Surface Reconstruction from Point Cloud Data using k-d Trees - Project Report

Jean-Yves Verhaeghe and Patrik Rác

Master of High Performance Computing, Sorbonne-Université
Algorithms and Data Structures

January 10, 2023

1 Introduction

Data collection using modern LIDAR technology is becoming more and more common. Hence, more detailed and openly available LIDAR datasets of the earth are getting available, which include in particular point positions. However, most graphics and numerical simulation programs do not operate on point cloud data. Usually, these types of programs require corresponding surface normals. This project implements a program that approximates the surface of a point cloud by computing the surface normal using best plane approximation with neighboring points. We specifically consider the LIDAR data provided by the Swiss government.
In this context, we implemented a k-d tree as an accelerator data structure to implement the nearest neighbor problem. Furthermore, we use the scan angle and flight data included in the LIDAR scans to reconstruct the direction of the aircraft and laser for each point and compare it with the surface normal, allowing us to correctly orient the normal more naturally. Multiple techniques for this process have either been considered or even implemented and will be discussed in this report.

2 Theoretical background

2.1 LIDAR

The project concerns itself with LIDAR (light detection and ranging) data or, more specifically, the resulting `.las` files produced by such LIDAR scans. To successfully create programs that use this data, it is important to understand the basic notions of the technology and file format it is connected with. We will mainly reference our experience obtained while working with the swissSURFACE3D LIDAR data that use the *LAS 1.2* file format in their `.las` files.

LIDAR scanning has allowed the mapping of physical features with extremely high resolution. To that end, especially for the swiss LIDAR surface data, an oscillating laser is fitted to an airplane. The oscillation is possible between $-90°$ and $90°$. In practice, however, the actual maximum angle is far lower.
The data collected by the LIDAR scanner is then processed and stored in the `.las` file format. The file features a *header* section containing relevant information about the scan, including the number of points and the relevant offsets inside the current *LAS* file. This is important as there are multiple separate *Point Record Data formats*, each with different lengths and data fields. Besides, necessary scaling information about the points is stored.
Inside the blocks of the *Point Record Data*, the scaled x, y, and z values of each point and essential flight data corresponding to the state of the scanner are stored. It also contains the *Scan Angle Rank* and *GPS Time* which are relevant for us.
A more detailed description of the file format can be found in the corresponding documentation.

2.2 k-d Tree

The primary data structure that facilitates our implementation is a k-d tree. It is used to store the points and speed up the query of nearest neighbors, which is frequently required by the computation of the surface normals.
A k-d tree is a tree structure that follows a special sorting of k-dimensional points by splitting the space in two hyperplanes successively along each direction.
The operations we are concerned about are the insertion, creation of a balanced k-d tree, and query of k-nearest neighbors.

Insertion

Inserting an element into a k-d tree is similar to inserting an element into a binary tree, with the only exception that the comparison coordinate is dictated by the current depth. Hence, at each level in the k-d tree, we periodically change the values used in the comparison. In the case of a 3-d tree, we cycle between $\{x, y, z\}$.
The complexity of the insertion comes down to $O(\log(n))$ in a balanced k-d tree and $O(n)$ in an unbalanced one.

Balanced k-d tree

Creating a balanced k-d tree is done by adjusting the order in which the elements of a given list L are inserted. By reordering the list, we can make sure to obtain a balanced k-d tree after inserting all elements from L. For this sorting, a divide-and-conquer recursive approach is used that first sorts the list given a comparison plane and then inserts the median. This operation is then recursively performed on the two remaining lists to the left and the right of the median. By altering the sorting plane in the recursive algorithm, we obtain an optimal insertion order for the k-d tree. As balancing

