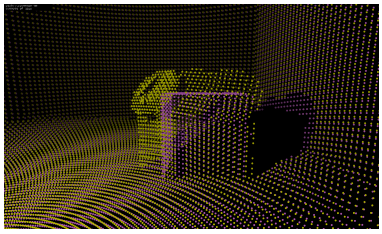
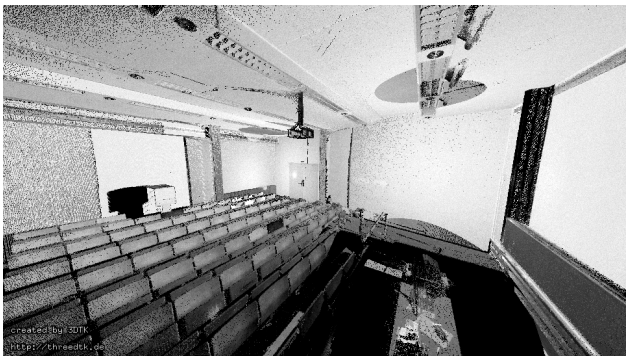
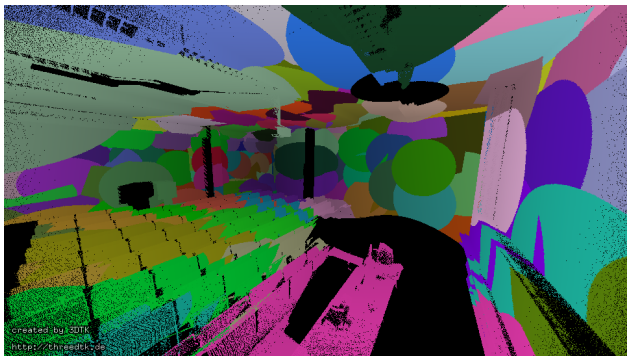


Figure 4.16: Synthetic dataset "sim" from Unterwood et al.
[26] Underwood, J. P., Gillsjö, D., Bailey, T., and Vlaskine, V. (2013). Explicit 3d change detection using ray-tracing in spherical coordinates. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4735–4741. IEEE

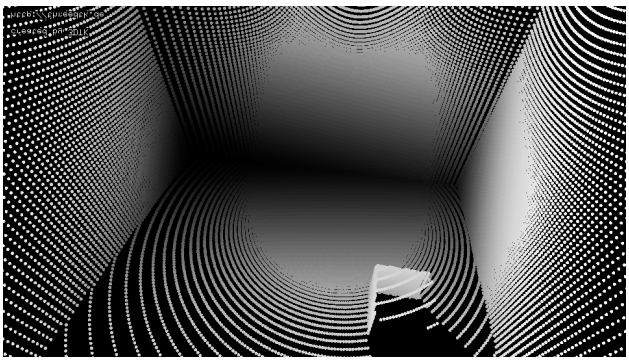


(a) colored by reflectance

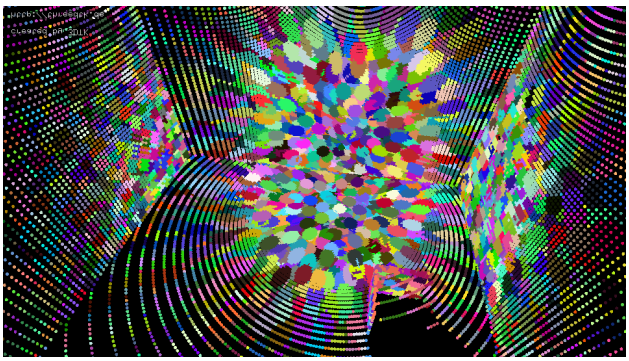


(b) colored by point shadow

Figure 4.17: lecturehall dataset in perspective projection



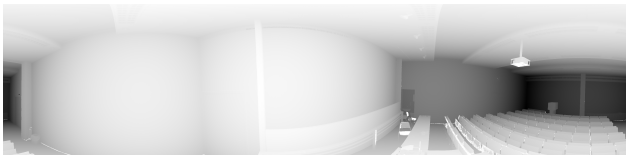
(a) colored by distance



(b) colored by point shadow

Figure 4.18: sim dataset in perspective projection

Figures 4.17 and 4.18 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. For visualization purposes, a very large voxel size of 20 *cm* was chosen for the "lecturehall" dataset.



(a) colored by distance

(b) colored by point shadow

Figure 4.19: lecturehall dataset panorama

The dependency of the shadow size on its distance is best visible from the panorama images in Figure 4.19. Bright areas in the panorama image on the left represent points close to the scanner. These areas create big shadows as is shown in the right-hand-side panorama image. Darker points are further away and create smaller shadows.

4.6.1 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: $normal \leftarrow$ CalcNorm($neighbors$)
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** $maxrange[p] < 0$ **then** ▷ Scanner behind plane
20: $maxrange[p] = 0$
21: **for** $q \leftarrow neighbors$ **do**
22: **if** $p = q$ **then** ▷ 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** $d < 0$ **then** ▷ Scanner behind plane
31: $d \leftarrow 0$
32: **if** $maxrange[q] < d$ **then** ▷ Already shadowed by a closer one
33: **continue**
34: $maxrange[q] \leftarrow d$

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 range search utilizes the spherical quadtree for fast lookups. 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. As will be shown in the qualitative assessments in section 4.10.4, fewer than half a percent of the points meet that criteria in our datasets.

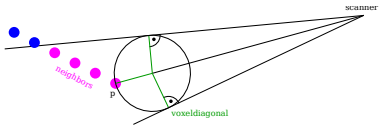


Figure 4.20: Step 1: compute point shadow

Figure 4.20 visualizes lines 6 and 7 from algorithm 4. 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. 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. 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.

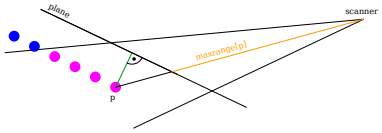


Figure 4.21: Step 2: compute normal

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 Figure 4.21. 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. 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 Figure 4.21, 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.

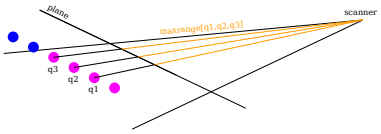


Figure 4.22: Step 3: capping the search distance at the computed plane

Figure 4.22 visualizes lines 21 to 34 from algorithm 4. 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. 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 have been computed in an earlier iteration (line 32).

4.7 *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 (neighbors)**.** 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 data structure, 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.

4.8 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 Figure 4.23, 4.24 and 4.25 to illustrate our approach to achieve subvoxel accuracy and reduce the number of false negative classifications. The three figures show two scans as red and green points of a horizontal surface and dynamic points only seen in the red scan.

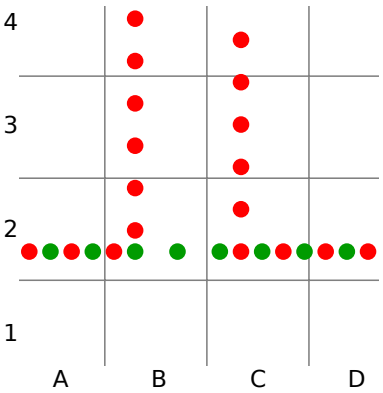


Figure 4.23: Initial situation

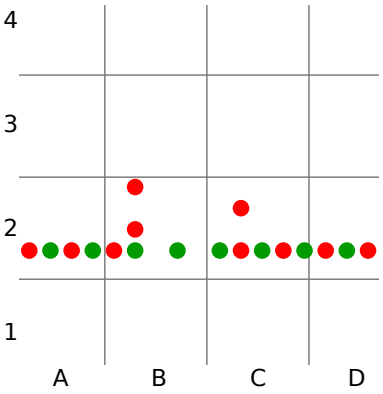


Figure 4.24: See-through voxels cleared

The original input with both scans and nothing removed is shown in Figure 4.23. 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 Figure 4.24. Voxels B3, B4, C3 and C4 got correctly classified as “see-through” and points in them were removed. What remains are false negative artifacts in voxel B2 and C2. Classifying

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

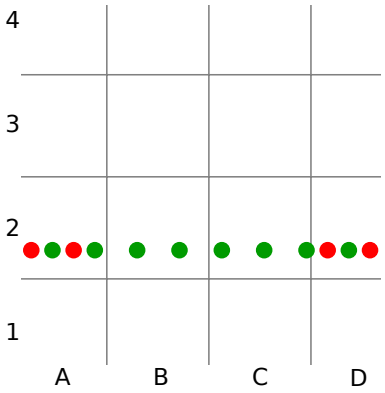


Figure 4.25: Adjacent voxels cleared of points from scan that was removed

The algorithm we use to remove these false negatives from voxels B2 and C2 will compute a situation as is seen in Figure 4.25: 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 Figure 4.25 is superior to the result in Figure 4.24. 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 Figure 4.23 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 Figure 4.23 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.

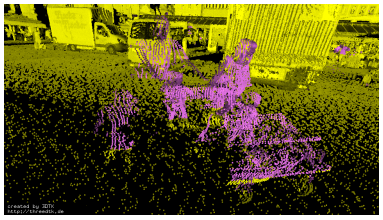


Figure 4.26: No subvoxel accuracy with dynamic points in magenta.

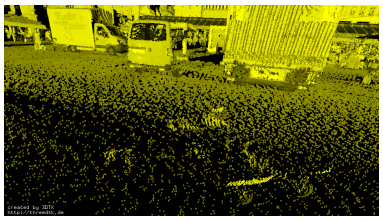


Figure 4.27: No subvoxel accuracy with leftover false negatives on the ground

Figures 4.26 and 4.27 show the result of the change detection algorithm without the algorithm for subvoxel accuracy applied. Dynamic points are colored magenta while static points are yellow. Both figures show, how some false negatives remain on the ground

as artifacts because they are part of voxels which intersect with the ground plane. Marking these voxels as free would be wrong because it would cause all points (including those from the ground) to be removed and thus introduce false positives.

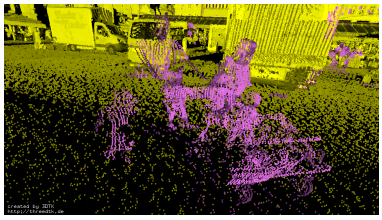


Figure 4.28: With subvoxel accuracy and dynamic points in magenta.

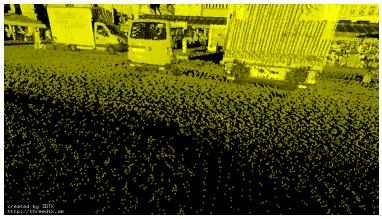


Figure 4.29: With subvoxel accuracy no false negatives remain on the ground.

Figures 4.28 and 4.29 show the same scene, but this time with the algorithm of subvoxel accuracy applied. The artifacts that remained on the ground as false negatives in Figures 4.26 and 4.27 are now removed. Points from the ground are also missing (false positives) but no noticeable holes are created because the respective voxels still contain points from all the other scans which also measured that volume close to the ground.

4.9 *Working on a reduced pointcloud*

This section evaluates a variant of the change detection algorithm where rays are not shot to all measured points but only to a specially chosen subset of them. We show that depending on the scenario, it's possible to shoot up to two orders of magnitude less rays through the scene without significantly affecting the F_1 score. Since the runtime of voxel-grid traversal linearly depends on the number of traversed rays, the runtime of the voxel traversal can similarly be reduced by up to two orders of magnitude without affecting the quality of the output.

The central idea is, that our approach to change detection marks voxels as free, even if they are traversed only once. A Riegl scan with an angular resolution of 0.04° will produce point clouds with up to 22.5 million points per scan. This means, that for each scan, up to 22.5 rays will be traversed through the voxel grid. Or in other terms: even at over 140 m distance, rays will not be further apart than a typical voxel size of 10 cm[27] thus allowing reliable change detection of a volume of a sphere with 280 m in diameter.

For many applications, like indoor environments, the input data will be far below that magnitude. Even in outdoor environments, like urban settings, such distances are rare. For example our Würzburg city dataset measured the Würzburg marketplace which is only 60 m across. Since voxels are marked as free even if they are traversed only once, in many settings it does not make sense to shoot the maximum number of rays as this will mean that the same voxel is traversed multiple times and the closer a voxel is to the origin, the more often it will be (uselessly) traversed.

For these kind of situations, when the scanned point cloud is very dense or if only the volume close to the scanner is of interest[28] it is sufficient to only traverse a subset of all possible rays towards the measured points. Since common scanner geometries result in an uneven angular density of the measured points (most dense around the primary rotation axis) a simple random sampling would again

[27] Due to the scanning pattern, point density is greatest at the poles and changes with azimuth angle. At the equator, the point density is lowest and the distance between two rays with angle θ between them at distance h from the origin can be computed as $2h \tan \frac{\theta}{2}$. With an angular resolution of 0.04° and at a distance $h = 140$ m this yields a distance of 9.77 cm

[28] like in an assembly line environment, see datasets Hannover and Wolfsburg

result in an uneven angular density. To achieve an even angular distribution of the traversed subset of rays, it would be possible to bin the points by their azimuth angle and then choose a random sample from each bin such that the overall number of points per unit sphere surface area stays equal across the whole surface. But such a method would only work if the scanner geometry is known upfront.

As outlined in section 2.2.5, the spherical quadtree can be used to obtain an even angular subsampling of the complete point cloud. Using the spherical quadtree reduction has the advantage, that the tree itself has already been computed to compute the point shadows and can be re-used for the purpose of point cloud reduction directly. For the lecturehall dataset with 22.3 million points per scan, acquiring the reduced point cloud took less than a second for all evaluated sample sizes and thus did not add significant overhead.

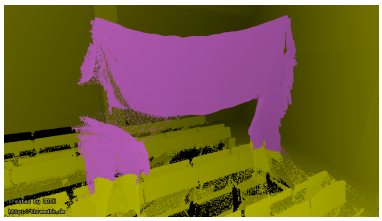


Figure 4.30: Lecturehall with all rays traversed for 22.3 million rays per scan.

Figure 4.30 shows the result of detecting changes in the lecturehall dataset with the original algorithm and overall 44.6 million traversed rays for an F_1 score of 0.955. The result is nearly indistinguishable from using only 334712 rays which resulted in an F_1 score of 0.952 at while the required runtime the voxel traversal step is reduced by two orders of magnitude (0.71 seconds instead of 72 seconds). This improvement in runtime is not much changed by taken into account the time it took to reduce the point cloud because the reduction step is carried out in less than a second.

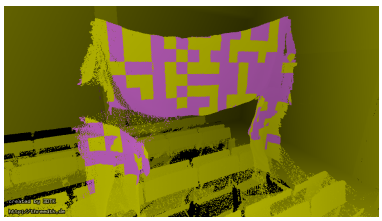


Figure 4.31: Lecturehall with only 10 rays traversed per 2.86° angle for 12k rays per scan.

Figure 4.31 shows the result of choosing an insufficient number of rays for the given dataset. In contrast to Figure 4.30, which shows a near-perfect result, Figure 4.31 illustrates how voxels which were not intersected by the reduced set of rays were not found to be empty and show up as false negatives. This result is not surprising, because the object shown in Figures 4.30 and 4.31 is about 3 meters away from the laser scanner. A sphere with that radius has a surface area of over 113 m^2. This leaves an average of about 94 cm^2 for each ray, which is about the same size as the intersection of a voxel of 10 cm with that sphere surface. Since the ray selection is done at random, some voxel at that distance will not get traversed at all (resulting in the false negatives seen in Figure 4.31) while others will get traversed once or more.

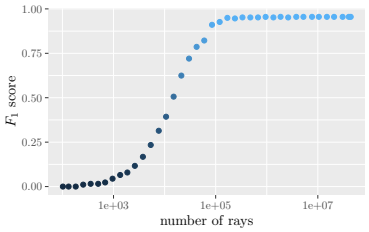


Figure 4.32: F_1 score by number of shot rays with a logarithmic x-axis

In Figure 4.32 we evaluated this method for the lecturehall dataset and plotted the resulting F_1 score by number of rays that were traversed. The original point cloud of 44.6 million points was reduced using the spherical quadtree. The reduction options were set such that 10 randomly chosen points would be selected per angular neighborhood. The angle size θ was chosen with $\sqrt{10^{-6}\sqrt{2}^{i}}$ with i in whole integer steps from 1 to 40. The figure shows how close to optimal F_1 scores are achieved for all subsamplings above

100000 points.

Lastly, Figure 4.33 shows empirical proof to our claim that the runtime of the ray traversal linearly depends on the number of rays. The x-axis is identical to the one in Figure 4.32 and both axes are scaled with a logarithm with base 10.

