Project Pointless.

Identifying, visualising and pathfinding through empty space in interior point clouds using an octree approach

Broersen, Tom	4418352
Fichtner, Florian W.	4409957
Heeres, Erik J.	4016424
de Liefde, Ivo	4384830
Rodenberg, Olivier B. P. M.	4040953

TU Delft, Faculty of Architecture, Geomatics for the Built Environment

Geomatics Synthesis Project Fall 2015-16

"The direct use of explorative point cloud data for calculations, observations and classifications"
Coordinator: Stefan van der Spek

Final Review Report
November 27, 2015

Supervisor
ir. E. Verbree (TU Delft)

Project Owners
dr.ir. B.M. Meijers (TU Delft)
ir. D. ten Napel (Rijkswaterstaat)

1 Preface

This document was written within the framework of the GEO1101 Geomatics Synthesis Project (GSP) fall 2015-16, which takes place during the second year of the Geomatics for the Built Environment MSc programme at the TU Delft. This edition of the GSP focuses on the direct use of point-cloud data, collected by image photography or laser scanning. The goal is to develop uses for calculations and measurements with the original, raw point cloud data. The idea behind this goal is that point cloud data sets do not merely contain geometry, but also detailed quantitative and qualitative information about the built environment (van der Spek and Verbree, 2015).

During the synthesis project, students will apply their knowledge about the contents of the courses taught in the first year of the programme. They will experience the entire process from project definition, data acquisition, data processing and analysis, to the application, presentation and delivery. Project management aspects and collaboration between students, staff and external resources will play an important role. The objectives of the GSP for the students can be summarized as follows:

- To apply their knowledge about the different disciplines taught in the first year of the MSc Geomatics for the Built Environment;
  - GEO1001: Sensing Technologies
  - GEO1002: GIS and Cartography
  - GEO1003: Location Awareness
  - GEO1004: 3D Modelling
  - GEO1005: Decision Support
  - GEO1006: GEO-DBMS
  - GEO1007: Geo Web Technology
  - GEO1008: Geo Datasets
  - GEO1009: Geo-Info Law
  - GEO3001: Python Programming for Geomatics
- Practice teamwork in small groups;
- Experience the entire range of acquisition and processing of geo-information
- To encourage collaboration between students and staff.

Students are divided into three groups of five people. The timespan of the project is 10 weeks. The project can be subdivided into three phases:

1. Requirements analysis and planning
2. Conceptual analysis and Information gathering

3. Detailed analysis

The GSP requires three separate reports: (1) the Baseline Review, (2) the Mid-Term Review, and (3) the Final Review. The contents of each of these reports will be presented to the students and staff members. A symposium will conclude the GSP, where each team will give a final presentation to any interested student, faculty staff member, or invited guest. This document comprises the Final Review, and complies with the DID-2 and DID-5 specifications outlined in the GSP Project Guide Fall 2015-16.

This Final Review document was created to provide the project owners, supervisor and members of the faculty, and other interested readers with a complete and formal overview of the project. The document includes a wrap-up of the development process, results, and conclusions, and a statement of compliance with the requirements.

Contents

1 **Preface** **II**

2 **Executive Summary** **VII**

2.1 Background . . . VII

2.2 Purpose statement . . . X

2.3 Methods . . . X

2.4 Support for the conclusions . . . XII

2.5 Conclusions . . . XIV

3 **Introduction** **1**

3.1 Background . . . 1

3.2 Purpose Statement . . . 4

3.3 Methods . . . 4

3.4 Top Level Requirements . . . 4

3.5 Boundary Conditions . . . 5

3.6 Reading Guide . . . 6

4 **Project Design and Development Logic** **7**

4.1 Rich Picture . . . 7

4.2 Organisational Breakdown Structure . . . 7

4.3 MoSCoW . . . 10

4.4 Work Breakdown Structure . . . 10

4.5 Media Outreach Strategy . . . 10

5 **Concepts** **16**

5.1 Octree Structure of Point Cloud . . . 16

5.2 Empty Space Identification . . . 26

5.3 Database management system . . . 28

5.4 Visualisation . . . 28

6 Implementation 31

6.1 Workflow Design . . . 31

6.2 Data Acquisition . . . 31

6.3 Interface Design . . . 32

6.4 Engineering Design . . . 33

6.5 System Characteristics . . . 43

6.6 Performance and Scalability . . . 45

6.7 Operations and Exploitation plan . . . 50

6.8 Discussion . . . 52

7 Use case: 3D path finding in empty space 54

7.1 Path Finding Algorithm . . . 54

7.2 Neighbour Finding . . . 55

7.3 Results of the Use Case . . . 58

8 Conclusions 62

8.1 Octree Generation and Space Finding . . . 62

8.2 Use case: 3D path finding in empty space . . . 63

9 Future Work and Recommendations 65

9.1 Rotate Point Cloud . . . 65

9.2 Improve Octree Visualisation . . . 65

9.3 Merging Coordinates with Materialised Path . . . 65

9.4 Path Finding . . . 66

9.5 Laz File Compression . . . 67

9.6 Sphere Leaf Nodes . . . 67

9.7 Connecting Pointless to Geographic Information Systems . . . 68

9.8 Combine Path of Scanner with octree Structure . . . 68
9.9 Noise removal . . . 68
9.10 Use octree for volume calculation . . . 69
9.11 Dynamic octree . . . 69
9.12 Improving performance . . . 69
9.13 Additional Areas for Future Research . . . 69
10 Acknowledgments **71**
11 Appendix **75**
11.1 MoSCoW . . . 75
11.2 Costs Overview . . . 79
11.3 FMW Workbench Worksheet . . . 80

2 Executive Summary

This executive summary describes the global content of Project Pointless, and contains the most important technical details of the project. It is written as an informative summary of the full report, and can be read as a separate document.

2.1 Background

Point clouds are (often large) collections of points registered in three dimensional space, which can be stored in combination with additional properties such as colour and signal strength. Together, they can provide a realistic view of the world as it was measured by laser scanning technologies or derived using photogrammetry. Point clouds are a collection of (x,y,z) coordinates and properties, which have the potential to provide a wealth of information, and can be used to accurately describe features, for example the point cloud of the Project Pointless team in figure 1. Although the individual points merely represent (x,y,z) and colour (r,g,b) values, the combination of these points provide an accurate representation of the five team members. The trained human eye can recognize their facial expressions, posture, type of clothing, and can even recognize that the team member in the middle is injured since he has a cast around his ankle. For applications involving point clouds, often intermediate geometric models such as 3D BIM models are generated, and the original dataset is deleted or archived (van der Spek and Verbree, 2015). This considerably reduces the amount of data, but also inherently infers a loss of information. Because these models are a simplification of the original point cloud, the full richness and details of captured data can go lost (Verbree and van Oosterom, 2014). However, if operations are carried out using the 'raw' data directly, this has the potential to enrich the original point cloud dataset.