Algorithm 1 Balanced k-d tree

```
1: function BALANCE(L, l, r, p)
2:     Sort L[l : r] according to splitting direction p
3:     Insert L[(l+r)/2] into the k-d tree.
4:     BALANCE(L, l, (l+r)/2 − 1, (p + 1)%dim)
5:     BALANCE(L, (l+r)/2 + 1, r, (p + 1)%dim)
6: end function
```

the tree is an a-priori operation, all elements have to be known beforehand. Additional insertions will not keep the balanced structure of the tree.

k-nearest neighbors

The most important operation of the k-d tree is the nearest-neighbor search, as it is mainly used in the computation of the surface normals. In our context, we consider a k-nearest-neighbor query by successively running the nearest neighbor routine and excluding already found points.
An advantage of the data structure is that one can use the spatial properties of the k-d tree to restrain the search space relatively fast and thus efficiently compute the nearest neighbor.
The algorithm starts by recursively iterating down the tree, similar to the insertion. Once a leaf is reached, we compare the distance from the current node n with the overall minimum distance.
In this fashion, we can now iterate back up in the tree. One more consideration is that the nearest neighbor is not necessarily on the same side of the splitting hyperplane that features our current point. Thus, we perform another check if the point could be on the other side of the current splitting plane by intersecting it with a hypersphere that has the current minimum distance as its radius and continue our search there if applicable.
The given pseudocode is missing some checks if the nodes of the tree actually exist but gives an overview of the general structure of the nearest neighbor search. In our case, we also carry a map

Algorithm 2 Nearest Neighbor

```
1: function NearestNeighbor(node, p, closest, min)
2:     if node is a Leaf then
3:         Compute dist_p and update min and closest
4:     else
5:         if p_split < node_split then                          ▷ Go to the left
6:             NearestNeighbor(node → left, p, closest, min)
7:             Compute dist_p and update min and closest
8:             if p_split + min ≥ node_split then                ▷ Intersect with hypersphere
9:                 NearestNeighbor(node → right, p, closest, min)
10:                Compute dist_p and update min and closest
11:            end if
12:        else                                                  ▷ Go to the right
13:            NearestNeighbor(node → right, p, closest, min)
14:            Compute dist_p and update min and closest
15:            if p_split − min ≤ node_split then                ▷ Intersect with hypersphere
16:                NearestNeighbor(node → left, p, closest, min)
17:                Compute dist_p and update min and closest
18:            end if
19:        end if
20:    end if
21: end function
```

with already found neighbors to form our k-nearest-neighbor algorithm.
The nearest neighbor search algorithm has an average complexity of $O(\log(n))$, given the k-d tree is balanced.
Overall, the k-nearest-neighbor method is by far the most performance-critical in this context, as it is performed for each point in the surface normal computation. Thus, improving this algorithm will improve the overall performance of the code.

2.3 Surface normal computation

Estimating surface normals from a point cloud is mainly done in two stages. First, we compute the normal of the best plane approximation for the neighbors of each point. This gives us the normal direction with two possible orientations. The correct orientation of the normal can then be estimated by using various techniques described later.

We use the k-d tree data structure to store all given points. For each point, we then query a neighborhood of k points. Using these points, we are now able to compute the best plane approximation.
In detail, we use the neighborhood N_k of each point P_i to assemble the covariance matrix