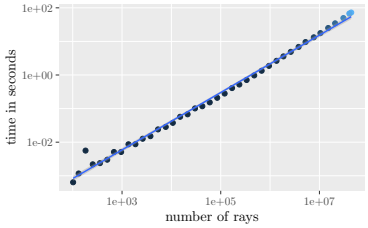


Figure 4.33: time for ray traversal by number of shot rays with regression line and 95% confidence interval

4.10 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 the method by Underwood et al. in terms of runtime and solution quality.

4.10.1 Quantitative Assessment

To perform quantitative analysis of our method, we compute the F_1 score[29] of our method on the datasets sim, lab, carpark, lecturehall and KITTI, all of which come with ground truth annotations. To establish that our method is an improvement over the state of the art, we compare our method to the one by Underwood et al in terms of their respective F_1 scores in different scenarios.

The quantitative results in this section can be reproduced by executing two shell scripts that we provide for download:

- For datasets sim, lab, carpark, lecturehall: `https://robotik.informatik.uni-wuerzburg.de/telematics/download/isprs2018/`
- For KITTI datasets: `https://robotik.informatik.uni-wuerzburg.de/telematics/download/kitti2020/`

The scripts 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, producing the F_1 scores found in the tables below.

We computed the results without running clustering for noise removal in the end. Since the clustering algorithms are in principle 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 for the datasets sim, lab, carpark and lecturehall.

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

[29] The F_1 score is the harmonic mean of precision p and recall r: $F_1 = \frac{2pr}{p+r}$. Precision is a measure of how many of the selected items (true positives plus false positives) are relevant (true positives). Recall is a measure of how many of the relevant items (true positives plus false negatives) are selected (true positives).

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. 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.

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

Table 4.1: Test parameters

dataset	T_a	$T_r(m)$	#cmp	voxel size (m)
sim	1.4	0.1	28	0.6
lab	1.2	0.2	66	0.175
carpark	1.0	0.35	6	0.125
lecturehall	0.8	0.3	1	0.1
KITTI	1.3	0.74	128547	0.39

The final test parameters can be seen in table 4.1. Columns T_a and $T_r(m)$ show the best parameters found for the method by Underwood et al. while the last column shows the best parameter found for our method. For the sim, lab, carpark, and lecturehall dataset, all scans were used to pick the best parameters. Due to the size of the KITTI dataset, only a subset of all scenarios was used to find the best parameters. The subset was chosen by first evaluating the KITTI datasets with the parameters from the carpark dataset and then picking the 11 scenarios where the Underwood method performed best. The final choice of KITTI scenarios can be seen in table 4.3. The fourth column shows the number of comparisons for each dataset. For the sim, lab, carpark and lecturehall dataset, that number is equal to $\frac{N(N-1)}{2}$[30] with N being the total number of scans in the dataset. If the same strategy would've been used for the KITTI dataset, then a total number of 6125127 comparisons would be needed. Since that would require more than a month of computation time on our hardware, we evaluated different strategies. In the chosen strategy, 10 random different scans are picked from all scans with a position that lies within 10 m of the current scan. That strategy produced better F_1 scores than other approaches like picking 5 scans immediately before and after the current scan. A possible explanation for this effect is, that the latter approach does not account for situations in which the vehicle is stationary, because picking scans from different vantage points improves the result.

[30] the number of edges in a complete graph i.e. comparing each scan with all other scans

Table 4.2: Test results for sim, lab, carpark and lecturehall

dataset	Underwood	3DTK
sim	**0.98**	**0.98**
lab	**0.71**	0.42
carpark	0.78	**0.83**
lecturehall	**0.96**	**0.96**

The results for the sim, lab, carpark and lecturehall datasets are shown in Table 4.2. 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.

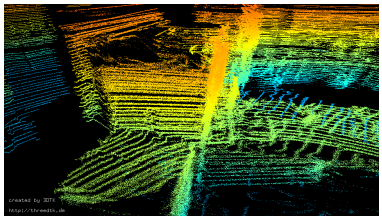


Figure 4.34: Dataset lab showing noise

Our method is outperformed in the "lab" dataset. The dataset is challenging because of its very noisy nature (see Figure 4.34) 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.

Table 4.3: Test results for 11 scenes from the KITTI dataset for which the Underwood method was optimized

#	Underwood	3DTK
9	0.3005	**0.3658**
11	0.4385	**0.5544**
13	0.2610	**0.5684**
15	0.4906	**0.6331**
17	**0.7122**	0.6285
18	0.4409	**0.4975**
39	**0.2994**	0.2739
46	0.2373	**0.6562**
48	0.2051	**0.5656**
51	0.2161	**0.6529**
59	0.4090	**0.4449**
overall	0.3269	**0.5290**

Table 4.3 show the results both methods achieve for the KITTI

dataset. The first column shows the scenes from the KITTI dataset that achieved the highest F_1 scores with the parameters from the carpark dataset. The parameters were then optimized to achieve an optimal overall score which can be found in the last line. Since we only used a subset of the whole dataset to optimize the parameters, this table represents the final result for this analysis. For reference, the result for all KITTI scenes with the chosen parameters is listed in table 4.4.

Both, the Underwood method and our method fare considerably worse on the KITTI dataset than on the other datasets. Numerous reasons are responsible for this. First and foremost, on average, only 1.2% of the points were marked as dynamic. The lower the relative amount of true positives, the harder it is to get good F_1 scores because only few false positives considerably lower the score. Furthermore, the dataset itself is not cleaned of reflections.

Figure 4.35 shows "impossible" points situated under the street surface which are a result of various reflections in the scene. Common sources for these reflections are parked cars and window fronts besides the street. Such points also exist in other locations but those below the street are easiest to visualize to demonstrate this problem. Since the algorithm assumes that the line-of-sight between the laser range finder and all points is "empty", reflected points will introduce false positives.

Another problem source are the masks from the FuseMODNet project themselves. The masks often misclassify points as can be seen in Figure 4.36. The figure shows the left camera frame number 385 from the KITTI scene 9 and is overlayed with the corresponding mask from the FuseMODNet project. Pixels belonging to dynamic objects are marked white. On the left-hand-side of the camera image, one can see that large parts of the traffic sign were marked as dynamic even though the traffic sign is static. In the center of the image, it can be seen, that the front of the car was not marked as dynamic even though it belongs to a dynamic object. Thus, imperfections of the underlying ground truth introduce false positives as well as false negatives into our results.

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 Figure 4.14 and 4.15. 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

Table 4.4: F_1 scores for all KITTI scenes

#	Underwood	3DTK
1	0	0
2	0.0019	**0.0047**
5	0.2633	**0.4279**
9	0.3005	**0.3658**
11	0.4385	**0.5544**
13	0.2610	**0.5684**
14	0.1311	**0.2582**
15	0.4906	**0.6331**
17	**0.7122**	0.6285
18	0.4409	**0.4975**
19	0.2287	**0.4534**
20	0.0055	**0.0281**
22	0.1035	**0.2295**
23	0.0004	**0.0005**
27	0.3008	**0.4856**
28	0.0373	**0.4312**
29	0.1426	**0.3943**
32	0.0270	**0.4960**
35	0	**0.0021**
36	0.0879	**0.4845**
39	**0.2994**	0.2739
46	0.2373	**0.6562**
48	0.2051	**0.5656**
51	0.2161	**0.6529**
52	0	0
56	0.1141	**0.4341**
57	0.2125	**0.2708**
59	0.4090	**0.4449**
60	0.1634	**0.3380**
61	0	0
64	0.0229	**0.0520**
70	0.0462	**0.3831**
79	0	0
84	0.1249	**0.2260**
86	0	0
87	0	0
91	0	**0.0005**
93	0	0

Figure 4.35: Points from reflections under the street surface

Figure 4.36: Examples of wrong classifications of binary masks from FuseMODNet from KITTI scene 9, frame 385.

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 labelling of a training dataset.

4.10.2 F_1 score by voxel size

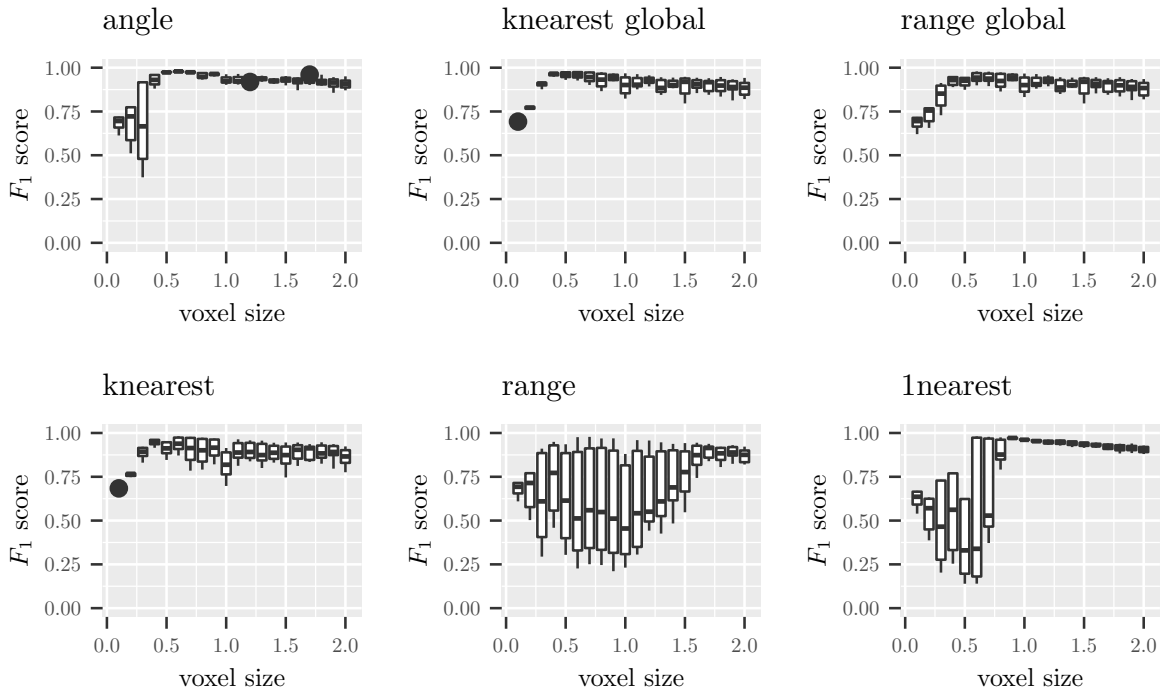


Figure 4.37: F_1 score per voxel size for different methods to acquire the points for normal computation in the sim dataset. Voxel sizes differ by 0.1 from each other and measurements are connected with a line for visual clarity.

The only variable of our algorithm is the voxel size. We display the dependency of the F_1 score on the voxel size in Figure 4.37 using the "sim" dataset as an example. For each voxel size, we shifted the dataset by eight different equally spaced values from zero to the voxel size. This is done because the F_1 score for the "sim" dataset heavily relies on how the "floor" with the boxes on it alignes relative to the voxel grid. 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 4.6 uses all points seen under a certain angle for normal computation we call that method "angle" in Figure 4.37. This method consistently achieved the best quantitative results on all datasets we tested our method on. Additionally, it is also the 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 4.6 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.

Figure 4.38 shows the influence of the voxel size on the KITTI dataset. The graph shows that acceptable results are produced for a relatively large range of voxel sizes between 25 and 50 cm.

4.10.3 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 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 coor-

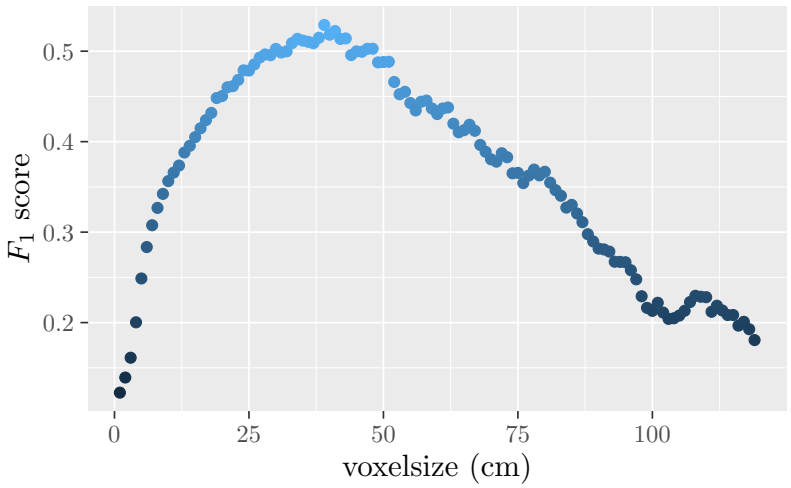


Figure 4.38: Overall F_1 score for the KITTI dataset with different voxel sizes

dinate 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 for $10 \times 10 \times 10 = 1000$ permutations. We do not check beyond 45 degrees as the results are symmetric due to the orthogonal nature of the voxel grid and the resulting symmetry.

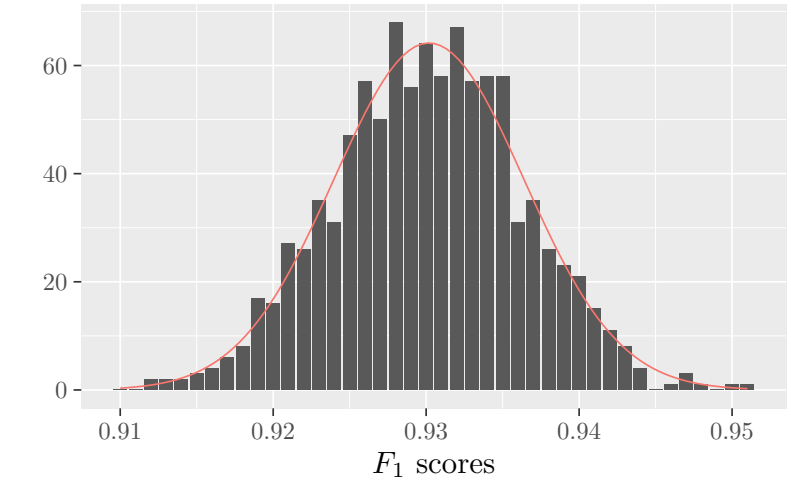


Figure 4.39: Histogram of F_1 scores for 1000 permutations of rotations of the input data around all three coordinate axes. The x-axis shows the F_1 scores. The y-axis shows the number of values falling into bins of 0.001 in width. A gaussian is fitted through the measurements.

We visualize our results using the histogram seen in Figure 4.39. 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.

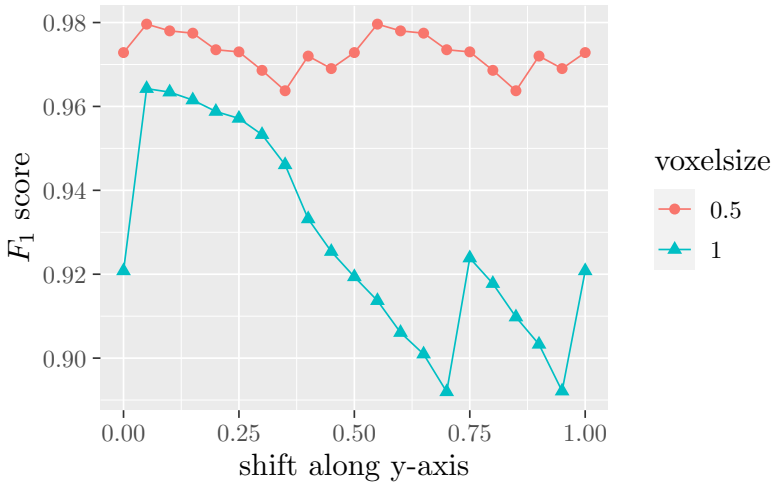


Figure 4.40: F_1 scores achieved by translating the input along the axis perpendicular to the plane on which the moving cubes are placed. The x-axis shows the offset along the axis. The y-axis the achieved F_1 score. Measurements are connected with a line for visual clarity.

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 Figure 4.40. 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" for all displayed voxel sizes as the value achieved for an offset of 0 are equal to the ones achieved for an offset of 1.0.