A more traditional implementation for using point clouds to create a 3D representation of a real world object, is to take a 2D representation (a polygon for example) and using the point cloud to extrude this polygon to level of detail (LOD) 1, 2 or 3. However, if this method is performed and the original point cloud is afterwards not being used anymore, then this creates a model with a lot less detail than was originally captured. Figure 2 shows what the 4 different levels of detail mean: LOD1 is a basic block model of a building and with LOD2 there is a more accurate roof shape. LOD3 includes more accurate facades and LOD4 models the interior of a building as well. Figure 3 shows LOD1, LOD2 and LOD3 models compared to the original point cloud. Even the LOD4 model, which also models interior space, does not have any information about the indoor world. It merely allows the user to go inside the extruded polygon. The original point cloud with all its detail does have information about the inside world, and applications like the ones presented in this report are the first steps into the full exploitation of all the richness of explorative point clouds. The usage of explorative point clouds holds major advantages such as no need for an intermediate geometric model. This saves storage space, time and most importantly retains all details included in the point cloud. Furthermore it is also easier to update and can be reused.

Point clouds can be classified into different categories with respect to their measurement environment; aerial point clouds are obtained using aerial vehicles such as airplanes and drones, terrestrial

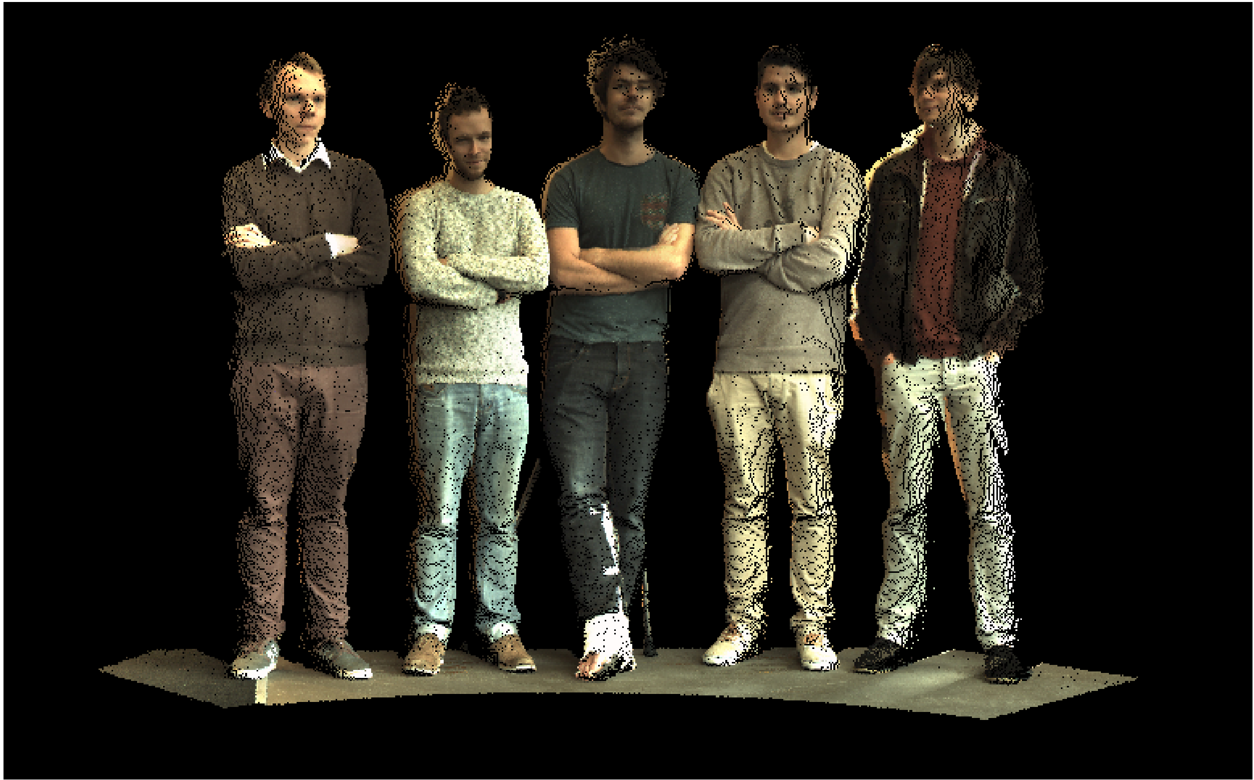

Figure 1: Point cloud of team Pointless (from left to right: Ivo de Liefde, Florian Fichtner, Erik Heeres, Olivier Rodenberg and Tom Broersen)

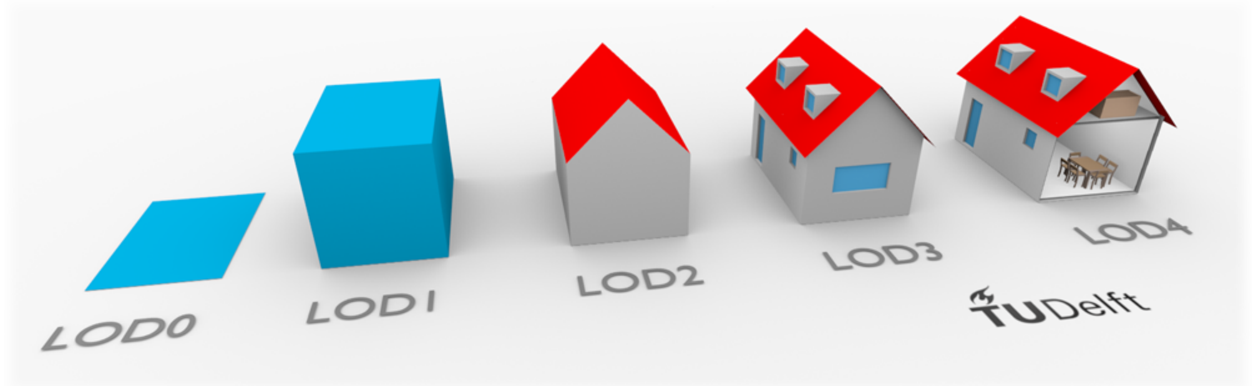

Figure 2: Levels of detail in GIS
image courtesy to Biljecki (2015)

Figure 3: Levels of detail of Bouwpub compared to point cloud

point clouds are obtained using ground equipment such as cars or total stations, while indoor point clouds focus specifically on interior spaces. Each of these types of point clouds serve their own range of applications. For indoor point clouds these applications range from automatic generation of indoor models to object recognition and path finding. The focus of applications involving indoor point clouds is almost always on identification of the boundaries of space, or of the objects inside it, instead of on the space itself. However, classification of the empty space can serve a multitude of purposes. One very obvious application would be (empty) volume estimation. A company might want to estimate how much storage space is left in a silo or warehouse, for which estimation of the volume of empty space from a point cloud can be useful. Also, the empty space can be used for (walkable) path finding and creating realistic point cloud walkthrough applications, as it can restrict users to walk only through the empty space in the point cloud. Especially for collision avoidance for autonomous robotics finding and structuring the empty space can be useful (Payeur, 2006). Space fitting queries provide another potential use case, these can be used to check whether large objects fit through the empty space. Rijkswaterstaat currently uses a single maximum height value to indicate whether a vehicle fits under a bridge or through a tunnel. For many cases this maximum height value will be sufficient, but when the shape of the vehicle or the shape of the bridge or tunnel are more complex, a single height value will not be an effective solution. A three-dimensional model of the empty space can then be of added value to calculate directly whether the passing object can fit through. This can also be useful in maritime applications to verify whether a ship fits through a specific waterway, or in indoor applications for calculating a possible route for a drone to fly through a building. Despite the many potential uses, no earlier research to identify empty space in point clouds is available in present day scientific literature.