$$C = \frac{1}{k} \sum_{p \in N_k} (pp^T - \bar{p}\bar{p}^T) \tag{1}$$

where

$$\bar{p} = \frac{1}{k} \sum_{p \in N_k} p.$$

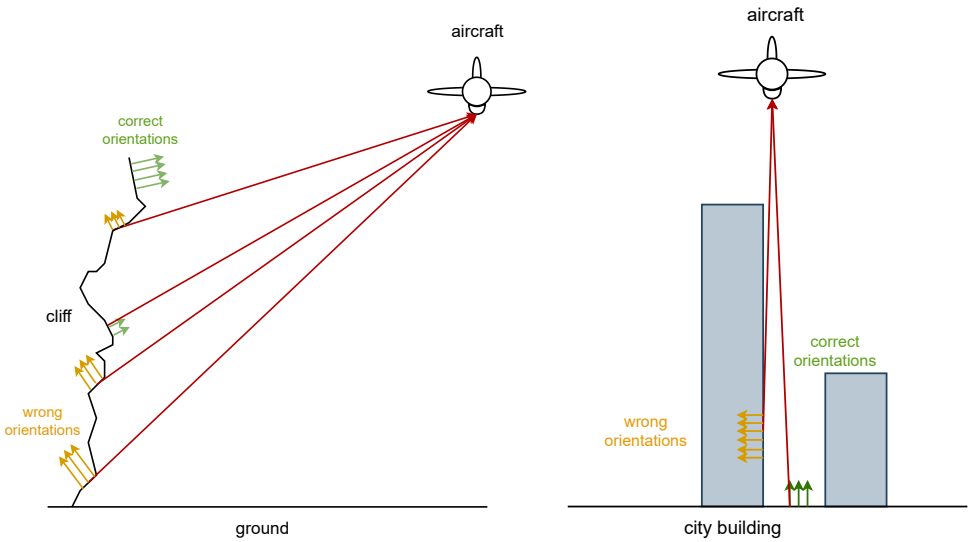
After the assembly of the covariance matrix C using 1, we can now compute the normal to the least square plane, which is just the eigenvector corresponding to the smallest eigenvalues of C. This normal is then used as the unoriented normal at the point P_i.
Mathematically this thus becomes a problem of computing the eigenvalues of a given 3×3 matrix. Multiple numerical techniques can be used in this context. However, the one employed by our code,

or specifically by the LAPACK `dgeev(...)` routine, is a QR iteration algorithm that computes an approximation of all eigenvalues and corresponding eigenvectors of a matrix. After using this algorithm to estimate all eigenvalues and eigenvectors, it remains to scan the array of eigenvalues for the smallest value and extract the corresponding eigenvector.

2.4 Determining the correct orientation of the surface normal

As mentionned already, the method we have just introduced only computes the direction of the normal vector, in other words, the homogeneous equation of the corresponding line in our three-dimensional space. Therefore, it bears, in fact, two possible solutions with opposite orientations when switching the sign for all the factors of $\{x, y, z\}$. For a vector field used to define a surface, the magnitude isn't of interest, and for more practical use, we choose to normalize it, but it is essential to determine the correct orientation to accurately depict a surface.

For the first approach, we chose to naively pick between the two vector possibilities, the one in the positive z orientation, as this is both straightforward and generally the case when we acquire points from a plane above. But in some cases, this would prove to be incorrect, as we can see in the examples from Figure 1. This will also be used as a reference for comparison with other methods.



(a) LIDAR scan of a cliff

(b) LIDAR scan of buildings in a city

Figure 1: Examples of the naively oriented normals in relation to the scan vector

2.4.1 The Scan Angle Rank Method

The Scan Angle Rank method is based on a scalar product between the laser direction and normal vector and is computed through the covariance matrix method. For the vast majority of cases, this scalar product is positive and close to one, which means the computed normal is already in the correct orientation, or it is negative and close to one, and thus we flip it.

To determine the direction of the laser, we can use the data provided in the `.las` file. Indeed, the *Point Record Data* includes the *Scan Angle Rank*, which is the oriented angle (in degrees) of the direction of the laser concerning the vertical vector pointing downwards, also called the *nadir* vector.

The LIDAR scanner has a pendulum movement that goes back and forth orthogonally to the aircraft, in the two-dimension plane containing the source point of the laser, and that is normal to the aircraft fuselage's longitudinal axis. This means the vertical wing-to-wing plane equation and the *Scan Angle Rank* value combined allow us to determine the general direction in the three-dimensional space of the laser. Hence, we need to first determine the aircraft's movement, to compute that plane's cartesian equation and in turn, be able, through some linear algebra, to end up finding out the normalized vector representing that direction.