4.10.4 Qualitative Assessment

Table 4.5: Overview of the datasets used for qualitative assessment

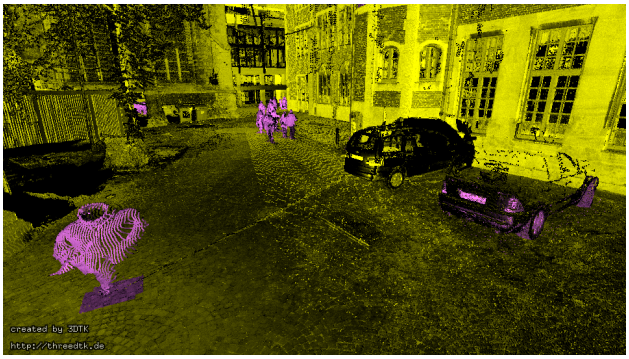
name	#points	#scans	normals(%)	t(s)
Bremen city	215652387	13	0.222	2939
Würzburg city	86585411	6	0.21	4967
Randersacker	194754633	11	0.010	1344

For qualitative analysis we are using the datasets Bremen city, Würzburg city and Randersacker that are shown in table 4.5. 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 4.6, the computations have to be carried out for only a very small fraction of all input points.

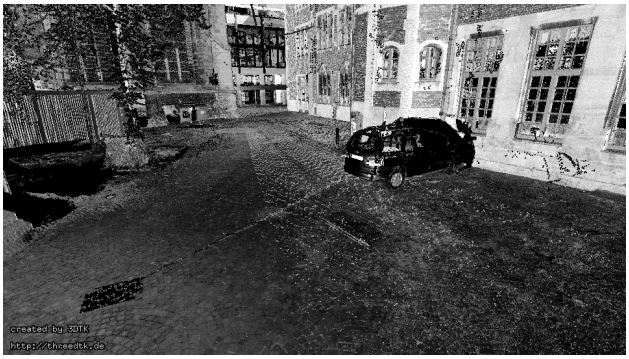
In contrasts to the datasets we used for quantitative analysis, these datasets do not come with any ground truth labeling of points, classifying them whether they are indeed static or dynamic. Thus, without being able to identify false positives and false negatives, F_1 scores cannot be computed.

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. All datasets were registered using slam6D from *3DTK – The 3D Toolkit*[31].

[31] http://threedtk.de

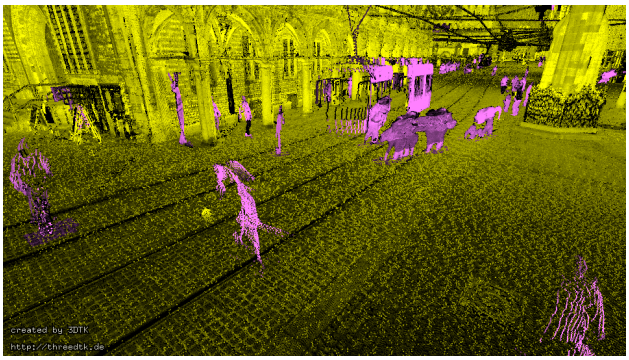


(a) static and dynamic



(b) cleaned

Figure 4.41: Bremen scene 1

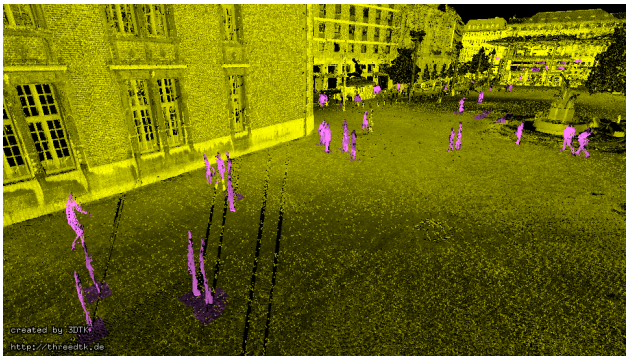


(a) static and dynamic

(b) cleaned

Figure 4.42: Bremen scene 2

Figures 4.41 until Figure 4.46 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.

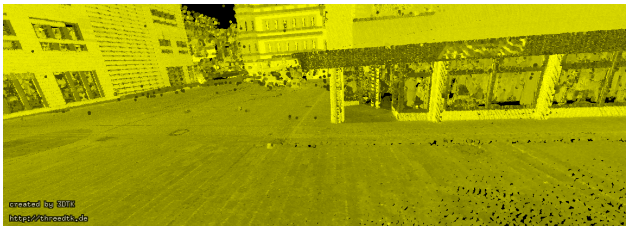


(a) static and dynamic

(b) cleaned

Figure 4.43: Bremen scene 3

(a) static and dynamic



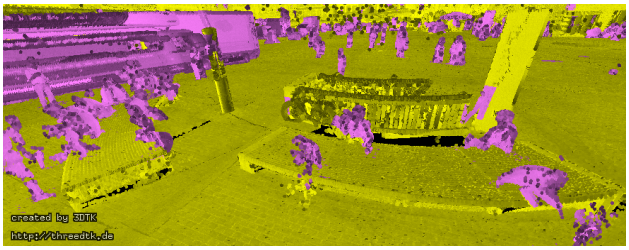
(b) cleaned

Figure 4.44: Würzburg scene 1

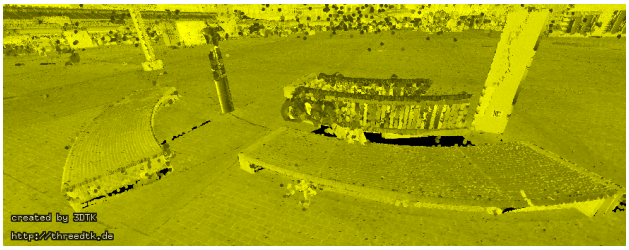
(a) static and dynamic

(b) cleaned

Figure 4.45: Würzburg scene 2

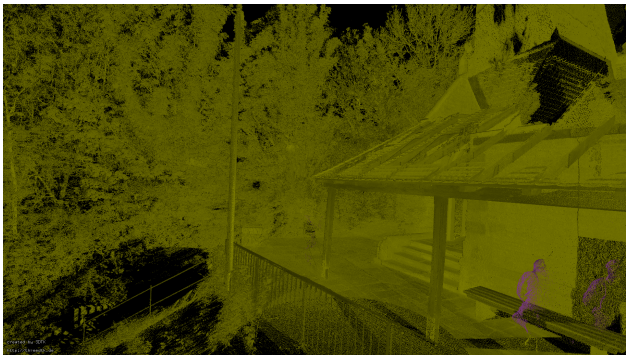


(a) static and dynamic

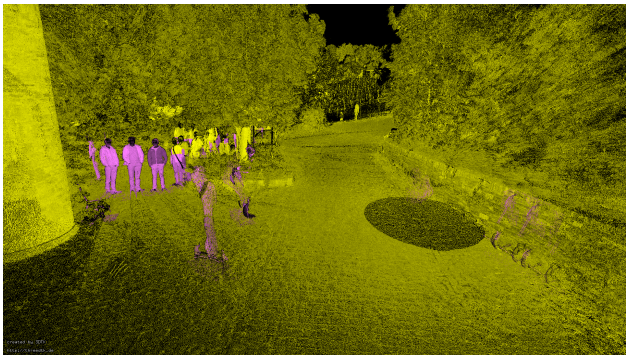


(b) cleaned

Figure 4.46: Würzburg scene 3



(a) Randersacker Dataset with trees



(b) Randersacker Dataset with segways

Figure 4.47: Randersacker dataset

The results from the "Randersacker" dataset are shown in Figure 4.47. The dataset contains a lot of foliage but despite not offering a clear surface, the foliage is not removed. 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.

4.10.5 Performance

To find the dependency of the algorithm runtime from the number of input points. We randomly sampled the first scan of the "lecture-hall" 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. Figure 4.48 shows the number of points the algorithm is able to process per second and as was already shown in Figure 4.33 indicates 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.

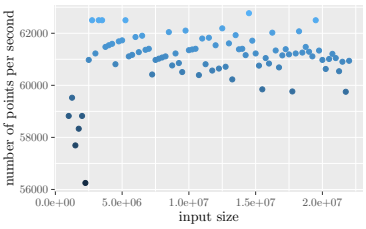


Figure 4.48: The x-axis shows the number of points passed to the algorithm. The y-axis shows the number of points that the algorithm is able to process per second.

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.

Table 4.6: Test parameters

dataset	T_a	$T_r(m)$	#cmp	voxel size (m)
sim	1.4	0.1	28	0.6
lab	1.2	0.2	66	0.175
carpark	1.0	0.35	6	0.125
lecturehall	0.8	0.3	1	0.1
campus	0.8	0.3	3456	0.1
würzburg	0.8	0.3	15	0.1
bremen	n.a.	n.a.	n.a.	0.1
randersacker	n.a.	n.a.	n.a.	0.1

Table 4.6 gives an overview of the parameters that were used for the benchmarks. The third 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. Overlap was determined by computing which scans measured points in the same voxels.

Table 4.7: Runtimes of our method versus the method by Underwood et al.

dataset	$t(s)$	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 hours**
würzburg	7961	0.21	**4967**
bremen	n.a.	0.222	2939
randersacker	n.a.	0.010	1344

The runtime measurements shown in Table 4.7 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 upfront. This conversion step was not part of the benchmark. 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 one to directly compare one run of the Underwood et al. method with one run of our approach. As listed in table 4.7, 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 with many scans, we used the "campus" dataset. 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 data structure that was already generated by previous steps 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.

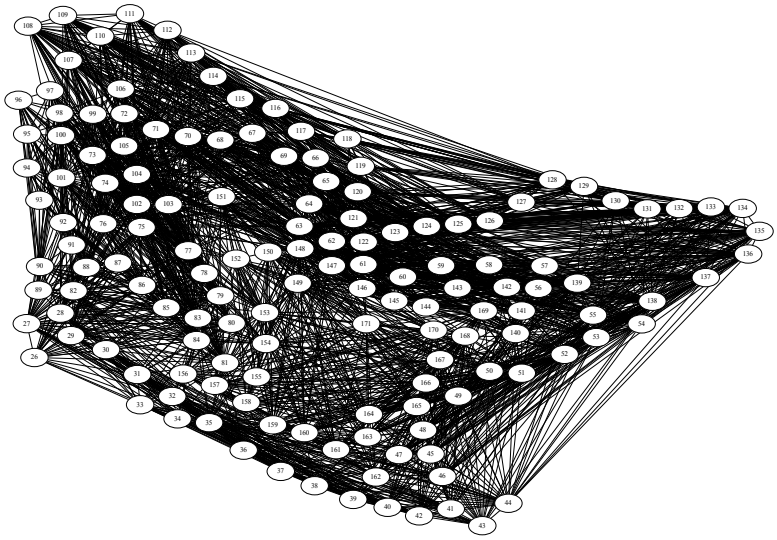


Figure 4.49: Graph of the 146 scans from the "campus" dataset with edges connecting the scans with more than 1000 voxel overlap.

This leaves 3456 scan pairs to compare. Figure 4.49 visualizes the scan pairs as edges in a graph. Nodes are positioned to roughly correspond with their scan coordinates. One can see that due to a lot of open spaces, even scans that were taken far away from each other show sufficient overlap. Since the "campus" dataset does not contain any labels of dynamic objects, we re-used the parameters that worked best for the "lecturehall" dataset. The results shown for the "campus" dataset in Table 4.7 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 4.7, 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 seconds. As discussed in section 4.9 even greater speedups while keeping a high F_1 score are possible when only traversing the lines of sight for a subset of all measured points by up to two orders of magnitude for the "lecturehall" dataset.

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.

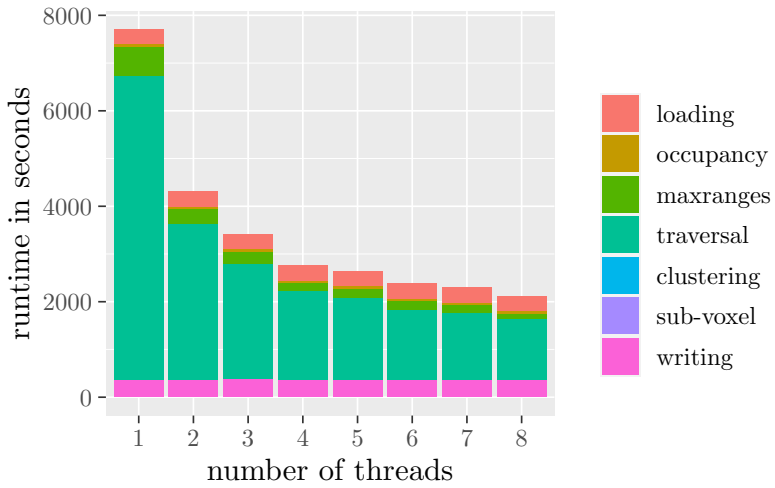


Figure 4.50: Runtime of our algorithm in seconds (y-axis) depending on the used number of threads (x-axis)

The results are shown in Figure 4.50. The phases of the algo-

rithm for which runtimes have separately been timed coincide with the enumeration from section 4.3. 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 data structures 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 seconds on the given dataset.

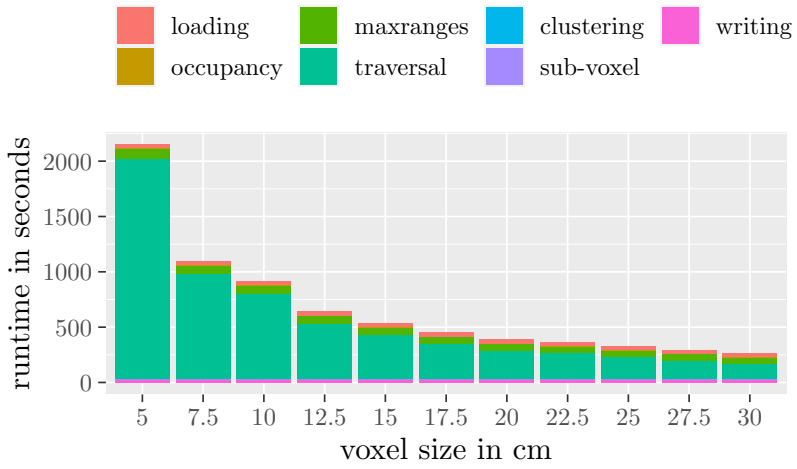


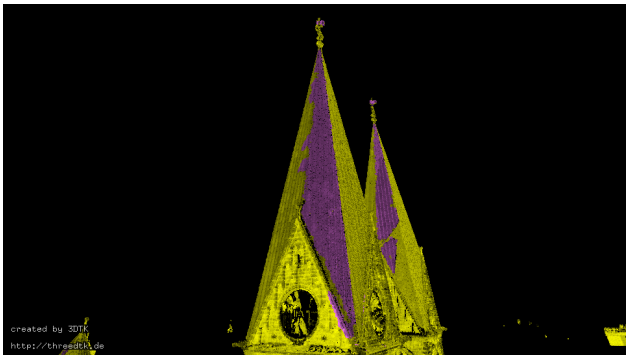
Figure 4.51: Runtime of our algorithm in seconds (y-axis) depending on the voxel size (x-axis)

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 Figure 4.51. 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.

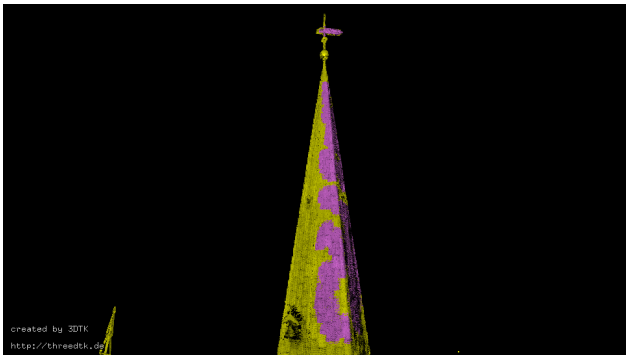
Lastly, we also investigated whether our approach can be leveraged for achieving better point cloud registration results. Our idea was, that dynamic objects may have a negative impact on how well two scans can be matched. To evaluate our hypothesis, we used the Würzburg city dataset as it contained the 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°.