Octrees are based on the recursive and non-uniform subdivision of space and are a common approach to structure and segment 3D point clouds. Earlier research, like Wang and Tseng (2011), Zhou et al. (2011) or Schön et al. (2013), used an octree structure to segment point clouds, but also applications such as Potree and the Point Cloud Library, make use of an octree structure. Octree structures provide an efficient way of indexing the point cloud, and are handy for neighbourhood operations (Burrough, 1986). Furthermore, they can be used for the partitioning of space and result in a hierarchical structure. Project Pointless hypothesizes that by using the octree structure to segment and index the point cloud, also the empty space can be derived in an efficient way.

2.2 Purpose statement

Many potential use cases of empty space classification were identified in section 1.1. Project Pointless tries to comply to these use cases by taking a novel approach for processing point clouds; instead of identifying boundaries of objects, it will classify the empty, pointless, space in any interior point cloud by using the efficient octree structure. This will be done directly from the raw point cloud data, without creating intermediate models of reality, thus retaining all the details present in the original point cloud. A smart workflow will be designed which efficiently identifies and visualises the empty space directly from the point cloud source. This workflow should be universally reusable as a basis for other projects involving (interior) point clouds. The computation adds a structure to the raw point cloud dataset which can be understood and interpreted by computers. To showcase the potential of empty space identification, an indoor routing algorithm will be used to compute the shortest path for a drone through a building on the basis of the identified empty space. The project objective can be described in one sentence:

"Creating a smart and universally reusable workflow which efficiently identifies and visualises the empty, pointless, space directly from any interior point cloud source using an octree approach, and computes the shortest path through an empty space."

2.3 Methods

Following from the purpose statement in section 1.2, this section will describe the methods used in this project to generate the octree, and accordingly identify and visualise the empty space in interior point clouds. This section will also explain the approach taken to implement the use case, using an indoor routing algorithm to compute the shortest path through a building on the basis of the identified empty space. The descriptions in this section will be brief. For an extensive documentation, the reader is referred to chapter 4: concepts of the Final Review report.

2.3.1 Octree generation and empty space finding

This project makes use of an octree structure to index and segment the original point cloud. An octree is a tree structure where the volume is recursively subdivided "into eight congruent disjoint cubes (called octants) until blocks of a uniform colour are obtained, or a predetermined level of decomposition is reached" (Samet, 1988). This means that with every level of the tree the entire space is split into 8 blocks. The space continues to recursively subdivide into 8 blocks until either the maximum amount of levels is reached or a block is completely empty.

The octree implemented by Project Pointless is a so-called linear octree, which makes use of materialised paths to refer to the nodes in the tree. Such materialised paths are built by taking a point and traversing the tree from root node to leaf node, thereby iteratively adding to a string the name of the node where the point is located in at that level of the tree. This means that every level in

the octree will be represented by one single digit in the materialised path. After octree generation, every point in the point cloud has one materialised path assigned to it, referring to the leaf node it is located in. The cells in the octree will be enumerated in z-order, with values of the octants ranging from 0 to 7. To obtain the maximum resolution for the empty cells in the octree, the non-empty nodes (thus nodes containing points) are always forced to split up to the maximum user-specified tree level. There can thus be no non-empty leaf nodes in the tree below this maximum level.

To generate the materialised paths for every point in the point cloud, a technique called binary masking is used. Since every cell in the octree splits into eight smaller octants, it is possible to encode with only 3 bits in which octant a point is located. Along every dimension of a cell, the point can be at one of the two sides of the split line, which can be indicated by either a 1 or a 0 (true / false). In three-dimensional space it is thus possible to specify for a certain point in which octant it is located by a combination of three bits; x[0 or 1], y[0 or 1], z[0 or 1]. To accordingly generate the materialised path using this method, first a conversion of the coordinates of the point into binary numbers for x,y, and z takes place. Then every digit in these binary numbers will be iteratively compared to the digits in a binary mask, the form of which depends on the current level in the octree. For level 1 (and binary x, y, z coordinates of [101,011,001]), the binary mask will be 100 (so only the first digit, corresponding to the first level in the tree, is evaluated). Iteratively comparing the digits in the (x,y,z) coordinates to the binary mask using a Boolean AND operator, will result in the following outcomes: True for x, False, for y, and False for z. Then, if x is true, a 1 is added to a temporary variable, if y is true, a 2 is added, and if z is true, a 4 is added. The sum of these numbers is the number that represents the node in which the point is located in at the current level in the tree. This procedure is repeated for every level in the tree, every time concatenating the node numbers to form the materialised path. For the above procedure to work, the point cloud first needs to be translated and scaled. The translation is necessary to define the lower left corner of the front of the bounding box as the origin of the point cloud. The scaling is needed so that the method can be used for any type and size of point cloud, and so that every cell has coordinates represented by integers between 0 and 2n. Then by removing the decimals from every point's coordinates, the points can be snapped to the particular cell it is located in.

The above procedure results in a list of materialised paths, referring to all the non-empty leaf nodes where the points in the point cloud are located in. This procedure does not yet explicitly store the empty nodes. To find these, a list is generated containing all non-empty nodes. This list is compared to a list of all potential nodes, and if any of these potential nodes are not in the list of non-empty nodes, then this means the node is empty. The result will be a list of empty nodes, which can occur at any level in the octree, and can have different sizes depending on their level in the tree. The geometry of these nodes then has to be determined. The size of the nodes in local coordinates is equal to 2n, where n is equal to the difference between the current level and the maximum number of levels in the tree. The origin point of the nodes can be found by using the same binary masking technique to convert from materialised paths back to decimal coordinates. All points and the octree structure with the empty and nonempty nodes are stored in a DBMS for efficient access and retrieval.

2.3.2 Visualisation

After the points and octree structure have been stored in the database, it is possible to use the data in further applications. However, it is useful to have a tool that allows visual inspection of the data to verify whether the generation of the octree structure and the empty nodes was successful. To do this a WebGL viewer was implemented that can retrieve data from the octree in the database and visualise it. The use of WebGL requires no additional plugins to be installed.