Aircraft movement

In the collected data, there isn't any set of points representing the aircraft's movement. We can only use the acquired points on the ground to indirectly estimate it. And because there is no data providing the aircraft's yaw and pitch angles, we posit that they are kept constant during acquisition. As a consequence, we can consider that the direction of the airplane's movement is the same as the direction of its longitudinal axis, and we can then use that movement direction as a normal vector to the laser plane, where the nadir vector and the laser direction vector lie.

To compute it, the general idea is to ignore the z component and to track the local offsets in the x and y coordinates of the acquired points over time. We thus need to use the acquisition time of the points and sort them in an array with regard to that time. To make it somewhat robust but still precise, in case of a change in the direction of the aircraft, a good idea is to compute the direction vector using a few consecutive points.

It is important to realize that we assume the aircraft remains constantly level, or more precisely, doesn't ever change its pitch, which is the aeronautical term for the aircraft rotating around a transverse axis going from wing to wing. Indeed, this would make the nose go upwards or downwards and would make the scanner point back and forth relative to the movement of the aircraft's axis, effectively acquiring points that would make our computation look like the aircraft itself is going back and forth.

We're assuming the acquisitions are done in this way, selecting days where there isn't too much wind, with a plane going as steadily as possible, with as little change as possible in speed, direction, and altitude. We also expected that the rolling movement from the plane would be taken into account in the saved *Scan Angle Rank*, which would be corrected with regard to the roll angle rather than simply keeping the raw angle since even minute differences could significantly affect the estimated locations of the points. We verified this is actually the case.

Mathematical computation

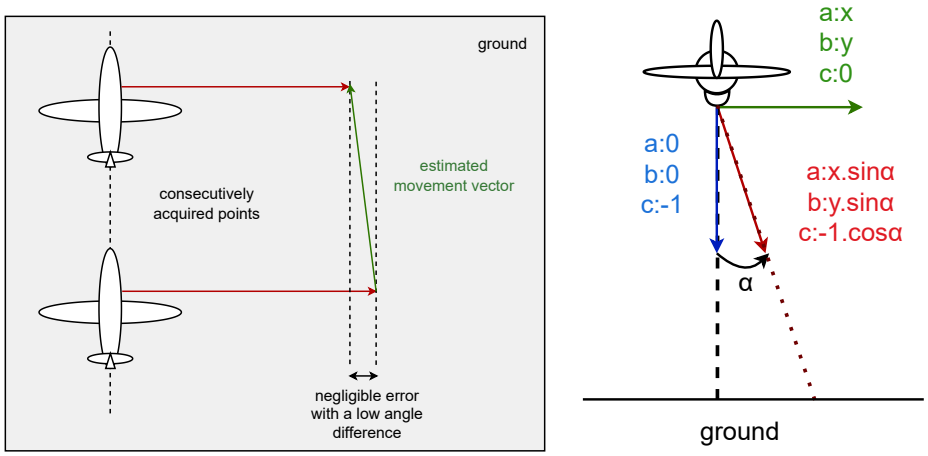
To get the direction of the plane, we use two acquired points that feature in the same scan line. We apply multiple restrictions to determine that these points are actually part of the same scan line. We check that the *Scan Angle Rank* difference of two consecutive points is at most 1 and that there are no significant jumps in GPS time. Furthermore, we check that the small direction vectors between two consecutive points are correctly oriented with regard to the gerenal movement of the plane. Finally, to avoid errors due to high angles and angle differences, which would make the plane look like it goes in a different direction or sideways, we compute the direction vector from the two furthest points with the same minimal *Scan Angle Rank*. We normalize it for the subsequent computations.