4.11 Limitations

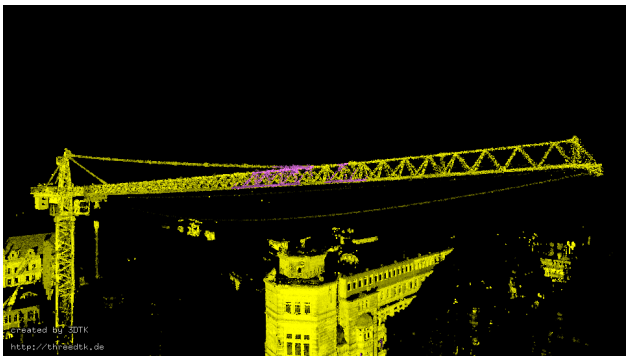


(a) Towers of Bremen Cathedral (St. Petri Dom zu Bremen)

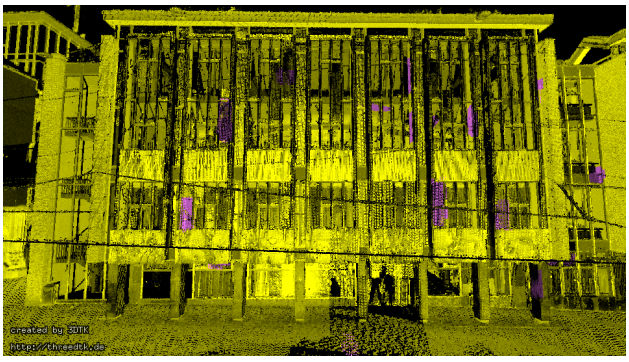


(b) Tower of Church of Our Lady (Kirche Unser Lieben Frauen)

Figure 4.52: Slight registration errors at the church towers lead to incorrectly aligned surfaces. The outer surface is thus marked as dynamic.



(a) False positives due to wrongly computed normal vectors along the boom of a crane

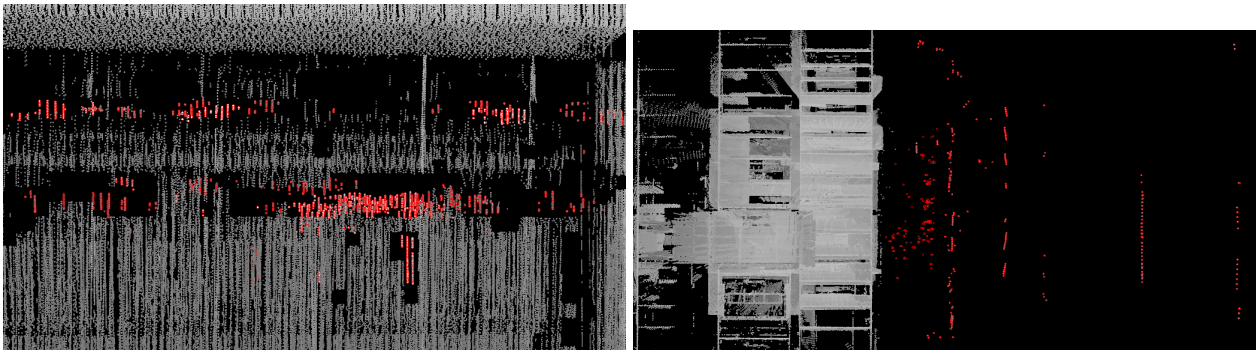


(b) False positives on a facade due to transparent windows

Figure 4.53: Examples for false positives

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 Figure 4.53.
- Solid objects that the laser beam can pass through due to their optical properties like glass will be marked as dynamic because



(a) "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.

(b) Top-view of the scene, showing the same points behind the wall (in red)

Figure 4.54: 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.

they are seen as see-through. An example is shown in Figure 4.53.

- 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 Figure 4.54. Work as by Koch et al.[32] can mitigate this effect.
- If our approach to subvoxel-accuracy is used, some false positives will be introduced as was shown in section 4.8.
- 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 Figure 4.52.

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 4.8.
- 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.

[32] Koch, R., May, S., Murmann, P., and Nüchter, A. (2017). Identification of transparent and specular reflective material in laser scans to discriminate affected measurements for faultless robotic slam. *Robotics and Autonomous Systems*, 87:296–312

4.12 Summary

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. By allowing to optionally work on a reduced set of rays, the runtime can be further improved. Using the concept of "point shadows" false positives that other voxel-based solutions generate are avoided.

5
Collision detection

An important task in civil engineering is the detection of collisions of a 3D model with an environment representation. Existing methods using the structure gauge provide an insufficient measure because the model either rotates or because the trajectory makes tight turns through narrow passages. This is the case in either automotive assembly lines, in narrow train tunnels or in urban environments.

In this part of this thesis we present an algorithm which, given two point clouds, one of the *environment* and one of a *model* and a trajectory with six degrees of freedom along which the model moves through the environment, finds all colliding points of the environment with the model within a certain clearance radius.

We present two collision detection (CD) methods called *kd-CD* and *kd-CD-simple* and two penetration depth (PD) calculation methods called *kd-PD* and *kd-PD-fast*. All four methods are based on searches in a *k*-d tree representation of the environment. The creation of the *k*-d tree, its search methods and other features that we use for our approach to collision detection was described in section 2.1.

The algorithms are benchmarked by moving the point cloud of a train wagon with 2.5 million points along the point cloud of a 1144 *m* long train track through a narrow tunnel with overall 18.92 million points. Points where the wagon collides with the tunnel wall are visually highlighted with their penetration depth. With a safety margin of 5 *cm* kd-PD-simple finds all colliding points on its trajectory which is sampled into 19392 positions in 77 s on a standard desktop machine of 1.6 GHz.

Furthermore, we benchmark our approach against a solution that computes collisions on the GPU. While our approach utilizes the main CPU with a *k*-d tree data structure to efficiently carry out fixed range searches around points in 3D, the other mainly executes on a GPU using a regular grid decomposition technique implemented in the CUDA framework. We will show how massively parallel 3D range searches on a grid based data structure on a GPU performs similarly well as a tree based approach on the CPU with orders of magnitude less parallelization. We also show how each method scales with varying input sizes and how they perform dif-

ferently well depending on the spatial structure of the input data.

5.1 Introduction and problem formulation

The minimum clearance outline or *structure gauge* has an important place in the planning of rail and automotive infrastructure as well as for factory assembly lines[1]. It is the swept volume of the minimum cross section that must be kept free of any obstacles. Measuring the structure gauge of railroad and motorway tunnels, bridges and production lines is a simple way to calculate whether vehicles, their cargo or arbitrary objects can pass through them. The structure gauge is an exact measure as long as the moving object travels along a straight line and does not rotate. But if the trajectory is not straight or rotation is involved, then the structure gauge can only serve as a rough estimation which becomes more imprecise the shorter the turn radius or the larger the rotation of the moving object.[2] Normal railroads and rural motorways usually are constructed with long turn radii and large safety margins, so the structure gauge is a sufficient measure to determine whether a vehicle can pass along a route. But there exist many examples where the structure gauge is an insufficient measure:

[1] Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014b). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5:117–122

[2] Assuming that the moving object is three-dimensional and not a two-dimensional slice

- transportation of exceptionally long, rigid cargo along motorways and railroads
- turns in very narrow tunnels, bridges or other passages
- street turns with a very small turn radius (for example in urban environments)
- rotating objects and sharp turns in tunnels along production lines

The collision detection method presented in this chapter solves this problem but can also be applied to general collision detection tasks. The difference to most other collision detection algorithms is that this method is purely point based and does not require to calculate a solid 3D mesh representation.

This method was first applied by the authors to find collisions in an automotive production line which involved sharp turns and rotations of the car body but the respective paper focuses on the techniques to register the environment[3]. In the following, the same method with some further improvements will be applied to a train moving through a very narrow tunnel where a structure gauge based approach does not suffice to find collisions but where there will be collisions in reality because of the relatively sharp turn the tunnel makes.

[3] Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014b). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5:117–122

A similar measure to the *structure gauge* is the *loading gauge* which is the swept volume of the cross section of a train wagon moved along a track. The difference between the two is the engineering tolerance or *clearance*. The structure gauges along a track together with the maximum loading gauge determine whether or not a train with certain cargo can go along a given route or how

much space around new tracks has to be kept clear and is subject to a number of decades old standards and regulations[4].

If the "track transition curve" at the start and the end of most turns is ignored, then turns of train tracks always represent circle segments (i.e. circular arcs).[5] Since the rotation centers of the two bogies of a train wagon both stay in the exact center between the train tracks, the part of the train wagon connecting the bogies will form the line segment of a secant cutting the circle segment of the track. Thus, the parts of the wagon between the bogies in the inside of the turn will take more space of the structure gauge within a turn compared to when the train wagon travels along straight tracks. Similarly, the parts of the train wagon on both ends outside of the bogies will take additional space as well.

[4] Siegmann, J. (2011). Lichtraumprofil und Fahrzeugbegrenzung im europäischen Schienenverkehr. `http://www.forschungsinformationssystem.de/servlet/is/325031/`. [Online; accessed 2014-07-14]

[5] Lueger, O. (1904). Krümmungsverhältnisse. In *Lexikon der gesamten Technik und ihrer Hilfswissenschaften*, pages 718–724. Stuttgart / Leipzig: DVA

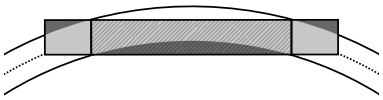


Figure 5.1: Top view of the train wagon

Figure 5.1 visualizes the problem. It shows a top view of the train wagon (in dark and light gray) and its curved loading gauge as it passes through a turn. The dark gray areas mark the volumes of the train wagon outside of its loading gauge. The striped volume indicates the volume of the train wagon between its two bogies. The dotted line indicates the wagon's trajectory. The amount of needed additional space is depending on the turn radius. To address the problem, there exist different regulations for structure gauge sizes depending on the turn radius.[6]

The algorithms that will be presented in the following requires three objects as input: The first input is the pointcloud of the *environment*. In the example presented throughout this part of the thesis we will be using the traintunnel dataset which was described in section 3.4.

The second input is a point cloud of the *model*. Here, it was acquired by taking seven terrestrial 3D scans of a real train wagon with a Riegl VZ-400 laser scanner and then registering them using *3DTK – The 3D Toolkit*[7]. The third input is the *trajectory* of the train tracks.

The goal is to determine which points of the environment collide with the model on its path, given a certain safety margin (the minimal allowed clearance) and how deep any colliding points of the environment penetrate the model. To this end a k-d tree of the environment is created, the model is moved through it along its trajectory and a k-d tree search is performed around the points of the model to find colliding points and their penetration distance.

The problem is highly parallelizable, as all points can be treated in parallel. The later part of this chapter compares algorithms using a search tree, namely a k-d tree, running on a CPU with OpenMP with a GPU implementation that exploits a regular grid decomposition.

[6] EBO (1967). Eisenbahn-Bau- und Betriebsordnung. `http://www.gesetze-im-internet.de/ebo/anlage_1_67.html`. [Online; accessed 2014-07-14]

[7] Nüchter, A., Elseberg, J., Schneider, P., and Paulus, D. (2010). Study of parameterizations for the rigid body transformations of the scan registration problem. *Computer Vision and Image Understanding*, 114(8):963 – 980

5.2 Related Work

Collision detection, which is also called interference detection or intersection searching, is a well studied topic in computer graphics because of its importance for dynamic computer animation and virtual reality applications. On the other hand, the work in that field is limited to collision detection between geometric shapes and polygonal meshes whereas most sensor data is acquired as point clouds. While collision detection is also relevant for motion planning in the field of robotics, it is a less studied problem there. Collision detection between point clouds was for example researched by Klein and Zachmann[8] who use the implicit surface created by a point cloud to calculate intersections. Another example is the recent work by Hermann et al.[9] who use voxels to check for spatial occupancy for robot motion planning.

Existing techniques make use of very similar approaches. One method is to apply a spatial hierarchical partitioning of the input geometry using octrees, AABB-trees, BSP-trees or k-d trees. Other solutions apply regular partitioning using voxels. The goal of any partitioning is to be able to quickly search and check only the relevant geometries in the same or neighboring cells. The method presented in this paper will make use of a hierarchical k-d tree for the environment in combination with a regular partitioning of the model into a grid of bounding spheres.

Another method is to use hierarchies of bounding volumes like spheres[10], axis aligned bounding boxes[11], oriented bounding boxes[12] or discrete oriented polytopes[13]. Optimizing the regular grid that was generated for the model into a hierarchical structure will be left for future work.

Collision detection methods can be divided in those for static and deformable objects[14]. While the method presented in this thesis does not easily allow changes in the environment because that does require a recalculation of its k-d tree (for the CPU based method) or the regular grid decomposition (for the GPU based mothod), arbitrary changes in the point cloud of the model are possible without any performance impacts.

Another classification is whether the algorithm easily allows multiple moving objects. Using a brute-force approach such algorithms have a runtime of $O(n^2)$ for n objects because every possible pair of objects is checked for collisions. Modern approaches like the I-COLLIDE system[15] use a "sweep and prune" approach to minimize the amount of necessary checks. Another approach is to dynamically adjust the search tree to account for object movements[16]. The method in this thesis does not handle multiple moving models.

Calculating the penetration depth of one object into another is important to calculate the force of collisions and respond accordingly in virtual reality applications[17]. It is also important for visualization purposes, to differently highlight objects reaching into a safety margin with an indication of how much they violate the

[8] Klein, J. and Zachmann, G. (2004). Point cloud collision detection. In *Computer Graphics Forum*, volume 23, pages 567–576. Wiley Online Library

[9] Hermann, A., Drews, F., Bauer, J., Klemm, S., Roennau, A., and Dillmann, R. (2014b). Unified gpu voxel collision detection for mobile manipulation planning. In *Intelligent Robots and Systems (IROS), 2014*

[10] Hubbard, P. M. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics (TOG)*, 15(3):179–210

[11] Sulaiman, H. A., Othman, M. A., Ismail, M. M., Said, M., Alice, M., Ramlee, A., Misran, M. H., Bade, A., and Abdullah, M. H. (2013). Distance computation using axis aligned bounding box (aabb) parallel distribution of dynamic origin point. In *Emerging Research Areas and 2013 International Conference on Microelectronics, Communications and Renewable Energy (AICERA/ICMiCR), 2013 Annual International Conference on*, pages 1–6. IEEE

[12] Gottschalk, S., Lin, M. C., and Manocha, D. (1996). Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180. ACM

[13] Klosowski, J. T., Held, M., Mitchell, J. S., Sowizral, H., and Zikan, K. (1998). Efficient collision detection using bounding volume hierarchies of k-dops. *Visualization and Computer Graphics, IEEE Transactions on*, 4(1):21–36

[14] Teschner, M., Heidelberger, B., Müller, M., Pomeranets, D., and Gross, M. (2003a). Optimized spatial hashing for collision detection of deformable objects. Technical report, Technical report, Computer Graphics Laboratory, ETH Zurich, Switzerland

[15] Cohen, J. D., Lin, M. C., Manocha, D., and Ponamgi, M. (1995). I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff. ACM

[16] Luque, R. G., Comba, J. L., and Freitas, C. M. (2005). Broad-phase collision detection using semi-adjusting bsp-trees. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 179–186. ACM

constraint. This application was shown in prior work on this topic by the authors of this thesis[18].

GPU enabled collision detection algorithms are mainly used in computer graphics for ray tracing. The algorithms utilize GPUs using shader language programming, OpenCL or Nvidia CUDA. The first GPU ray tracer was using a uniform grid for acceleration and was implemented in shader language[19]. Zhou et al.[20] show an algorithm of constructing *k*-d trees using CUDA enabled GPUs is shown. To cope with large datasets a method for incremental construction of Bounding Volume Hierarchies (BVH) that incrementally constructs a BVH with quality comparable to the best surface area heuristic (SAH) builders was introduced by Bittner et al. [21].

In the context of nearest-neighbor search in 3D point datasets, Qiu et al[22] use *k*-d trees while Bedkowski et al.[23] make use of regular grid decomposition. The latter authors also compare the performance between these two data structures[24].

5.3 *Collision detection*

Our two variants of collision detection are implemented using the *k*-d tree. One variant, called kd-CD-simple, is based on a range search around each point of the model using `FixedRangeSearch` and the other, called kd-CD, is based on a segment search between two subsequent points of the model on its trajectory using `segmentSearch_all`. Refer to section 2.1.6 for an explanation of the functions. In both variants, the model is moved along its trajectory and a range or segment *k*-d tree search with radius r is performed at each position.

When points are found to be colliding, then this information is saved in a separate boolean vector which stores for each point in the environment whether it ever collided with the model on its trajectory or not. The search radius r determines the precision of both algorithms. The smaller the search radius, the more precise the collision detection is. For smaller search radii, the model has to be sampled densely enough to not leave any unoccupied volume. The search radius r is the required "safety distance" between the model and the environment within which no point of the environment must lie. At the end, the collision information from the boolean vector is used to partition the environment into colliding and non-colliding points.