The visualisation of the empty space is done using a combination of three.js and FLASK. Three.js is an open source JavaScript library for visualising three dimensional scenes in the browser. Its goal is to make WebGL easy work with. Since Three.js runs in the browser a website had to be created which runs Three.js and connects it to the database with the empty space and the point cloud. FLASK has been selected to deal with the backend of this website. FLASK "is a micro framework for Python" (Flask, 2014). This allows to use the backend to call the octree generation script which was also written in Python. After the script has been called FLASK connects to the Postgres database and loads the data into a Three.js scene.

2.3.3 Use case: 3D route calculation

The A* algorithm was used to compute the shortest path between two points in the point cloud. A path is here defined as the route between the two nodes the points are located in. To find the shortest route, recursive exploration of adjacent nodes is required (Noto and Sato, 2000). These so-called 'neighbours' of every node are found by using a method developed by Vörös (2000). This method is able to find all inner and outer equal-, smaller-, and larger-sized neighbours for every node by utilizing the hierarchical structure of the octree.

2.4 Support for the conclusions

The project successfully created a Python script, which takes a point cloud from any .las file, and efficiently generates a linear octree using binary masking. All points in the original point cloud, with the materialised path of the nonempty leaf nodes they are located in, were stored in a PostgreSQL database. From these materialised paths of the nonempty leaf nodes, the empty nodes and their geometry can be quickly extracted. The current octree implementation was written such that an octree of 8 levels deep is generated, which equates to +/- 10cm resolution for the dataset used here. The speed of the octree generation script was evaluated by comparing between generation of octrees of 6, 7, and 8 levels deep. This comparison showed that doubling the octree resolution from 6 to 7 levels deep, causes only a small increase in total processing time (+/- 10%), and the increase from 7 to 8 levels causes an increase of only 2%. Moreover, almost all of this increase in processing time can be accounted to the increased time needed to write the results to the database. The generation of the octree itself, and derivation of the empty space from this octree, only takes 1% more time when increasing the number of levels in the octree from 6 to 8. This indicates that the octree generation

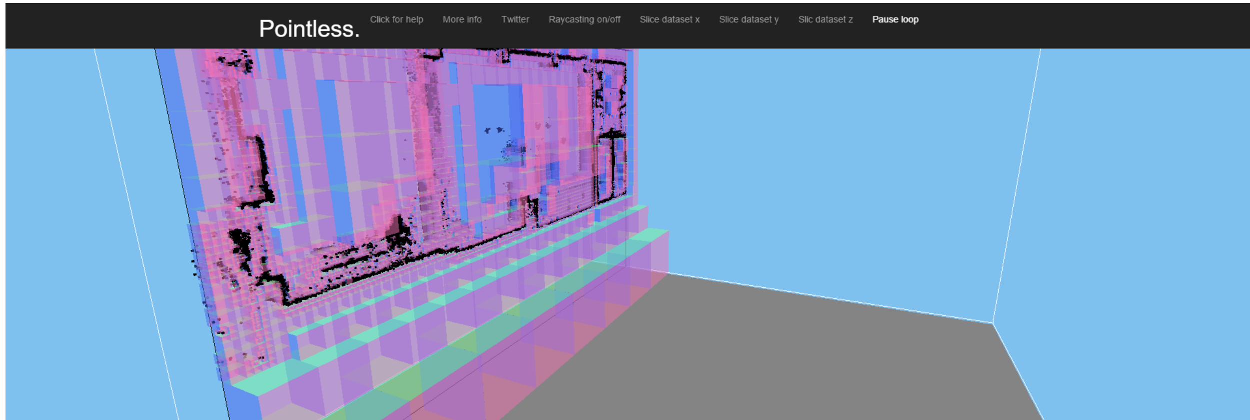

Figure 4: visualisation of empty space in a slice of a point cloud

script scales very well when increasing resolution. Moreover, 65% to 74% (depending on the depth of the octree) of the total processing time is spent on writing points to a file, thus indicating that the script can become even faster if a method is found to avoid this step. The influence of a change in point cloud size was also tested. The script was tested with point cloud sizes of 2.3 million, 4.6 million, and 9.2 million points. This comparison shows that the increase in the amount of points in the point cloud has an impact on performance, the time spent on writing points to a file more than doubles when doubling the amount of points. Also, the relative amount of time spent on this procedure increasing from 65% for 2.3 million points, to 74% for 9.2 million points. Writing points to a text file is by far the least scalable component of the script. However, here again the generation of the octree and derivation of the empty space shows to scale well, with the amount of processing time increasing 3.6 times when including 4 times more points. It is presumably the combination of fast snapping of points to the leaf nodes, binary masking, and membership testing using sets in Python, which makes the octree generation and empty space finding so efficient.

Visualisation of the points and empty space using the currently implemented combination of WebGL, FLASK, and three.js, worked for relatively small point clouds (e.g. 2 million points). A slice of the data (empty nodes and points) is visualised in figure 4. It shows that the empty space in between the points is successfully classified into the empty cells, and the cells become smaller when getting nearer to the points. The current implementation of the viewer is however not sufficient for larger point clouds, which can cause crashes. This could be solved in the future by using Potree, or an adaptation of Potree for visualisation purposes.

The route finding algorithm (Figure 5) has been timed while computing routes in octrees with different amount of levels. These numbers show that there is still room for improvement of the performance: for six levels the algorithm takes less than 20 seconds to find a path, for seven levels it takes two minutes. However for eight levels it suddenly takes almost half an hour to compute a route. This is because the number of neighbours that should be checked in the A* algorithm increases exponentially with the addition of an extra level.

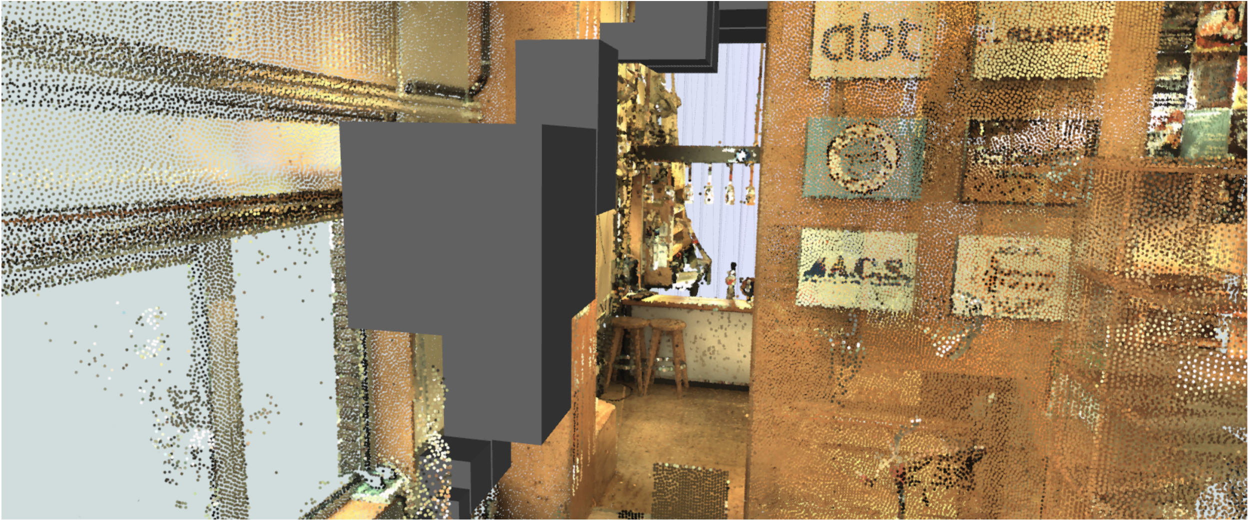

Figure 5: visualisation of a route (gray cubes) through the empty space in a point cloud