Then, to get an orthogonal vector in the horizontal plane, we can simply switch coordinates x and y and switch the sign for one of them. This leaves two possibilities, one pointing to the left side of the aircraft and the other pointing to the right side. We chose the latter since its positive

orientation is aligned with the positive values for the *Scan Angle Rank*, as defined in the *Point Record Data Format 1* from the *LAS* file format, which goes from $+90°$ when pointing horizontally to the right side of the aircraft, to $-90°$ when pointing to the left.

$$\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} -y \\ x \end{bmatrix} \tag{2}$$

We normalize this vector and finally, we use the *Scan Angle Rank* α and find vector v



(a) Diagram of the plane movement vector computation using the scanned points

(b) Laser direction vector in the vertical wing-to-wing plane

Figure 2: Plane movement and laser direction vector calculation

$$v = \begin{bmatrix} -y\sin(\alpha) \\ x\sin(\alpha) \\ -\cos(\alpha) \end{bmatrix} \tag{3}$$

which is the direction of the laser from the aircraft. Figure 2a shows the estimation of the aircraft direction vector, while Figure 2b shows the calculation of the laser direction vector. We normalize it again, multiply it by -1 to orient it towards the aircraft instead of from it, and it is ready to be used for the final scalar product with the corresponding point's normal vector to determine its correct orientation.

Limitations of this method

We may encounter problems in cases where the direction of the normal vector is close to being orthogonal with the direction of the laser. Indeed, since near-orthogonal vectors' scalar products are close to 0, which means close to both a negative and a positive value, and since the location of the points bears itself some inaccuracy, a small error for the direction of a normal could arise, and easily lead to a significant change, making us pick an orientation that turns out to be the wrong one. An example of this can also be seen in Figure 1. Consequently, we could set a limit in absolute value, under which we would reject this method's determination. Possible values for these limits are shown in Table 1. We, therefore, can track the points where the method was uncertain and use another method to confidently orient the remaining subset of vectors.

2.4.2 The normal orientation continuity method

For the rejected normals, we could use a continuity method based on the fact that there is surface continuity along the terrain, which means that within a certain neighborhood, the normals are locally

$> +0.4$	accept
< -0.4	flip and accept
else	uncertain

Table 1: Proposed values for acceptence and rejection of the normal orientation

similar (all coordinates are bounded by an epsilon). Even if the set of acquired points is, of course, discrete, they are deemed forming a fine enough mesh to be able to pick up on very uneven surfaces. Hence, we could use our k-d tree to take an odd number of closest neighbors that happen to also have their corresponding normals in the "confirmed" subset. We then compute a scalar product for each of these normals and flip the current one in case the majority of computed values are negative.

Limitations of this method

For the same reason mentioned before, specifically the normals being locally the same, we generally will encounter the rejected normals all in the same spots located next to each other and forming clusters. This means that in the case of a point in the middle of this cluster, we would find the nearest "confirmed" neighbors to be not very close or even possibly very distant from that considered point. This would evidently pose great accuracy problems and could, thus, not be viewed as a continuity method since the locality is lost.
We thus need to implement a way to only consider the points close to the confirmed ones, perform our computation, and include them in the confirmed subset before running the method again. This is done until all points have been confirmed. The degree to which they are close must be decided through a threshold that is best decided empirically. However, we didn't yet test or implemented that method fully concerning the theoretical strategy we devised.

3 Implementation

In this section, we go over the structure and detail of our implementation. The theoretical considerations specified above are linked to specific parts of our code, and different stages in the development process are presented. Furthermore, the most relevant interfaces for the classes are presented.
The project is entirely implemented in `C++` and requires the `LAPACK` and `BLAS` libraries as well as the `tinyply` library to facilitate `.ply` file output. Additionally, the implementation can take some advantage of `OpenMP`.

3.1 Structure

The code is structured into multiple files implementing the different aspects of the code. More specifically, the project is divided into

- `point.<hpp/cpp>`:
  Implements a simple `Point` class. Next to the position values, the structure also stores the associated GPS time, scan angle, laser direction vector, and confidence flag.

- `point_cloud.<hpp/cpp>`:
  Implements the class `PointCloud` that stores the points inside a vector and a k-d tree. It additionally stores the normal vectors upon computation.