Figures 5.2 and 5.3 show the two collision detection variants in two dimensions with $T = 3$ points on the trajectory and $M = 3$ points of the model. A model consisting of three co-linear points is moved through the environment along a trajectory (dashed line) with three positions (indicated by numbers at the top). The first position of the three points of the model is marked with red dots,

[17] Tzafestas, C. and Coiffet, P. (1996). Real-time collision detection using spherical octrees: virtual reality application. In *Robot and Human Communication, 1996., 5th IEEE International Workshop on*, pages 500–506

[18] Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014b). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5:117–122

[19] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 703–712. ACM

[20] Zhou, K., Hou, Q., Wang, R., and Guo, B. (2008). Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126

[21] Bittner, J., Hapala, M., and Havran, V. (2015). Incremental bvh construction for ray tracing. *Computers & Graphics*, 47:135–144

[22] Qiu, D., May, S., and Nüchter, A. (2009). Gpu-accelerated nearest neighbor search for 3d registration. In *Computer Vision Systems*, pages 194–203. Springer

[23] Bedkowski, J., Maslowski, A., and De Cubber, G. (2012). Real time 3d localization and mapping for usar robotic application. *Industrial Robot: An International Journal*, 39(5):464–474

[24] Bedkowski, J., Majek, K., and Nüchter, A. (2013). General purpose computing on graphics processing units for robotic applications. *Journal of Software Engineering for Robotics*, 4(1):23–33

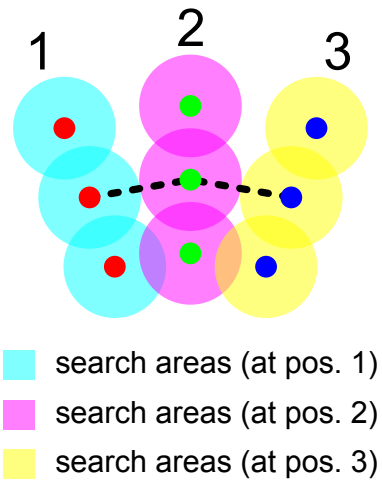


Figure 5.2: kd-cd-simple for a model with three points on three different positions along its trajectory

the second position of the model with green and the third position with blue dots. The area that is searched for collisions with the environment is indicated by the transparent colored areas. For kd-cd-simple, $M \times T = 9$ `FixedRangeSearch` operations have to be carried out. For kd-cd, $M \times (T-1) = 6$ `segmentSearch_all` operations have to be carried out.

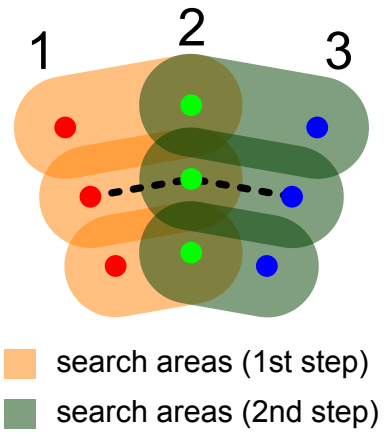


Figure 5.3: kd-cd for a model with three points on three different positions along its trajectory

5.3.1 *kd-CD-simple*

In this variant, on each position of the model on its trajectory, a fixed range search using `FixedRangeSearch` is done around each point of the model. All points of the environment that are found to be within range r of any point of the model at any position on its trajectory are updated to be colliding. The performance of kd-CD-simple is improved by sampling the model in a way such that the search radii around its points overlap in the desired amount.

Figure 5.2 shows a simplified, two-dimensional visualization of the algorithm. A model consisting of three co-linear point is moved along a trajectory with three positions. At each position, a `FixedRangeSearch` is carried out around each point of the model. The figure shows a disadvantage of this approach: if the trajectory is not sampled densely enough, then some volumes along the path will not be checked for collisions as can be seen at the upper points in the graphic.

For a linear, non-parallel execution the time complexity of the algorithm is $O(MT \log n)$ where M is the number of points in the model, T is the number of sampled positions on the trajectory and n the number of points in the environment. For parallel execution, the time complexity is $O(\frac{MT}{p} \log n)$ where p is the number of worker processes. The complexity is as such because M times T searches in the k-d tree of the environment have to be done, where each search is of complexity $O(\log n)$. The complexity in the parallel case highlights that all M times T searches in the k-d tree can be carried out in parallel.

5.3.2 *kd-CD*

Instead of searching a fixed radius around every point of the model at each position on its trajectory like kd-CD-simple, this variant linearly connects the same point of the model at two consecutive positions on its trajectory and searches a fixed radius around all the line segments that are created in this manner.

Figure 5.3 shows a simplified, two-dimensional visualization of the algorithm. The model of three co-linear points is moved along a trajectory with three positions just as for the kd-CD-simple example. But instead of executing a `FixedRangeSearch` around each point of the model, a search is done around the line segments connecting the same point at two consecutive positions on the trajectory. The area that is searched this way is highlighted in orange and dark-green in the figure for the first and second search-pass,

respectively.

This means that with T positions on the trajectory, this method will execute $M(T-1)$ k-d tree searches using `segmentSearch_all`. Thus, the time complexity of this algorithm is very similar to the one of kd-CD-simple $O(M(T-1)\log n)$ and becomes close to the one of kd-CD-simple for large numbers of T.

Since the trajectory can be less densely sampled than would be required for kd-CD-simple, kd-CD can thus require less search operations while maintaining a similar result quality. It also has the advantage that in contrast to the kd-CD-simple, the volumes of the environment that are searched for collisions are not spheres but cylinders with half spheres on both ends. This "smoothes" the found colliding points along the direction of movement of the model.

5.4 *Depth of penetration calculation*

Two variants to calculate depth of penetration will be presented: kd-PD-fast and kd-PD. They perform differently depending on the kind of input data and yield different results depending on the sampling rate of the model trajectory. kd-PD-fast is generally faster but produces only good results for objects protruding the path of the model through the environment. It does not produce correct results when the model moves alongside a wall and collides with it.

kd-PD-fast is an embarrassingly parallel operation just as the collision detection methods. The other variant, kd-PD, is easy to parallelize as well and the only part of kd-PD that has to be synchronized between workers is the updating of the penetration depth because it requires reading and checking the already stored depth of penetration per colliding point.

5.4.1 *kd-PD-fast*

This variant is a good heuristic for protruding sharp objects into the work space. At each position along the trajectory, it iterates through all points of the environment that are found to be colliding and finds the closest non-colliding point using `FindClosest`. The distance between the two points is then recorded as the depth of penetration. Thus, the time complexity of this algorithm is the same as for the collision detection algorithms and can be completely parallelized.

This variant works well for objects that "stick" into the path of the model because the penetration depth of the tip of that object will be about as deep as its distance to the closest non-colliding point. This method is shown to work well for automotive assembly lines as shown in prior work of the authors[25].

[25] Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014b). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5:117–122

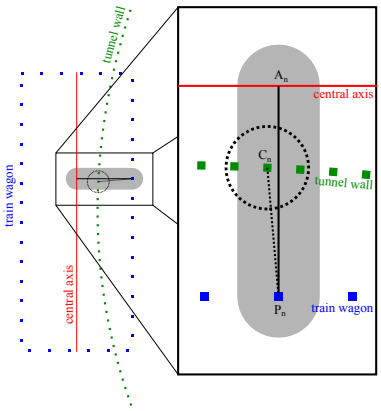


Figure 5.4: top view of the train wagon

5.4.2 *kd-PD*

Figure 5.4 shows on the left a top view of the train wagon (blue) at a position through the tunnel (green). On the right it shows a magnified and rotated part of the left figure with point names. The gray area represents the segment search volume between point P_n of the train wagon and point A_n on the wagon's central axis (red). The dotted black line is the distance between P_n and C_n which is the point that is found to be closest to P_n within the search area. The dotted circle shows the search radius around C_n. All points of the tunnel wall within this radius are updated with the same distance that C_n has to P_n if that distance is greater than the previously stored one.

kd-PD represents a general penetration depth method. Consider Figure 5.4 which illustrates this method. Figure 5.4 shows a top view of the train wagon model at one point of its trajectory inside the tunnel. It is shown colliding with the right hand side tunnel wall. The algorithm iterates over every point of the model P_n and finds its projection to the wagon center A_n. Since the central axis is the y-axis in the coordinate system of the train wagon , this projection is simply done by setting the x and z coordinates to zero. Then a segment search using `segmentSearch_1NearestPoint` on the line segment from P_n to A_n is performed for every point of the model: for each point P_n the closest point C_n of the colliding environment within the search radius is found. A fixed range search using `FixedRangeSearch` of radius r around C_n is performed and all points within that search radius including C_n are collected. This collecting of points has to be performed because otherwise, many points of the environment are missed by `segmentSearch_1NearestPoint`. The distance between C_n and P_n is calculated and that distance is assigned to all points that are found by `FixedRangeSearch` if the new distance value is greater than the old one. This set of calculations is done for each point of the model on each position of its trajectory. In the end, every colliding point of the environment has attached to it the greatest distance found by this method over the whole trajectory. As is seen from Figure 5.4, the maximum error of the calculated penetration distance is the size of the search radius.

Figure 5.5 visualizes how, as the wagon (dark gray) moves along the tunnel (light gray), each point of the tunnel wall is updated with its maximum distance to the wagon exterior (stripes) on any point along the trajectory. The figure shows the calculated distances between each point of the model and each set of points in the colliding environment.

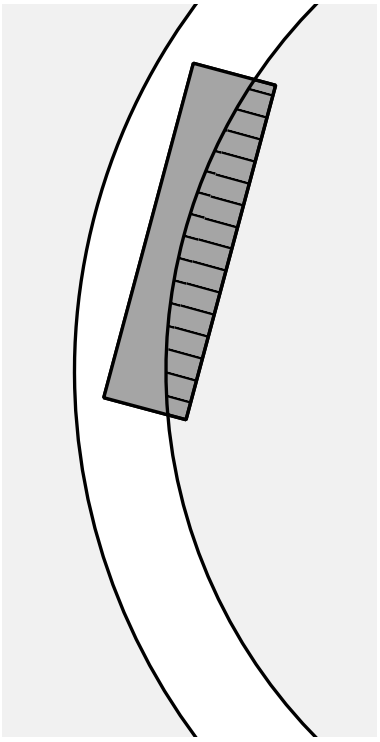


Figure 5.5: Top view of train wagon in tunnel

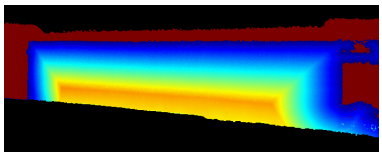


Figure 5.6: Penetration depth as calculated by kd-PD-fast. The colors indicate the distance to the closest non-colliding point of the tunnel wall.

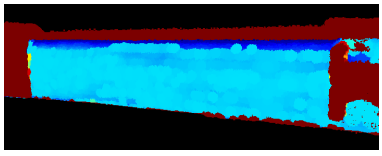


Figure 5.7: Penetration depth as calculated by kd-PD. The colors indicate the maximum penetration depth of the tunnel wall into the moving train wagon on any point of its trajectory.

Figures 5.6 and 5.7 shows a comparison of the penetration depth as calculated by kd-PD-fast (left) and kd-PD (right). Both figures show a narrow piece of tunnel from the outside with the calculated penetration depth indicated by the point color. Non-colliding points are shown in dark red.

This method requires that the individual points of the trajectory are not further apart than the search radius. While this is also one of the reasons why this method is more computationally expensive than the first heuristic, it also yields better results when applied to a collision with the tunnel wall. Figures 5.6 and 5.7 illustrate the difference.

The time complexity in the non-linear case is the same as for kd-PD-fast and for the collision detection algorithms. In parallel execution, some time has to be spent synchronizing the access to the data structure that stores the currently closest penetration distance before updating it.

5.5 Design and Implementation

In the following section we describe the design and implementation of the two methods we present in this chapter. The first method is based on a k-d tree search which runs entirely on the CPU. The second method is based on a regular grid decomposition and runs on the GPU. Both methods take the following inputs:

- a set of points making the environment E
- a set of points making the model M
- a set of 6DOF transformations making a trajectory T and
- a search radius r

Both methods will find all points in the environment E which fall within a certain radius r around any point of the model M at any point on its trajectory T. The CPU method works with double precision (eight bytes) while the GPU based method is limited to floating point precision (four bytes).

5.5.1 3dtk k-d tree

The collision detection method using a k-d tree was described in detail in one of our earlier papers that was published in the Journal of Advanced Engineering Informatics.[26] We are using the method called kd-CD-simple from that publication in these benchmarks. Using the nomenclature for E, M, T and r from above, the basic algorithm is as follows:

$K \leftarrow create\_kd\_tree(E)$
$c \leftarrow [false \forall p \in E]$
for all $t \in T$ **do**
  for all $m \in M$ **do**
    $m' \leftarrow transform(m, t)$
    $s \leftarrow rangesearch(K, m', r)$

[26] Schauer, J. and Nüchter, A. (2015). Collision detection between point clouds using an efficient k-d tree implementation. *Advanced Engineering Informatics*, 29(3):440–458

$update\_colliding(s, c)$

In other words: Create a k-d tree K from the environment and create an array c which stores for each point in the environment whether it collides with the model at any point on the trajectory or not. Then do the following: For each 6 DOF transformation t on the trajectory and for each point m of the model, apply t to m, producing m' and find all points s in K that lie within radius r around m' and update c to set these colliding points to *true*. Since searches in the k-d tree are a read-only operation and since even updating the same point in c from different threads does not lead to any race conditions (because values are set to true irrespective of their former value), the algorithm is embarrassingly parallel. In other words, if one could run $|T| * |M|$ threads in parallel, then the whole algorithm would only take as long as the longest search in the k-d tree would take.

5.5.2 *regular grid decomposition (RGD)*

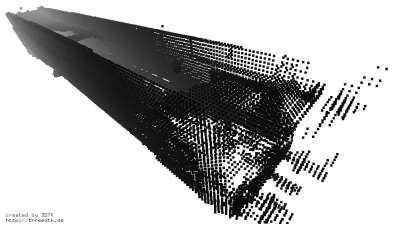
The GPU accelerated implementation of collision detection is based on regular grid decomposition and GPGPU accelerated nearest-neighbor search. The collision detection algorithm, using the nomenclature from above with E as the environment, M as the model, T as the trajectory and r as the search radius is is follows:

$R_e \leftarrow create\_RGD(E)$
$S = \emptyset$
$c \leftarrow [false \forall p \in E]$
for all $t \in T$ **do**
  $M' \leftarrow transform\_in\_parallel(M, t)$
  $S' = find\_corresponding\_cells\_in\_RGD(M', E)$
  if $S' \neq S$ **then**
    $sync\_data\_with\_GPU\_memory(S')$
    $R'_e \leftarrow create\_RGD(S')$
    $S = S'$
  for all $m \in M$ **do** ▷ In parallel
    $s \leftarrow rangesearch(R'_e, m', r)$
    $update\_colliding(s, c)$

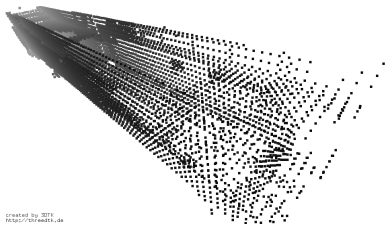
The core concept of the algorithm is to use regular grid decomposition R_e to split large environment point cloud E into smaller cells and then use only the cells which intersect with the bounding box of the transformed model for collision detection at each point of the trajectory T. For each point of the trajectory, all points of the model M are transformed (in parallel) using t, producing M'. The axis aligned bounding box of M' is compared to R_e to find all cells S' containing possibly colliding points, ie. all points in S' must be more than r away from the boundary of the axis aligned bounding box defined by S'. If there are new cells in S' compared to S, or some are no longer relevant, points are copied to or removed from GPU memory respectively. If the data in GPU memory was

updated, for all points in S' a regular grid decomposition R'_e is carried out. After the data on the GPU has been made current, for all points m in M the range search is performed with radius r in R'_e. Finally the array c is updated and copied back to host memory. In this approach parallelism is limited by two factors, device memory and number of cores. All transformations are done theoretically in parallel, the serialization process is low level and provided by the device driver and depends on used device.

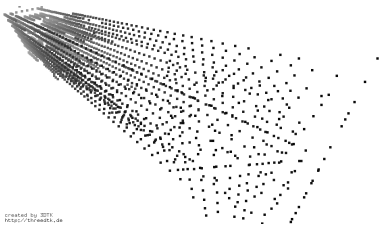
5.6 Experiments and results



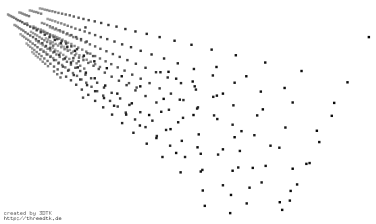
(a) Minimum point distance $d_m = 0.058\ m$ for a search radius of $r = 0.05\ m$ makes a model with 28622 points.