2.5 Conclusions

Project Pointless aimed to create a smart and universally reusable workflow which efficiently identifies the empty space in a point cloud, store this in an efficient way, visualise it and use this octree structure in an example application. The empty space as well as the points were structured in an octree. This allowed the team to build a successful application of a path finding algorithm using only the empty space of a point cloud. The performance and scalability have been statistically examined. The results indicate that the octree implementation is efficient and scalable. It is presumably the combination of fast snapping of points to the leaf nodes, binary masking, and membership testing using sets in Python, which makes the octree generation and empty space finding so efficient.
The method of Project Pointless does not use intermediate geometric model and therefore is explorative. The raw point cloud data is merely indexed and stored in a database management system. The procedure is fast and automated, enriches point clouds with a structure that can be understood by computers, while keeping all advantages of the raw data set.

From a performance viewpoint, the octree generation and empty space finding can be considered successful, and possible further optimisations have already been identified. Furthermore, the script is flexible since it is able to use any .las file containing an interior point cloud as input, due to the translation and scaling operations carried out on the points before further processing. The script is also free for anyone to use and completely open source. The implementation is also reliable since the script checks all input parameters on start-up to prevent catching invalid input, it makes sure that the input point cloud is in a valid file format, and whether a database connection can be established. Furthermore, the .las file containing the point cloud is first copied to ensure the original stays intact and can still be worked with during processing. Finally, before writing results to a database, they are first written to a .csv, which minimizes the connection to the DBMS and thus failure when this connection is lost.

The current implementation still knows some limitations. The performance and scalability of the implementation is limited by the writing of points to a file, and subsequently copying this file into the DBMS. Also the octree generation script is not yet scalable to multiple users since it can locally run only once at a time. Finally, visualisation of the points and empty space using the currently implemented combination of WebGL, FLASK, and three.js, is not sufficient for larger point clouds, which can cause crashes. This could be solved in the future by using Potree, or an adaptation of Potree for visualisation purposes.

The use case of 3D route calculation was found to work well and does not require any intermediate geometric models or graph network to find a route in 3D. All calculations are performed directly using the original point cloud source and generated octree structure. The current implementation of the 3D path finding algorithm is already able to successfully find a route through the empty space, but knows some limitations in its performance. Neighbours are currently checked on the fly, which means that they have to be recomputed every time the script is executed. Also the script does not scale well. However, these are limitations that can be overcome, and the use case effectively showed that the octree structure and empty space generated here has great potential for real-world use cases, and allows to quickly develop useful applications on top of it. The idea of finding and structuring the empty space and making direct use of an explorative point cloud was successfully implemented and also proved to be fruitful for future applications.

The project successfully fulfilled its goals set out at the start. The media campaign utilised by Project Pointless attracted significant attention, which shows that the topic and idea of Project Pointless meets the state of the art and that people are interested in the result as well as its process.

3 Introduction

The Final Review Report of the Geomatics Synthesis Project (GSP) aims to describe the results of the analysis and research that the group has performed. It includes the processing methodology and the quality assessment, as well as the technical details associated with the project. It describes and shows how the technology proposed by Project Pointless works.

3.1 Background

Point clouds are (often large) collections of points registered in three dimensional space, which are sometimes stored in combination with additional properties such as colour and signal strength. Together, they can provide a realistic view of the world as it was measured by laser scanning technologies or derived using photogrammetry. Point clouds are a collection of (x,y,z) coordinates and properties, which have the potential to provide a wealth of information, and can be used to accurately describe features, for example the point cloud of the Project Pointless team in figure 6. Although the individual points merely represent (x,y,z) and colour (r,g,b) values, the combination of these points provide an accurate representation of the five team members. The trained human eye can recognize their facial expressions, posture, type of clothing, and can even recognize that the team member in the middle is injured since he has a cast around his ankle. For applications involving point clouds, often intermediate geometric models such as 3D BIM models are generated, and the original dataset is deleted or archived (van der Spek and Verbree, 2015). This considerably reduces the amount of data, but also inherently infers a loss of information. Because these models are a simplification of the original point cloud, the full richness and details of captured data can go lost (Verbree and van Oosterom, 2014). However, if operations are carried out using the 'raw' data directly, this has the potential to enrich the original point cloud dataset.

A more traditional implementation for using point clouds to create a 3D representation of a real world object, is to take a 2D representation (a polygon for example) and using the point cloud to extrude this polygon to level of detail (LOD) 1, 2 or 3. However, if this method is performed and the original point cloud is afterwards not being used anymore, then this creates a model with a lot less detail than was originally captured. Figure 7 shows what the 4 different levels of detail mean: LOD1 is a basic block model of a building and with LOD2 there is a more accurate roof shape. LOD3 includes more accurate facades and LOD4 models the interior of a building as well. Figure 8 shows LOD1, LOD2 and LOD3 models compared to the original point cloud. Even the LOD4 model, which also models interior space, does not have any information about the indoor world. It merely allows the user to go inside the extruded polygon. The original point cloud with all its detail does have information about the inside world, and applications like the ones presented in this report are the first steps into the full exploitation of all the richness of explorative point clouds. The usage of explorative point clouds holds major advantages such as no need for an intermediate geometric model. This saves storage space, time and most importantly retains all details included in the point cloud. Furthermore it is also easier to update and can be reused.