  The class features a reader for *LAS* files in its constructor and writers for both `.vtk` and `.ply` files in the `writeVTK()` and `writePLY()` methods. The *VTK* writer is manually implemented and uses the vtk legacy file format 2.0. The *PLY* writer, on the other hand, uses the `tinyply` library.

The class also features a method `prepareDirection()` that computes the scan direction vectors from the flight data using the above-described method. It implements a sliding window algorithm over the points that are a-priori sorted by GPS time. Here points are assumed to be in a single line with the same direction vector until the data indicates otherwise. Once a set of points with these properties has been identified, the window takes the first and last point with the minimum scan angle and computes a direction vector.

```
class PointCloud
{
public:
  int n_points = 0; /*Number of points*/
  std::vector<Point> points; /*Vector containing all points*/
  KdTree kdtree; /*KdTree allowing for fast processing of the points*/
  std::vector<std::vector<double>> normals; /*Surface Normals*/

  /*Constructor with LAS file reader*/
  PointCloud() = default;
  PointCloud(const char *filename);

  /*Pre-processes the flight data */
  void prepareDirection();

  /*Main computation function for the Surface normals.*/
  void computeSurfaceNormals(const int &n);
  /*Writer functions for the VTK and PLY file format*/
  void writeVTK(const char *filename, const bool set_surface_normal=true);
  void writePLY(const char *filename);
};

```

Listing 1: PointCloud class

It also features the method `computeSurfaceNormals()` to compute all surface normals to the points of the point cloud. The computation uses methods implemented in the `normal.cpp` file to compute the individual normals. Additionally, some light `OpenMP` parallelization to speed up the computation is implemented. For this purpose, it simply computes the normal vector for multiple points in parallel. Although a higher level of parallelism is possible, it has not been explored in our implementation.

- `kd_tree.<hpp/cpp>`:
  Implements the classes `KdNode` and `KdTree` of the main k-d tree datastructure. Here the methods described above are implemented.

```
class KdTree
{
  std::shared_ptr<KdNode> root;

public:
  KdTree() :root(nullptr)
  {}
  KdTree(std::shared_ptr<KdNode> &node) : root(node)
  {}
  KdTree(Point &p) : root(std::make_shared<KdNode>(p))
  {}

  /*Operations concerning a single node of the tree*/
  void insert(const Point &p);
  std::shared_ptr<KdNode> find(const Point &p);
  std::vector<Point> kNearestNeighbors(const int k, const Point &p);

  /*Function that builds a more balanced tree from a set of points*/
  void buildBalanced(std::vector<Point> points);
};

```

Listing 2: KdTree class

- `normal.<hpp/cpp>`:
  Implements the methods for computing the surface normals using LAPACK and BLAS routines.

  Specifically, the method `computeSurfaceNormal(Point &p, std::vector<Point> &pts)` that takes a point and its neighborhood. It then uses the `BLAS` routine `dger(...)` to successively assemble the covariance matrix using the rules from 1. This has the added advantage that the resulting matrix is already properly formatted for the application of the Eigenproblem solver from `LAPACK`.
  After the assembly of the matrix, we use the `dgeev(...)` function for general eigenvalue problems to compute all eigenvalues and associated right eigenvectors of the covariance matrix.
  Following this step, the orientation of the resulting surface normal is checked and adjusted according to the given rules.

- `main.cpp`:
  Main file using the above specified implementations to build the program.

```
1     PointCloud pc("../data/example.las");
2     pc.prepareDirection();
3     pc.kdtree.buildBalanced(pc.points);
4     pc.computeSurfaceNormals(15);
5     pc.writePLY("example.ply");
6
```

Listing 3: Main algorithm

  The execution follows the main steps of reading the *LAS* file, preparing the laser direction vector for better surface normal orientation, building the balanced k-d tree, computing the surface normals, and writing the results in the specified file.
  This order of execution should not be drastically changed as unforeseen bugs might arise.