(b) Minimum point distance $d_m = 0.115\ m$ for a search radius of $r = 0.1\ m$ makes a model with 7546 points.



(c) Minimum point distance $d_m = 0.231\ m$ for a search radius of $r = 0.2\ m$ makes a model with 2041 points.



(d) Minimum point distance $d_m = 0.462\ m$ for a search radius of $r = 0.4\ m$ makes a model with 461 points.

(e) Minimum point distance $d_m = 0.924\ m$ for a search radius of $r = 0.8\ m$ makes a model with 93 points.

Figure 5.8: Five point models of the train wagon with different sampling densities. In all reductions, points are aligned in a 3D square lattice.

To benchmark the developed algorithms, the train wagon model as well as the trajectory are sampled with several different point distances. For the train wagon, the original amount of 2.5 million points is reduced using 3DTK's `scan_red` program which allows an octree based reduction of a point cloud with a given voxel size. As the search volume for collision detection must not contain any holes, a model of equidistant points is created by saving the center of each occupied octree voxel as point of the reduced model. This creates a 3D square lattice of points. Five different reductions of the train wagon point cloud are created to run benchmarks on them and are visualized in Figure 5.8. Due to the structure of the

underlying octree, the voxel size d_m is repeatedly halved starting from a maximum voxel size of 0.924 m and down to a voxel size of 5.8 cm. For each of the five reductions, the search radius is chosen to create a bounding sphere of an octree voxel of the respective size. That way, all space occupied by the model is searched for collisions without leaving any holes. This means that the voxel size d_m computes from the bounding sphere and search radius r as $d_m = \frac{2}{3}\sqrt{3}r$. Similarly, the trajectory is sampled such that the individual positions are between 5.8 cm and 14.78 m apart.

Table 5.1: Overview of test setup parameters

r in m	$d_m = d_t = \frac{2}{3}\sqrt{3}r$	#model	#trajectory
0.05	0.058	28622	19392
0.1	0.115	7546	9780
0.2	0.231	2041	4869
0.4	0.462	461	2434
0.8	0.924	93	1217
	1.848		609
	3.695		304
	7.390		152
	14.780		76

Table 5.1 gives an overview of the chosen search radii, the according voxel size and trajectory position distances and the resulting number of points in the model and on the trajectory. The first column shows the choice of collision detection search radius r. The second column shows the resulting distance between the points of the wagon d_m and the points on the trajectory d_t. The third column shows the resulting number of points in the model. The fourth column shows the resulting number of points on the trajectory. The second and fourth column are extended as the results in Figure 5.11 are calculated for higher distance values as well.

The benchmarks omit runtime results that only modify either the amount of points in the model or the amounts of positions in the trajectory. Both collision detections algorithms, kd-CD-simple and kd-CD, scale completely linearly and is completely parallelized by splitting the workload over different sets of points in the model or positions in the environment.

To test the claim that the structure gauge is an insufficient measure, given the provided environment and trajectory, a slice of the train wagon is moved through the tunnel. The slice is created by collapsing the y-coordinate of the train wagon model. The trajectory is created using the method described in section 3.4.3 but assuming a bogie length of zero. This effectively lets the slice travel exactly along the trajectory with the correct orientation perpendicular to the trajectory.

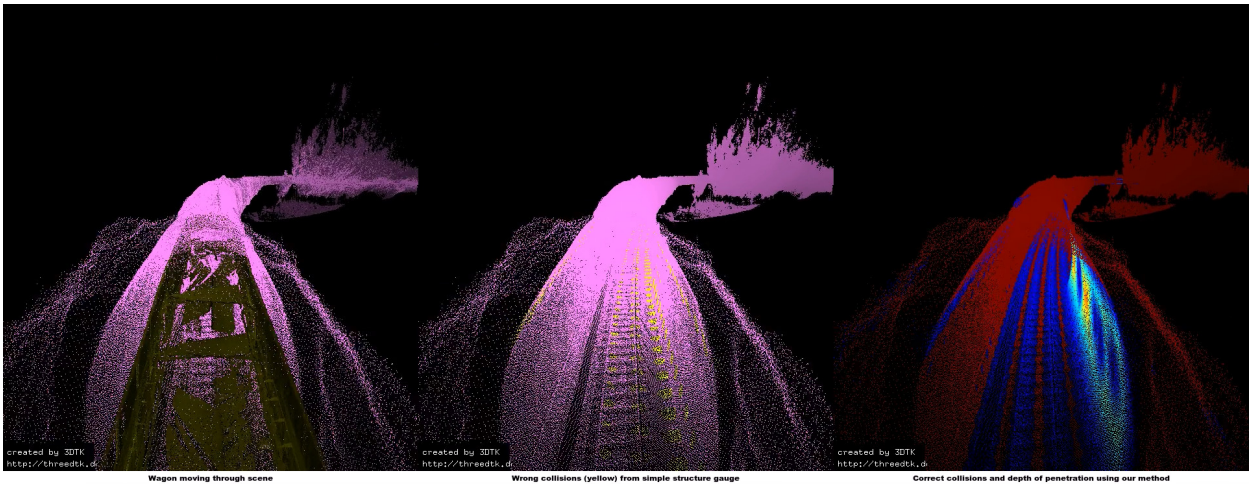


Figure 5.9: A frame from `http://youtu.be/ylp4mD5XZaQ`
[27] `http://youtu.be/ylp4mD5XZaQ`

A video[27] was created to visually illustrate the difference between a structure gauge based method and kd-CD-simple. The video shows the train moving along its trajectory through the tunnel environment from the perspective of an observer who follows closely behind the train wagon. The view is split into three frames arranged next to each other. A single frame from the video is shown in Figure 5.9. The leftmost frame shows the model of the train wagon in yellow moving through the environment in magenta. The center and right frame do not show the train wagon model for better visibility. The center frame shows the colliding points according to the structure gauge method in yellow. The rightmost frame shows the colliding points and their penetration depth as calculated by kd-CD-simple and kd-PD-fast. At multiple points during the video one observes that the center frame does not highlight points as colliding which are highlighted by the rightmost frames. Those points are most often found on the right tunnel wall as the train tracks make a turn to the right. This shows how the structure gauge based method is not able to find some of the collisions that are found by kd-CD-simple.

Figure 5.10 shows the influence of the search radius r on the runtime of both collision detection variants, kd-CD-simple and kd-CD. The distance between individual points on the trajectory d_t and the distance between points in the model d_m is chosen to be $d_t = d_m = 0.231\ m$. While all other variables are kept constant, the algorithm is benchmarked with different search radii. The figure shows the runtime of both collision detection variants as well as the number of points that are found to be colliding in each variant. One can observe that the segment based variant finds more colliding points but that it is also slower than the fixed range search based method. Both variants increase exponentially in runtime with higher search radii. With small radii in the centimeter scale, which is desirable for precise results, the runtime of both variants stays below 10 seconds.

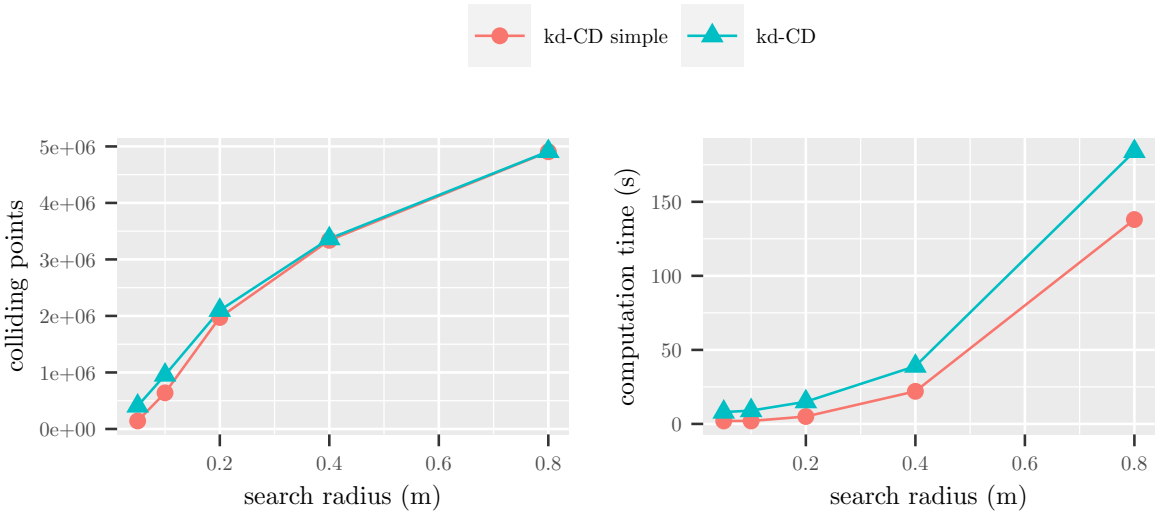


Figure 5.10: Computation time of both collision detection variants

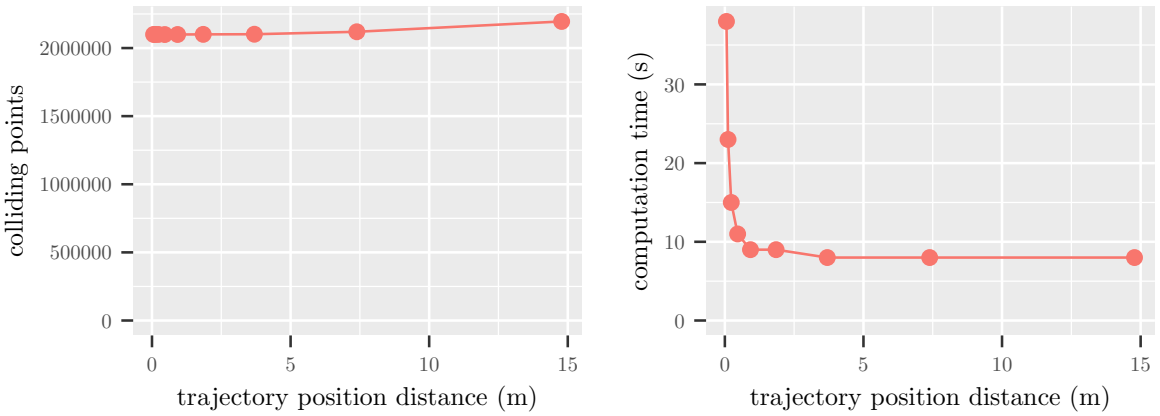


Figure 5.11: Computation time with different distances between points on the trajectory.

In Figure 5.11 the search radius is kept constant and the sampling rate of the trajectory is modified to investigate the dependency of the segment based collision detection method on the segment size. The model was sampled with $d_m = 0.231\ m$ and a search radius of $0.2\ m$. One can observe that as the segment size grows larger, the computation time quickly converges to a constant value of under 10 seconds. The amount of found colliding points slightly increases with larger segment sizes as more colliding points will be found inside the curvature of the tunnel wall.

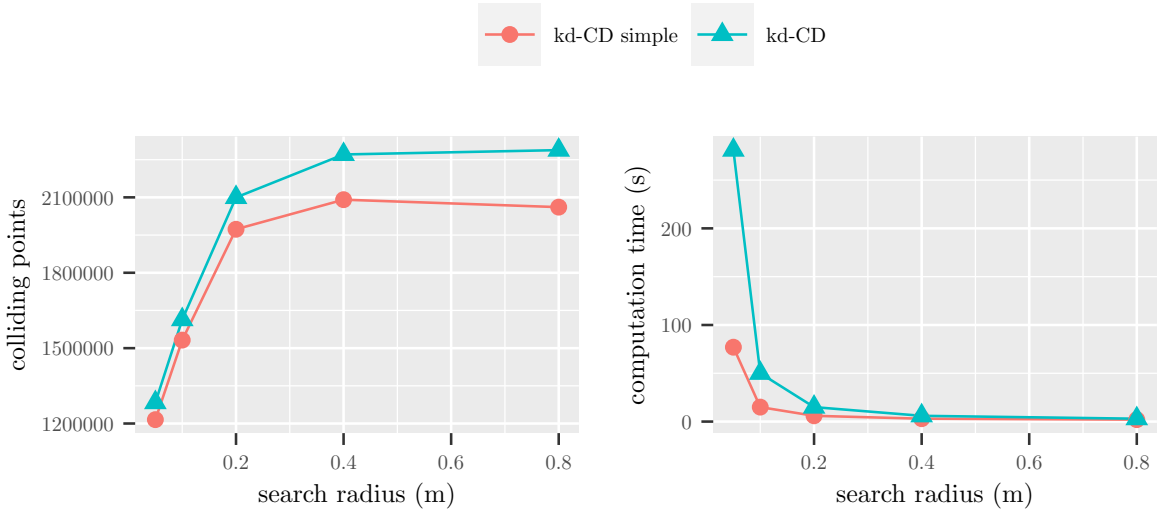


Figure 5.12: Computation time with different search radii and corresponding sampling rates of the model and trajectory.

Figure 5.12 shows a more realistic setup in the sense that not only the search radius is modified like in figure 5.10 but also the sampling rate of the trajectory and train wagon model. If the search radius grows, lower sampling rates are possible because more volume is covered. For each value of search radius the sampling rates have been chosen such that no points of the environment are skipped as the model moves along its trajectory. The distance between individual points on the trajectory d_t and the distance between points in the model d_m is chosen such that $d_t = d_m = \frac{2}{3}\sqrt{3}r$. The graph in Figure 5.12 shows that the both algorithms, kd-CD-simple and kd-CD, quickly approaches runtimes below five seconds as the amount of required k-d tree searches decreases with higher search radii and thus lower sampling rates. On the other hand, the graph also shows, that with the lowest and thus most precise search radius of 5 cm which searches on a trajectory of $19,392$ positions a model of $28,622$ points, our k-d tree is able to make all required $19,392 \times 28,622 = 555,037,824$ k-d tree searches in only 77 s. This means that the average k-d tree search in a dataset of 18.92 mill points takes 139 ns. This in turn means that collision detections of even complex models with up to 287000 points can be done in real

time speed of 25.0 frames per second with the presented k-d search tree implementation.

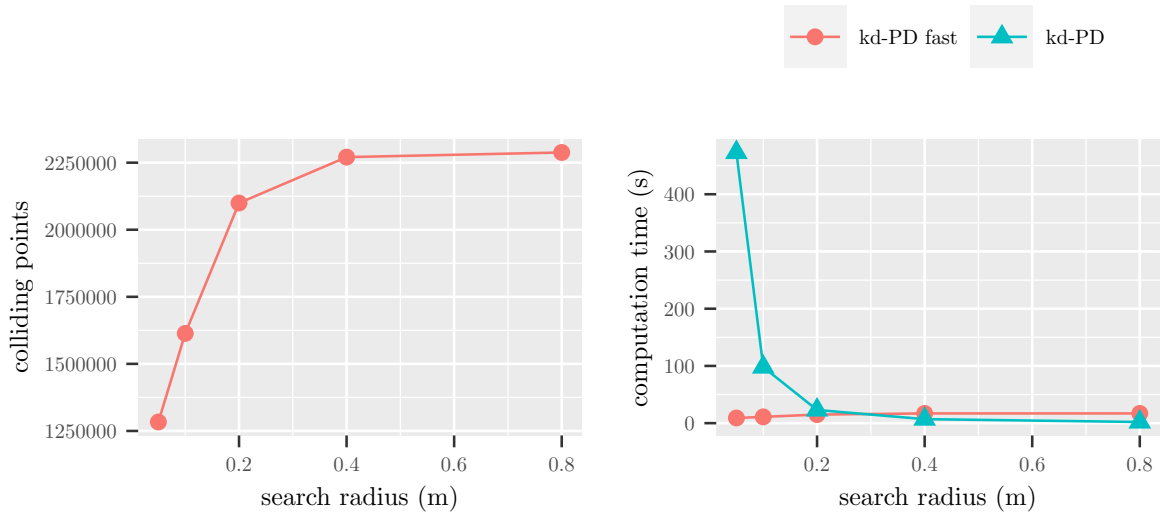


Figure 5.13: Computation time of both penetration depth variants, kd-PD-fast and kd-PD, with different search radii

Figure 5.13 shows the computation time of both penetration depth variants, kd-PD-fast and kd-PD, with different search radii r. The distance between individual points on the trajectory d_t and the distance between points in the model d_m is chosen such that $d_t = d_m = \frac{2}{3}\sqrt{3}r$. Colliding points are computed using kd-CD. One can see that kd-PD-fast stays below 20 s of computation time. This is expected as the performance of kd-PD-fast only depends on the amount of colliding points found. We can observe that kd-PD-fast increases in runtime slightly as the mount of colliding points rises with increased search radius. kd-PD performs badly for very small search radii for which a large number of k-d tree searches have to be performed but quickly approaches runtime values below one minute as the search radius grows larger than 10 cm.