Point clouds can be classified into different categories with respect to their measurement environ-

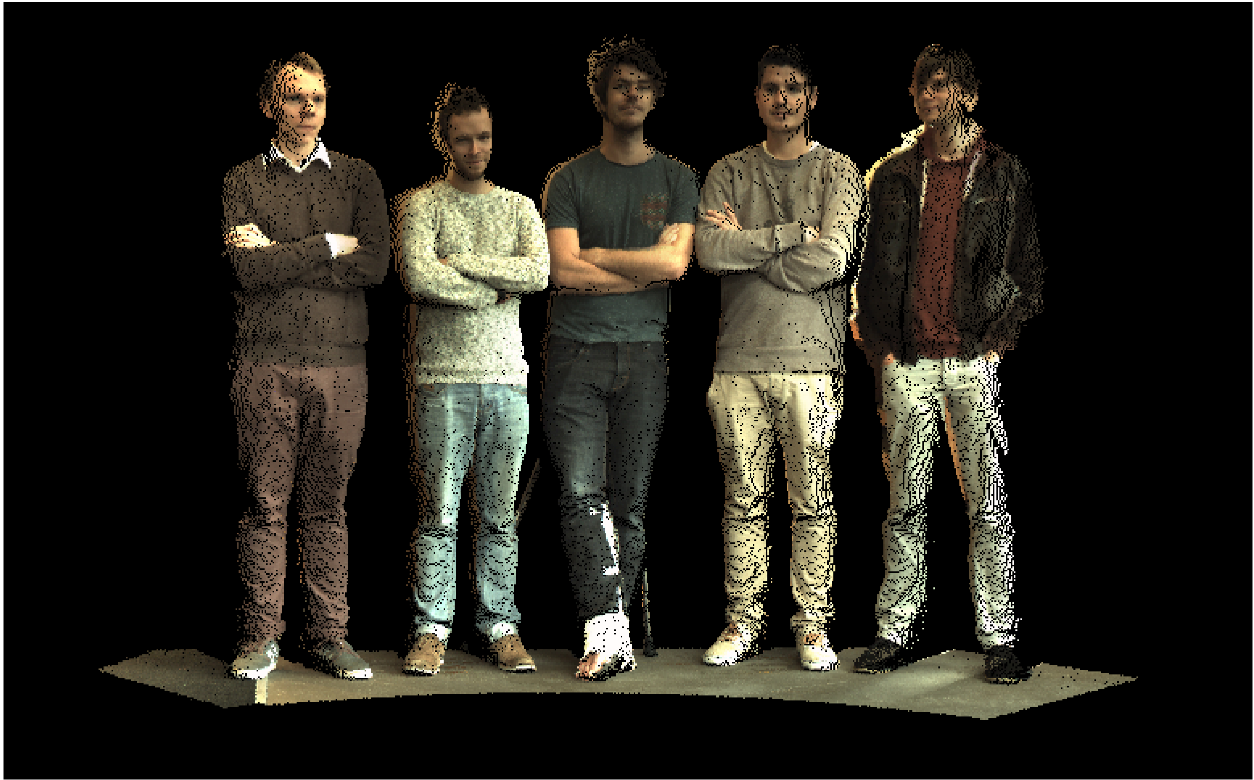

Figure 6: Point cloud of team Pointless (from left to right: Ivo de Liefde, Florian Fichtner, Erik Heeres, Olivier Rodenberg and Tom Broersen)

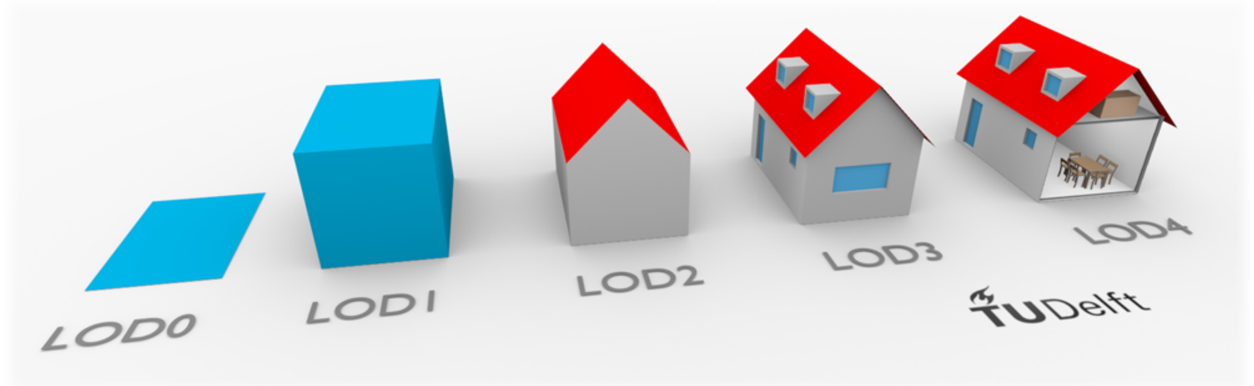

Figure 7: Levels of detail in GIS
image courtesy to Biljecki (2015)

Figure 8: Levels of detail of Bouwpub compared to point cloud

ment; aerial point clouds are obtained using aerial vehicles such as airplanes and drones, terrestrial point clouds are obtained using ground equipment such as cars or total stations, while indoor point clouds focus specifically on interior spaces. Each of these types of point clouds serve their own range of applications. For indoor point clouds these applications range from automatic generation of indoor models to object recognition and path finding. The focus of applications involving indoor point clouds is almost always on identification of the boundaries of space, or of the objects inside it, instead of on the space itself. However, classification of the empty space can serve a multitude of purposes. One very obvious application would be (empty) volume estimation. A company might want to estimate how much storage space is left in a silo or warehouse, for which estimation of the volume of empty space from a point cloud can be useful. Also, the empty space can be used for (walkable) path finding and creating realistic point cloud walkthrough applications, as it can restrict users to walk only through the empty space in the point cloud. Especially for collision avoidance for autonomous robotics finding and structuring the empty space can be useful (Payeur, 2006). Space fitting queries provide another potential use case, these can be used to check whether large objects fit through the empty space. Rijkswaterstaat currently uses a single maximum height value to indicate whether a vehicle fits under a bridge or through a tunnel. For many cases this maximum height value will be sufficient, but when the shape of the vehicle or the shape of the bridge or tunnel are more complex, a single height value will not be an effective solution. A three-dimensional model of the empty space can then be of added value to calculate directly whether the passing object can fit through. This can also be useful in maritime applications to verify whether a ship fits through a specific waterway, or in indoor applications for calculating a possible route for a drone to fly through a building. Despite the many potential uses, no earlier research to identify empty space in point clouds is available in present day scientific literature.

Octrees are based on the recursive and non-uniform subdivision of space and are a common approach to structure and segment 3D point clouds. Earlier research, like Wang and Tseng (2011), Zhou et al. (2011) or Schön et al. (2013), used an octree structure to segment point clouds, but also applications such as Potree and the Point Cloud Library, make use of an octree structure. Octree structures provide an efficient way of indexing the point cloud, and are handy for neighbourhood operations (Burrough, 1986). Furthermore, they can be used for the partitioning of space and result in a hierarchical structure. Project Pointless hypothesizes that by using the octree structure to segment and index the point cloud, also the empty space can be derived in an efficient way.