3.2 Usage

The creation of the buildsystem is done using `CMake` with the provided `CMakeLists.txt`. More information about the compilation and how to use the code can also be found in the corresponding `README.md` file.
The code should only be used with LIDAR files from the swissSURFACE3D dataset, although might even work with others that feature the same scanning method and file format.

3.3 Challenges

The development of the code proved to be challenging at times. This section seeks to discuss the different stages and the related difficulties that followed the implementation process.

Most difficulties came with computing the correct orientation of the surface normal vector. While the process of computing its direction is well documented mathematically, the orientation requires more work and experimentation.
First, estimating the aircraft's direction proved challenging since there were errors induced by the GPS time data. We found occasions where multiple points of the data set featured the exact same GPS time. This leads to instabilities when calculating the flight line pointwise.
The problem was ultimately solved by introducing the above-described sliding window algorithm for computing the flight direction. As two neighboring points are now never directly compared to compute the direction.

Additionally, a first approach to computing the flight direction was using the so-called *Edge of Flight Line*, which is set to be true if the end of a scan is reached. However, it turns out that the scan referred to by the documentation is the entire scan spanning over multiple files. Thus, it is

impossible to predict if a file will even contain any indications about the edge of flight.
As with the previous difficulty, this was circumvented by using point data, scan angle, and GPS time values as indicators for a flight edge. Using this approach, we are able to thus relatively accurately reconstruct the flight directions.

4 Experiments

Multiple experiments have been conducted to observe the behavior of the surface normal estimation in different terrain. We used the tool `meshlab` to reconstruct the surface of the given point cloud using our estimated surface normals while using 15 nearest neighbors in computation. This was done using the *Screened Poisson reconstruction* filter with the following settings:

- *Reconstruction Depth* = 10
- *Minimum number of samples* = 4
- *Interpolation Weight* = 8

The experiments were thus largely evaluated by looking at the quality of the resulting surface reconstruction. While a mathematical evaluation may lead to more insight into the correctness of the estimated normals, it is not considered here.
We consider different areas of Switzerland with different geographical features to analyze the approximation quality.

5 Results

The results obtained by the *Screened Poisson reconstruction* with `meshlab` are presented here. To

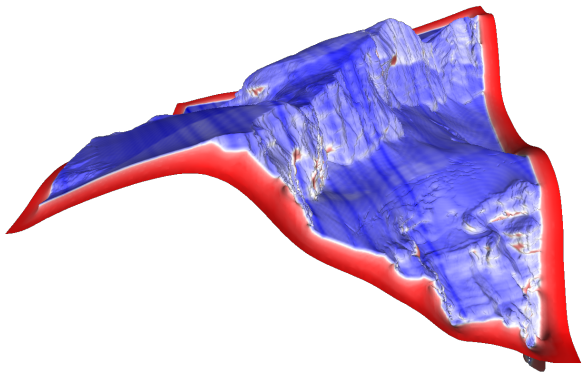


Figure 3: Surface reconstruction of a mountain peak

get a broad sense of the quality of the computed surface normals in practice, we purposefully chose