5.6.1 *CPU tests*

The tests of the 3dtk k-d tree implementation have been carried out on two systems. We call the first system "e5-2630 v3" which is a modern desktop system with a 2.4 GHz 8 core processor and 32 GB of RAM. The second test system is dual CPU server system with two 2.8 GHz 10 core CPUs (for a total of 20 cores) and 256 GB of RAM. We call the second system "e5-2680 v2". Both systems are based on Intel Xeon processors and support Hyper-Threading with 16 and 40 threads, respectively. The operating system in both cases was Debian unstable with GCC 5.3.1 and Linux 4.3.5 on the amd64 architecture.

5.6.2 GPU tests

To test the GPU accelerated implementation 3 different Nvidia graphic cards were used. The first GPU is a Geforce Titan X with 3072 CUDA cores (base clock: 1000 MHz, boost: 1075 MHz) and 12 GB GDDR5 384-bit memory. The second one is a Geforce GTX980 with 2048 CUDA cores (base clock: 1126 MHz, boost: 1216 MHz) and 4 GB GDDR5 256-bit memory. The third one is Tesla K40 with 2880 CUDA cores (base clock: 745 MHz, boost: 810/875 MHz) and 12 GB GDDR5 384-bit memory. The GPU performance is tested with the first system "e5-2630 v3".

5.6.3 CPU specific benchmarks

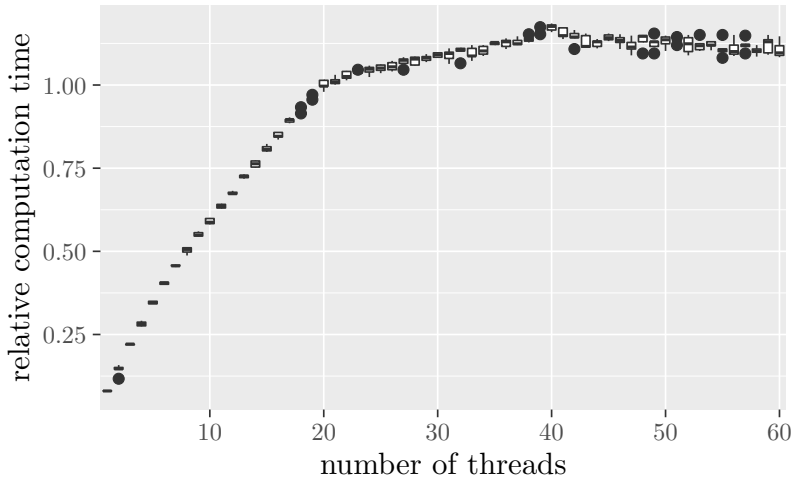


Figure 5.14: Box plot of 3DTK runtime on the Hannover dataset by number of threads

In an earlier publication[28] we claimed that our k-d tree collision detection algorithm would scale completely linearly with increasing number of threads. In Figure 5.14 we show proof of this claim. The figure shows a box-and-whisker plot of 3dtk runtime on the Hannover dataset by number of threads on the "e5-2680 v2" setup (20 cores). The x-axis shows the number of threads from 1 to 60. The y-axis is scaled to show relative runtime compared to using 20 threads. Five benchmarks were carried out for each indicated number of threads. Values indicate the multiple of the runtime per number of threads compared to 20 threads. Higher values mean faster computation. The runtime at 40 threads is close to 1.2 times the runtime with 20 threads. As the test system had 20 individual CPU cores, performance increases with the same slope until that number of threads. After that, performance increases with a less steep slope until 40 threads which can be explained by Intel Hyper-Threading which is able to boost performance by an additional 17.3 % compared to the 20 thread case. No further performance improvement is gained after 40 threads, which lets us conclude that the best CPU utilization is achieved by using the exact same

[28] Schauer, J. and Nüchter, A. (2015). Collision detection between point clouds using an efficient k-d tree implementation. *Advanced Engineering Informatics*, 29(3):440–458

amount of threads as virtual cores are available.

5.6.4 GPU specific benchmarks

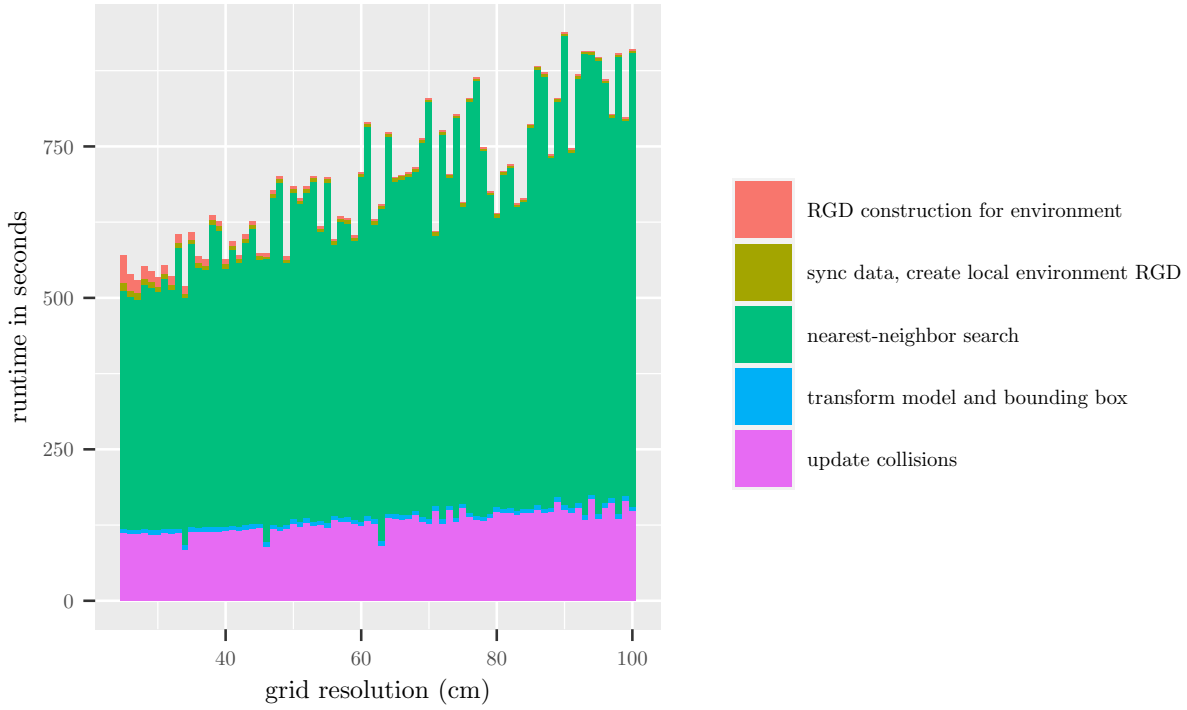


Figure 5.15: Performance of the GPU method by grid resolution

Figure 5.15 shows the performance of the GPU method by grid resolution on the Hannover dataset. The x-axis shows the grid size. The right-hand-side x-axis belongs to the line plot and marks the resulting number of cells. The left-hand-side x-axis belongs to the stacked bar chart and indicates the computation time in seconds for each step of the GPU computation.

For high number of cells the construction time of regular grid is increased, but is still below 5% of the total computation time. The most time is spent during nearest-neighbor search. The computation time increases with the grid cell size but also heavily fluctuates. These fluctuations can be explained by slight variations in the decompositioning resulting in different number of cells being needed on the GPU for collision detection. These fluctuations in run-time are thus deterministic and depend on the input model and trajectory.

5.6.5 CPU versus GPU benchmarks

The shown timings are for the collision search only and do not include the fixed times per dataset that is needed to create the necessary initial data structures like the k-d tree for the CPU based method or the regular grid decomposition of the environment for

the GPU based method. In all experiments we used a search radius of 10 *cm*.

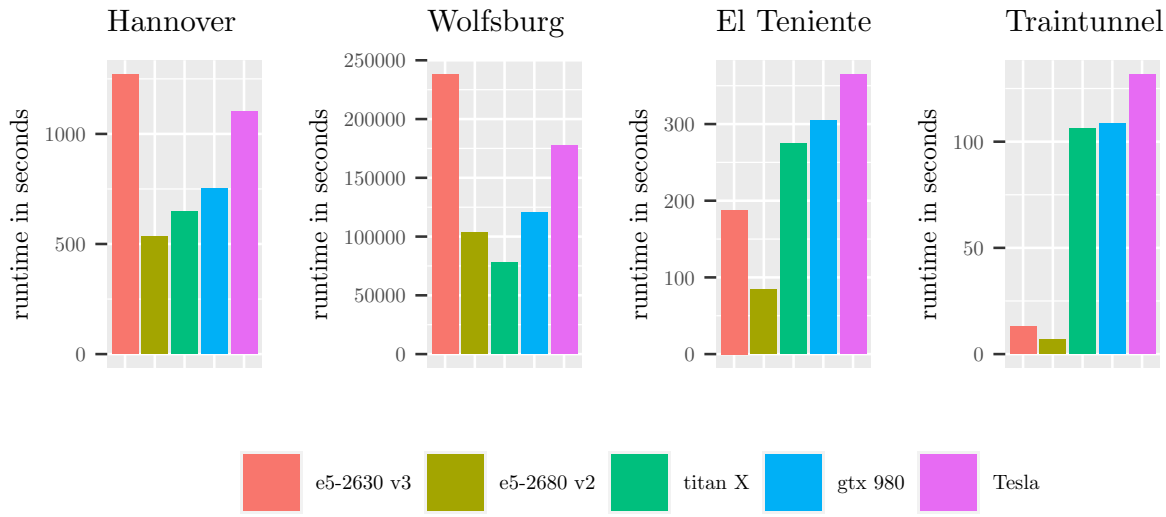


Figure 5.16: Runtime in seconds on each platform for different datasets.

Figure 5.16 shows four graphs comparing the run-time of the CPU and GPU implementations on our four datasets. The CPU approach using the 3dtk *k*-d tree implementation uses as many threads as the respective machines have virtual CPU cores. The GPU approach using regular grid decomposition uses 50 as the grid size. It can be seen that the CPU and GPU based methods perform differently well, depending on the dataset. The GPU based method on the "titan X" platform is the fastest on the "Wolfsburg" dataset but the CPU based "e5-2680 v2" platform vastly outperforms the GPU based methods on the Train Tunnel dataset. We attribute the spatial differences between the datasets to this effect. The "El Teniente" and "Train Tunnel" datasets are similar in that the bounding box of the environment is mostly empty space in both cases, but more so for the "Train Tunnel" dataset which does not contain a loop like the "El Teniente" dataset. Thus, the regular grid decomposition for these datasets will yield a high number of empty cells which will never be queried and the cells with points which end up getting needed comparatively large and overapproximate the actual space to check for collisions. The Hannover and Wolfsburg datasets on the other hand are indoor datasets where the points are more or less evenly distributed over the constructed grid cells.

The last comparison we carried out was to evaluate the two approaches by how they behave depending on the number of points of the environment. For this purpose we created 111 random samplings of the environment point cloud of the "Hannover" dataset in from 500000 points up to 55500000 points in steps of 500000. We then executed our algorithms on the "e5-2680 v2" as well as on the

"titan X" platform for each of the resulting 111 datasets, each with the original model and trajectory.

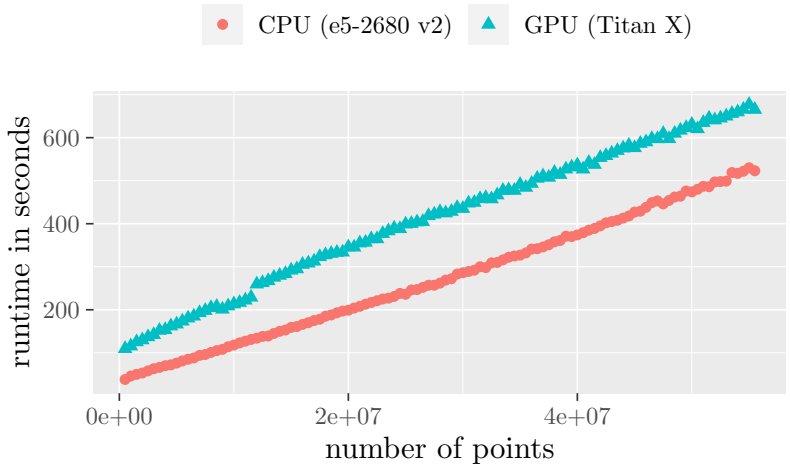


Figure 5.17: Performance with varying numbers of points

The resulting runtimes can be seen in Figure 5.17. The x-axis shows the number of points in the environment while the y-axis shows the runtime in seconds. The graphs indicate a nearly linear behaviour, but we suspect that a very shallow logarithmic function underneath. More research is needed to properly attribute the behaviour seen in the graph.

Table 5.2: Number of colliding points for each dataset

Name	#colliding CPU	#colliding GPU
El Teniente	35225149	35225399
Hannover	2495803	2495804
Wolfsburg	26089196	26089208
Train Tunnel	1627225	1627233

The last two columns of table 5.2 show a difference between the number of colliding points that were found with each method. An analysis showed that both methods produce at least 99.999 % common points given our datasets. The differing points were found to lie on the give search radius. The number of common points is higher for datasets where the input points are given with low precision values. This lets us conclude that the differences stem from floating point errors due to our differing algorithms as well as from the CPU method using double precision while the GPU method uses float precision. Both methods reliably produce the same set of points between different runs and are thus fully deterministic.

5.7 Summary

This chapter presented a highly efficient *k*-d tree implementation which is used to perform collision detection of a sampled arbitrary point cloud against an environment of several million points. It is shown that even though this is a partly brute-force method as it checks all sampled points of the model, both, kd-CD-simple and kd-CD perform well enough such that real queries of densely sampled trajectories are completed in a matter of seconds. Two heuristics for calculating penetration depth, kd-PD-fast and kd-PD have been presented which work for different scenarios and have different precision and runtime properties.

Both variants, kd-CD-simple and kd-CD, are embarrassingly parallel operations. All *k*-d tree searches can be run in parallel and even updating of the boolean collision vector can be done in parallel as its values are only ever written but not read during collision detection. Thus, it should easily be possible to run the algorithm which is currently executed in series, in parallel instead. Verifying the possible performance improvements of this measure is up to further research.

Additionally this chapter has presented a comparison of CPU and GPU implementations for the collision detection problem. With clever implementations, the run time is lowered to an acceptable level for telematics applications.

6
Future Work

Needless to say a lot of work remains to be done. Throughout this work we already pointed out many of the remaining unsolved challenges ahead of us.

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 data structures which are directly designed for fast traversal through an occupancy grid. Examples for such implementations are Octomap[1] as well as our own Octree implementation[2] 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[3] 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[4]. 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.

More work has to be done to research which checks to abort the k-d tree traversal for different search geometries and input data

[1] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206

[2] Elseberg, J., Borrmann, D., and Nüchter, A. (2013). One billion points in the cloud–an octree for efficient processing of 3d laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing*, 76:76–88

[3] Kämpe, V., Sintorn, E., and Assarsson, U. (2013). High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32(4):101

[4] Elseberg, J., Borrmann, D., and Nüchter, A. (2011). Full wave analysis in 3d laser scans for vegetation detection in urban environments. In *Information, Communication and Automation Technologies (ICAT), 2011 XXIII International Symposium on*, pages 1–7. IEEE

perform best. Another easy way to increase the performance could be to change the sampling of the model from bounding spheres to different geometries like axis aligned bounding boxes which are similarly quick to check for collisions. Lastly, instead of checking every point of the model, a hierarchy of bounding spheres or other geometries could be used[5] but that would destroy the property of the current algorithm that the input model is allowed to arbitrarily deform.

While we have now presented flexible CPU and GPU implementations, we further aim at improving run time. To this end, we will look into the issue of regular resampling of the trajectories through B-Spline approximation, support of double precision calculations for the GPU method as well as enhancing the GPU method with more collision detection methods from 3dtk (see [6]). Another useful feature would be a heuristic which is able to pick a good cell size for the regular grid decomposition or a way to work around the limitations of the GPU methods for very sparse environments. Lastly, more experiments are needed to verify the actual dependence of our algorithms on the input pointcloud.