3.2 Purpose Statement

Many potential use cases of empty space classification were identified in section 2.1. Project Pointless tries to comply to these use cases by taking a novel approach for processing point clouds; instead of identifying boundaries of objects, it will classify the empty, pointless, space in any interior point cloud by using an octree approach. This will be done directly from the raw point cloud data, without creating intermediate models of reality, thus retaining all the details present in the original point cloud. A smart workflow will be designed which efficiently identifies and visualises the empty space directly from the point cloud source. This workflow should be universally reusable as a basis for other projects involving interior point clouds. The computation adds a structure to the raw point cloud data set which can be understood and interpreted by computers. To showcase the potential of empty space identification, an indoor routing algorithm will be used to compute the shortest path for a drone through a building on the basis of the identified empty space. The project objective can be described in one sentence:

"Creating a smart and universally reusable workflow which efficiently identifies and visualises the empty, pointless, space directly from any interior point cloud source using an octree approach, and computes the shortest path through an empty space."

3.3 Methods

The focus of the GSP is to get hands on experience of an entire 'geomatics' project. A literature study will be combined with a practical approach including the whole workflow of point cloud processing, starting with the acquisition of the data. The research consists of not only literature analysis, but also of discussion and problem solving sessions between team members, supervisors, stakeholders and other researchers. The generation of the reports will make the team consider each part of the project extensively before implementing it. This has helped to develop a well-structured workflow.

3.4 Top Level Requirements

The requirements specifications describe the functional and non-functional requirements of Project Pointless. The functional requirements describe what the system has to do; what should be its functionalities. The non-functional requirements describe how the system will achieve the functional requirements (Eriksson, 2012). The requirements of the project listed here are the system requirements of the project's end product.

The functional requirements of Project Pointless are:

- Any indoor point cloud which is input into the workflow should be processed to classify the empty space.

- A relational database should be used to store the point cloud and intermediary results, to allow for efficient access to the data.
- The empty space should be visualised to prove the concept.
- A 3D route finding algorithm should be able to use the empty space to find a path without going through walls and other objects. This is to show the use case of empty space classification.
- The workflow should be inspired by already existing software where possible, to save time and find the best practice, this applies both to classification and visualisation of the empty space, and identification and visualisation of visible points.

The non-functional requirements are:

- The workflow should be interoperable; it should be applicable to any type of point cloud.
- Processing of the point cloud should only take a reasonable amount of time.
- The quality of the end result should be validated; the estimated amount of empty space should be compared to empty space estimated by other techniques.
- The quality of the route finding should be validated; the resulting path has to be visualised and the distance should be compared with manual computation.
- The workflow should be open source and available online.
- The point cloud used in the project should be available online as open data.
- The end results of the project will be made available for download online.
- The visualisation of the empty space inside a test point cloud and its use cases will be made available online.
- The project will be documented in several management and technical reports, which shall be made publicly available online.

3.5 Boundary Conditions

The following boundary conditions form the framework of the Synthesis Project for team Pointless:

- The workflow should not generate any intermediary geometrical models to come to an end result.
- The workflow should use the original point cloud as source to compute the results, to avoid losing detail in the end product.

- An interior point cloud should be obtained, which will serve as input to the proposed workflow.
- The project has to be done in a 10 weeks period; the final report has to be finished by November 2nd, 2015, and the results will be presented on November 6th, 2015.
- A media campaign should be developed about the progress throughout the project.

3.6 Reading Guide

This document is divided into six further chapters. Chapter three explains the project design and development logic of Project Pointless. Afterwards the concept of the technological approach to structure, find and visualise the empty space will be explained in chapter four. In chapter five the actual implementation will be addressed. This includes the engineering design of all major components of the application. Subsequently a use case of the application will be described in chapter six to illustrate the practical benefits of the research. In chapter seven the conclusions of the research will be addressed. It reflects on the performance and scalability of the application and relates it to the methods described in the concept. Chapter eight presents the recommendations for future research.

4 Project Design and Development Logic

Because the project was mainly research based and the objective left a lot of room to shape it based on our own ideas, the process changed over time. Project owners changed, goals were readjusted along the way and new possibilities for acquiring data occurred. In this chapter the final state of the project design is discussed. The complete project management diagrams can be found in the appendix.

4.1 Rich Picture

The Rich Picture (figure 9) indicates the scope, important aspects, stakeholders and potential pitfalls of the project. The pictures show clearly that the work is divided into three key topics; the acquisition, the processing and the visualisation. The dotted lines show the alternatives on the planned process. The project group Pointless forms the connection between the different topics. There are three main risks identified in the rich picture: a financial risk in the data acquisition phase for both the use of the Leica C10 laser scanner and the Zeb1 mobile laser scanner. This means that if there would not be enough financial resource for Project Pointless there would be no point cloud to create and to test the implementation with (see chapter 6.2). The third risk is about the time it would take to create a viewer that can visualise large amounts of data. Even though the main focus of Project Pointless does not lay in the visualisation of the results, a method to show the result to a broad audience increases the communication possibilities of the project and can play an important role for the reception of the team's ideas and methods.

The rich picture also visualises the project owners: Martijn Meijers from Delft University of Technology and Dick ten Napel from Rijkswaterstaat. During the GSP the project owners have provided the team with vital information and valuable insights. Martijn Meijers has been the person that guided the team on a scientific level and helped with technical challenges. Dick ten Napel has been very important to keep the scientific researched connected to the outside world. Project Pointless is doing research about point clouds to increase their values. Therefore, getting input from Dick ten Napel as a potential user has proved to be very valuable. All of the feedback the team received from the project owners has been carefully documented and used as input for the decisions that have been made over the course of this project.

4.2 Organisational Breakdown Structure

In the organizational breakdown structure (figure 10) the different stakeholders are grouped in three different categories; supervision, the project team and the 3rd party stakeholders. The latter consist out of companies or researches which are not directly (through supervision) involved in the project, but instead can help through feedback or direction pointing.

The hierarchy inside the team of Project Pointless was flat and the structure can be described as a