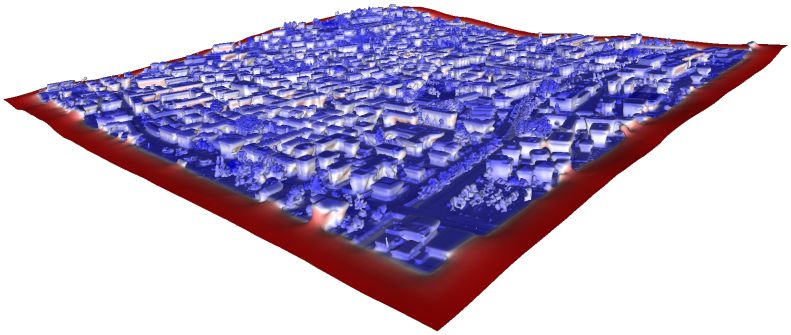


Figure 4: Surface reconstruction of part of the city of Lugano

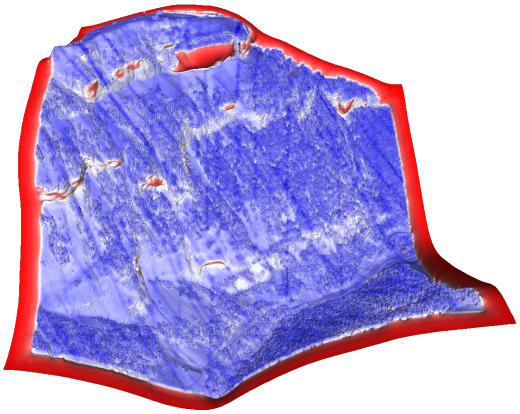


Figure 5: Surface reconstruction of a mountain site with many cliffs and overhanging features

three different areas of Switzerland for our evaluation. These areas are chosen such that they have distinct unique geographical features and allow for a general representation of Switzerland's overall landscape. Figure 3 shows the resulting surface on the peak of a mountain. Figure 4 shows the surface reconstructed from inside the city of Lugano, and Figure 5 shows the result of an area with many cliffs and overhanging features.
Furthermore, every picture is colored such that the red areas show the sections where the approxi-

mation quality is low, and the normals are probably not correctly oriented.

We can observe that the approximation quality for relatively smooth terrain, as seen in Figure 3, is approximated with very high detail and without many errors. On the other hand, however, Figure 5 shows some of the weaknesses of the code. Here we can see a large red patch, where normally an overhanging cliff should be found. This largely comes down to incorrectly orienting the surface normals in those areas.
Another interesting observation can be made by looking at the approximation of a cityscape in Figure 4. Here the general features and overall facade of many buildings are correctly reconstructed. However, some buildings still feature red patches or oddly shaped fasades with occasional holes. These are the areas where the normals are not consistently oriented. Such a city landscape also requires a high reconstruction depth when running the *Screened Poisson reconstruction* filter, which could also be a source of error.

Notes on performance

As mentioned in the Implementation section, the code features some light multithreading using `OpenMP` to speed up the concurrent computation of the surface normals. However, the overall performance of the code is still severely impacted by the speed of the underlying k-d tree. As every step of the method utilizes some part of its functionality, it is critical for the overall program runtime. Especially the k-nearest neighbor method is critical for the performance of the surface normal computation. Thus, improvement attempts should start by optimizing this method.
As expected, the runtime is largely determined by the number of points in the current file. Unfortunately, the range of points included for the swiss LIDAR files varies greatly. For files with more than 60,000,000 points, performance can be very slow. Additionally, the output file will have the size of multiple gigabytes, which also limits the performance of the subsequent `meshlab` visualization.
In our case, the k-d tree implementation is certainly the limiting factor and performance might simply be improved using a third-party k-d tree library.

6 Conclusion and Outlook

In this project, we implemented a program computing surface normal vectors for a point cloud. Reading modern, high-resolution LIDAR scan files from Switzerland, we implement the calculation of the surface normals by computing the best plane approximations for each point in the cloud. We implemented a k-d tree to accelerate the nearest neighbor search and implemented and discussed multiple techniques for determining the correct orientation of the normal after calculating its direction.

The implemented techniques show promising results and deliver robust reconstruction of the underlying surface. The k-d tree works as intended but could be improved to speed up the entire program. One approach could be to eliminate the costly recursions found in most k-d tree algorithms. However, this is not trivial and thus requires a lot of additional work. We could also replace the k-d tree with either an already implemented and optimized k-d tree library or different accelerator data structures.

The computation of the surface normal orientation, although providing a solid basis, still fails in some cases. For this reason, we can implement an additional step of checking the continuity of the vectors that seeks to improve the accuracy even further. Next to this improvement, there might be even more sophisticated methods that are yet to be explored and could provide even more accurate results.