The spherical quadtree data structure could be applied in the context of the work of Houshiar et al.[7] since panorama projections typically fail at the poles while the spherical quadtree does not have such singularities. Similarly in the context of point reduction, our work on compressing point clouds[8] could be extended to use the sphere quadtree as a reduction method without differing point densities towards the poles.

[5] Tzafestas, C. and Coiffet, P. (1996). Real-time collision detection using spherical octrees: virtual reality application. In *Robot and Human Communication, 1996., 5th IEEE International Workshop on*, pages 500–506

[6] Schauer, J. and Nüchter, A. (2015). Collision detection between point clouds using an efficient k-d tree implementation. *Advanced Engineering Informatics*, 29(3):440–458

[7] Houshiar, H., Elseberg, J., Borrmann, D., and Nüchter, A. (2013). Panorama Based Point Cloud Reduction and Registration. In *Proceedings of the 16th IEEE International Conference on Advanced Robotics (ICAR '13)*, pages 1–8, Montevideo, Uruguay

[8] Seufert, M., Kargl, J., Schauer, J., Nüchter, A., and Hoßfeld, T. (2020). Different points of view: Impact of 3d point cloud reduction on qoe of rendered images. In *Proceedings of the Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*

7
Conclusions

The field of 3D point cloud processing poses great demands on algorithms and data structures. Modern laser scanners return point clouds with millions of points per scan. When multiple scans get registered, datasets can easily contain many hundred million points.

In this thesis we showed multiple algorithms to process large spatial datasets using various tree data structures as well as regular grid data structures. We showed that even very large datasets can be processed in a fast and efficient way.

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 achieves similar quality as an existing solution for scan pairs. In terms of runtime our method is superior for the purpose of cleaning scans from dynamic objects as it compares arbitrarily many scans with a linear increase of the runtime.

Additionally, this thesis has presented a comparison of CPU and GPU implementations for the collision detection problem. With clever implementations, the run time is lowered to an acceptable level for telematics applications and compares favourably to massively parallel operations on a GPU.

Lastly, this thesis presented a highly efficient *k*-d tree implementation which is used to perform collision detection of a sampled arbitrary point cloud against an environment of several million points. It is shown that even though this is a partly brute-force method as it checks all sampled points of the model, both, kd-CD-simple and kd-CD perform well enough such that real queries of densely sampled trajectories are completed in a matter of seconds. Two heuristics for calculating penetration depth, kd-PD-fast and kd-PD have been presented which work for different scenarios and have different precision and runtime properties.

Bibliography

Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10.

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

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

Asvadi, A., Premebida, C., Peixoto, P., and Nunes, U. (2016b). 3d lidar-based static and moving obstacle detection in driving environments: An approach based on voxels and multi-region ground planes. *Robotics and Autonomous Systems*, 83:299–311.

Bedkowski, J., Majek, K., and Nüchter, A. (2013). General purpose computing on graphics processing units for robotic applications. *Journal of Software Engineering for Robotics*, 4(1):23–33.

Bedkowski, J., Maslowski, A., and De Cubber, G. (2012). Real time 3d localization and mapping for usar robotic application. *Industrial Robot: An International Journal*, 39(5):464–474.

Bittner, J., Hapala, M., and Havran, V. (2015). Incremental bvh construction for ray tracing. *Computers & Graphics*, 47:135–144.

Blanco-Claraco, J. (2014). Mobile robot programming toolkit (mrpt).

Borrmann, D., Elseberg, J., Lingemann, K., Nüchter, A., and Hertzberg, J. (2008). Globally consistent 3d mapping with scan matching. *Robotics and Autonomous Systems*, 56(2):130–142.

Boute, R. T. (1992). The euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(2):127–144.

Budavári, T., Szalay, A. S., and Fekete, G. (2010). Searchable sky coverage of astronomical observations: Footprints and exposures. *Publications of the Astronomical Society of the Pacific*, 122(897):1375.

Cohen, J. D., Lin, M. C., Manocha, D., and Ponamgi, M. (1995). I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff. ACM.

Curless, B. and Levoy, M. (1996). A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312.

Dierckx, P. (1993). *Curve and surface fitting with splines*. Oxford University Press, Inc.

Drews-Jr, P., Núñez, P., Rocha, R. P., Campos, M., and Dias, J. (2013). Novelty detection and segmentation based on gaussian mixture models: A case study in 3d robotic laser mapping. *Robotics and Autonomous Systems*, 61(12):1696–1709.

EBO (1967). Eisenbahn-Bau- und Betriebsordnung. `http://www.gesetze-im-internet.de/ebo/anlage_1_67.html`. [Online; accessed 2014-07-14].

Elseberg, J., Borrmann, D., and Nüchter, A. (2011). Full wave analysis in 3d laser scans for vegetation detection in urban environments. In *Information, Communication and Automation Technologies (ICAT), 2011 XXIII International Symposium on*, pages 1–7. IEEE.

Elseberg, J., Borrmann, D., and Nüchter, A. (2013). One billion points in the cloud–an octree for efficient processing of 3d laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing*, 76:76–88.

Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014a). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2(5):117.

Elseberg, J., Borrmann, D., Schauer, J., Nüchter, A., Koriath, D., and Rautenberg, U. (2014b). A sensor skid for precise 3d modeling of production lines. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5:117–122.

Elseberg, J., Magnenat, S., Siegwart, R., and Nüchter, A. (2012). Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1):2–12.

Fekete, G. (1990). Rendering and managing spherical data with sphere quadtrees. In *Proceedings of the 1st Conference on Visualization'90*, pages 176–186. IEEE Computer Society Press.

Gálai, B. and 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*, pages 210–220. Springer.

Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*.

Goodchild, M. F. and Shiren, Y. (1992). A hierarchical spatial data structure for global geographic information systems. *CVGIP: Graphical Models and Image Processing*, 54(1):31–44.

Gottschalk, S., Lin, M. C., and Manocha, D. (1996). Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180. ACM.

Guadarrama, S. and Ruiz-Mayor, A. (2010). Approximate robotic mapping from sonar data by modeling perceptions with antonyms. *Information Sciences*, 180(21):4164–4188.

Herbert, M., Caillas, C., Krotkov, E., Kweon, I. S., and Kanade, T. (1989). Terrain mapping for a roving planetary explorer. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 997–1002. IEEE.

Hermann, A., Drews, F., Bauer, J., Klemm, S., Roennau, A., and Dillmann, R. (2014a). Unified gpu voxel collision detection for mobile manipulation planning. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4154–4160. IEEE.

Hermann, A., Drews, F., Bauer, J., Klemm, S., Roennau, A., and Dillmann, R. (2014b). Unified gpu voxel collision detection for mobile manipulation planning. In *Intelligent Robots and Systems (IROS), 2014*.

Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206.

Houshiar, H., Elseberg, J., Borrmann, D., and Nüchter, A. (2013). Panorama Based Point Cloud Reduction and Registration. In *Proceedings of the 16th IEEE International Conference on Advanced Robotics (ICAR '13)*, pages 1–8, Montevideo, Uruguay.

Hubbard, P. M. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics (TOG)*, 15(3):179–210.

Kämpe, V., Sintorn, E., and Assarsson, U. (2013). High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32(4):101.

Klein, J. and Zachmann, G. (2004). Point cloud collision detection. In *Computer Graphics Forum*, volume 23, pages 567–576. Wiley Online Library.

Klosowski, J. T., Held, M., Mitchell, J. S., Sowizral, H., and Zikan, K. (1998). Efficient collision detection using bounding volume hierarchies of k-dops. *Visualization and Computer Graphics, IEEE Transactions on*, 4(1):21–36.

Knuth, D. E. (2011). *The Art of Computer Programming*. Addison-Wesley Professional.

Koch, R., May, S., Murmann, P., and Nüchter, A. (2017). Identification of transparent and specular reflective material in laser scans to discriminate affected measurements for faultless robotic slam. *Robotics and Autonomous Systems*, 87:296–312.

Kruger, J. and Westermann, R. (2003). Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society.

Larsson, T., Akenine-Möller, T., and Lengyel, E. (2007). On faster sphere-box overlap testing. *journal of graphics, gpu, and game tools*, 12(1):3–8.

Leung, K., Lühr, D., Houshiar, H., Inostroza, F., Borrmann, D., Adams, M., Nüchter, A., and Ruiz del Solar, J. (2017). Chilean underground mine dataset. *The International Journal of Robotics Research*, 36(1):16–23.

Liu, K., Boehm, J., and 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*, volume 41, pages 309–313. International Society of Photogrammetry and Remote Sensing (ISPRS).

Lueger, O. (1904). Krümmungsverhältnisse. In *Lexikon der gesamten Technik und ihrer Hilfswissenschaften*, pages 718–724. Stuttgart / Leipzig: DVA.

Luque, R. G., Comba, J. L., and Freitas, C. M. (2005). Broad-phase collision detection using semi-adjusting bsp-trees. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 179–186. ACM.

Magnenat, S. (2014). libnabo.

Moravec, H. and Elfes, A. (1985). High resolution maps from wide angle sonar. In *Proceedings. 1985 IEEE international conference on robotics and automation*, volume 2, pages 116–121. IEEE.

Mount, D. M. and Arya, S. (2010). Ann: a library for approximate nearest neighbor searching, 2005.

Muja, M. and Lowe, D. G. (2012). Flann - fast library for approximate nearest neighbors.

Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohi, P., Shotton, J., Hodges, S., and Fitzgibbon, A. (2011). Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136. IEEE.

Nüchter, A., Elseberg, J., Schneider, P., and Paulus, D. (2010). Study of parameterizations for the rigid body transformations of the scan registration problem. *Computer Vision and Image Understanding*, 114(8):963 – 980.

Nuchter, A., Surmann, H., Lingemann, K., Hertzberg, J., and Thrun, S. (2004). 6d slam with an application in autonomous mine mapping. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 2, pages 1998–2003. IEEE.

Núñez, P., Drews, P., Bandera, A., Rocha, R., Campos, M., and Dias, J. (2010). Change detection in 3d environments based on gaussian mixture model and robust structural matching for autonomous robotic applications. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2633–2638. IEEE.

Oriolo, G., Ulivi, G., and Vendittelli, M. (1997). Fuzzy maps: a new tool for mobile robot perception and planning. *Journal of Robotic Systems*, 14(3):179–197.

Pfaff, P., Triebel, R., and Burgard, W. (2007). An efficient extension to elevation maps for outdoor terrain mapping and loop closing. *The International Journal of Robotics Research*, 26(2):217–230.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 703–712. ACM.

Qin, R., Tian, J., and Reinartz, P. (2016). 3d change detection–approaches and applications. *ISPRS Journal of Photogrammetry and Remote Sensing*, 122:41–56.

Qiu, D., May, S., and Nüchter, A. (2009). Gpu-accelerated nearest neighbor search for 3d registration. In *Computer Vision Systems*, pages 194–203. Springer.

Rashed, H., Ramzy, M., Vaquero, V., El Sallab, A., Sistu, G., and Yogamani, S. (2019). Fusemodnet: Real-time camera and lidar based moving object detection for robust low-light autonomous driving. In *The IEEE International Conference on Computer Vision (ICCV) Workshops*.

Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., and Strasser, W. (2003). Smart hardware-accelerated volume rendering. In *VisSym*, volume 3, pages 231–238. Citeseer.

Ruixu Liu, V. K. A. (2017). 3d indoor scene reconstruction and change detection for robotic sensing and navigation.

Rusu, R. B. and Cousins, S. (2011). 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4. IEEE.

Schauer, J., Bedkowski, J., Majek, K., and Nüchter, A. (2016). Performance comparison between state-of-the-art point-cloud based collision detection approaches on the CPU and GPU. In *Proceedings of the 4th IFAC Symposium on Telematics Applications (TA '13)*, volume 49, pages 54–59, Porto Alegre, Brazil.

Schauer, J. and Nüchter, A. (2014). Efficient point cloud collision detection and analysis in a tunnel environment using kinematic laser scanning and kd tree search. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 40(3):289.

Schauer, J. and Nüchter, A. (2015). Collision detection between point clouds using an efficient k-d tree implementation. *Advanced Engineering Informatics*, 29(3):440–458.

Schauer, J. and Nüchter, A. (2017). Digitizing automotive production lines without interrupting assembly operations through an automatic voxel-based removal of moving objects. In *Control & Automation (ICCA), 2017 13th IEEE International Conference on*, pages 701–706. IEEE.

Schauer, J. and Nüchter, A. (2015). Collision detection between point clouds using an efficient *k*-d tree implementation. *Journal Advanced Engineering Informatics (JAdvEI)*, 29(3):440–458.

Schauer, J. and Nüchter, A. (2018a). Removing non-static objects from 3d laser scan data. *ISPRS Journal of Photogrammetry and Remote Sensing (JPRS)*, 143:15–38.

Schauer, J. and Nüchter, A. (2018b). The Peopleremover — Removing Dynamic Objects From 3-D Point Cloud Data by Traversing a Voxel Occupancy Grid. *IEEE Robotics and Automation Letters (RAL)*, 3(3):1679–1686.

Sellis, T., Roussopoulos, N., and Faloutsos, C. (1987). The r+-tree: A dynamic index for multi-dimensional objects.

Seufert, M., Kargl, J., Schauer, J., Nüchter, A., and Hoßfeld, T. (2020). Different points of view: Impact of 3d point cloud reduction on qoe of rendered images. In *Proceedings of the Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*.

Siam, M., Mahgoub, H., Zahran, M., Yogamani, S., Jagersand, M., and El-Sallab, A. (2017). Modnet: Moving object detection network with motion and appearance for autonomous driving. *arXiv preprint arXiv:1709.04821*.

Siegmann, J. (2011). Lichtraumprofil und Fahrzeugbegrenzung im europäischen Schienenverkehr. `http://www.forschungsinformationssystem.de/servlet/is/325031/`. [Online; accessed 2014-07-14].

Sulaiman, H. A., Othman, M. A., Ismail, M. M., Said, M., Alice, M., Ramlee, A., Misran, M. H., Bade, A., and Abdullah, M. H. (2013). Distance computation using axis aligned bounding box (aabb) parallel distribution of dynamic origin point. In *Emerging Research Areas and 2013 International Conference on Microelectronics, Communications and Renewable Energy (AICERA/ICMiCR), 2013 Annual International Conference on*, pages 1–6. IEEE.

Szalay, A. S., Gray, J., Fekete, G., Kunszt, P. Z., Kukol, P., and Thakar, A. (2007). Indexing the sphere with the hierarchical triangular mesh. *arXiv preprint cs/0701164*.

Teschner, M., Heidelberger, B., Müller, M., Pomeranets, D., and Gross, M. (2003a). Optimized spatial hashing for collision detection of deformable objects. Technical report, Technical report, Computer Graphics Laboratory, ETH Zurich, Switzerland.

Teschner, M., Heidelberger, B., Müller, M., Pomerantes, D., and Gross, M. H. (2003b). Optimized spatial hashing for collision detection of deformable objects. In *VMV*, volume 3, pages 47–54.

Triebel, R., Pfaff, P., and Burgard, W. (2006). Multi-level surface maps for outdoor terrain mapping and loop closing. In *2006 IEEE/RSJ international conference on intelligent robots and systems*, pages 2276–2282. IEEE.

Tufte, E. R. (2006). *Beautiful evidence*. Graphis Press.

Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., and Norman, M. L. (2011). yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement Series*, 192:9.

Tzafestas, C. and Coiffet, P. (1996). Real-time collision detection using spherical octrees: virtual reality application. In *Robot and Human Communication, 1996., 5th IEEE International Workshop on*, pages 500–506.

Underwood, J. P., Gillsjö, D., Bailey, T., and Vlaskine, V. (2013). Explicit 3d change detection using ray-tracing in spherical coordinates. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4735–4741. IEEE.

Vieira, A. W., Drews, P. L., and Campos, M. F. (2014). Spatial density patterns for efficient change detection in 3d environment for autonomous surveillance robots. *IEEE Transactions on Automation Science and Engineering*, 11(3):766–774.

Weinlich, A., Keck, B., Scherl, H., Kowarschik, M., and 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*, volume 1, pages 25–30.

Xiao, W., Vallet, B., Brédif, M., and Paparoditis, N. (2015). Street environment change detection from mobile laser scanning point clouds. *ISPRS Journal of Photogrammetry and Remote Sensing*, 107:38–49.

Xiao, W., Vallet, B., and Paparoditis, N. (2013). Change detection in 3d point clouds acquired by a mobile mapping system. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 1(2):331–336.

Zhou, K., Hou, Q., Wang, R., and Guo, B. (2008). Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126.