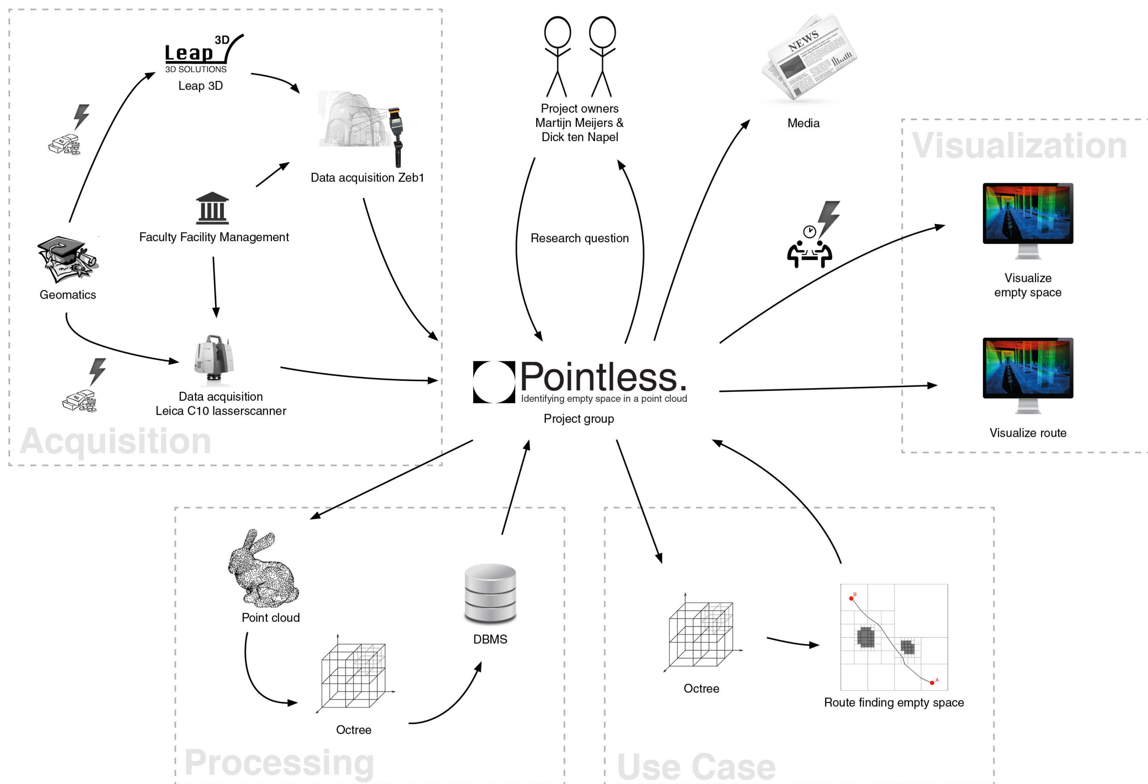

Figure 9: Rich Picture of project Pointless

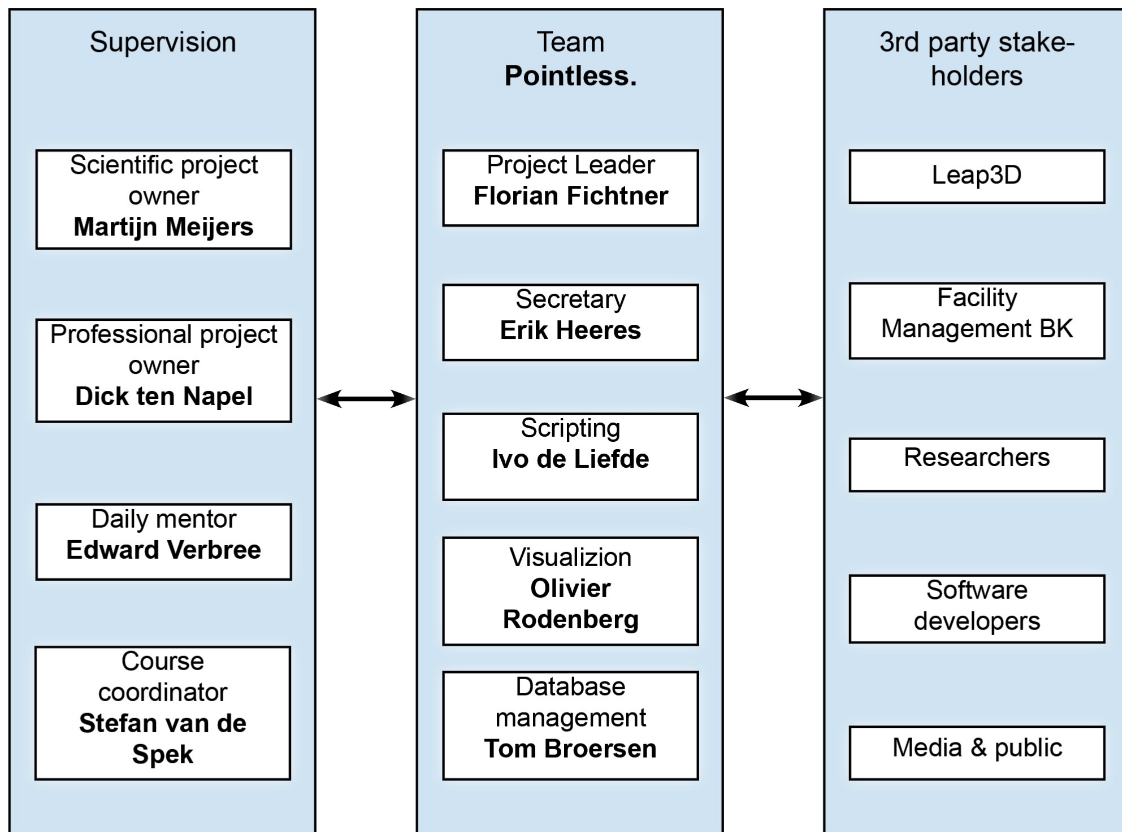

Figure 10: Organizational Breakdown Structure

network rather than a tree. Beforehand, the responsibility of specific tasks were assigned to team members, but in practice the tasks were divided on a daily basis in a more natural way. At every phase of the project, the different tasks that had to be done were defined and divided between the team members during SCRUM meetings.

4.3 MoSCoW

The MoSCoW method is used for the prioritization of individual project requirements to reach a common understanding with stakeholders about each delivery. The term MoSCoW is derived from the first letter of each of four prioritization categories: Must have, Should have, Could have, and Would like but won't get (Wikipedia, 2015). The quality of each requirement depends on sub-deliveries which are also structured using the MoSCoW method. In the appendix 11.1 the complete overview of the MoSCoW are given and compared with the results.

The Musts of the Musts are the killer requirements of the project, without accomplishing them the project would fail. An example for such a necessity would be the acquirement of a point cloud, which is absolutely necessary for achieving all other requirements. Driving requirements can be found in the category Should. They help pushing the whole project forward. An example would be to store the point cloud in a database, which would increase performance and handling possibilities of the dataset and enable the team to perform more efficient on the following tasks. However, the project can also succeed without them being fulfilled.

4.4 Work Breakdown Structure

As already mentioned the team changed the use case from improving the visualisation of point clouds by only show 'visible' points, to the use of the structured empty space to find a route in a point cloud. This also had influence on the Work Breakdown Structure. Other changes that have been made are that the team only acquired a point cloud of the Bouwpub instead of the whole faculty and that the team does not cluster the data in the database. The index used for indexing the data in the DBMS, is the self-defined octree structure developed during the project. All changes are marked in red in the following figure 11.

4.5 Media Outreach Strategy

Right from the start of the GSP an active media outreach strategy was executed by Project Pointless. A Twitter account (@ProjPointless) was created and multiple times per week tweets were sent out to inform the followers about the progress (figure 12). Within one week there was also a website online containing information about the project, including weekly updates on the progress and the Twitter feed. Twitter was mainly used for reaching out to Geomatics Delft and the faculty of architecture, on top of that it was also found useful to connect to companies that started to follow

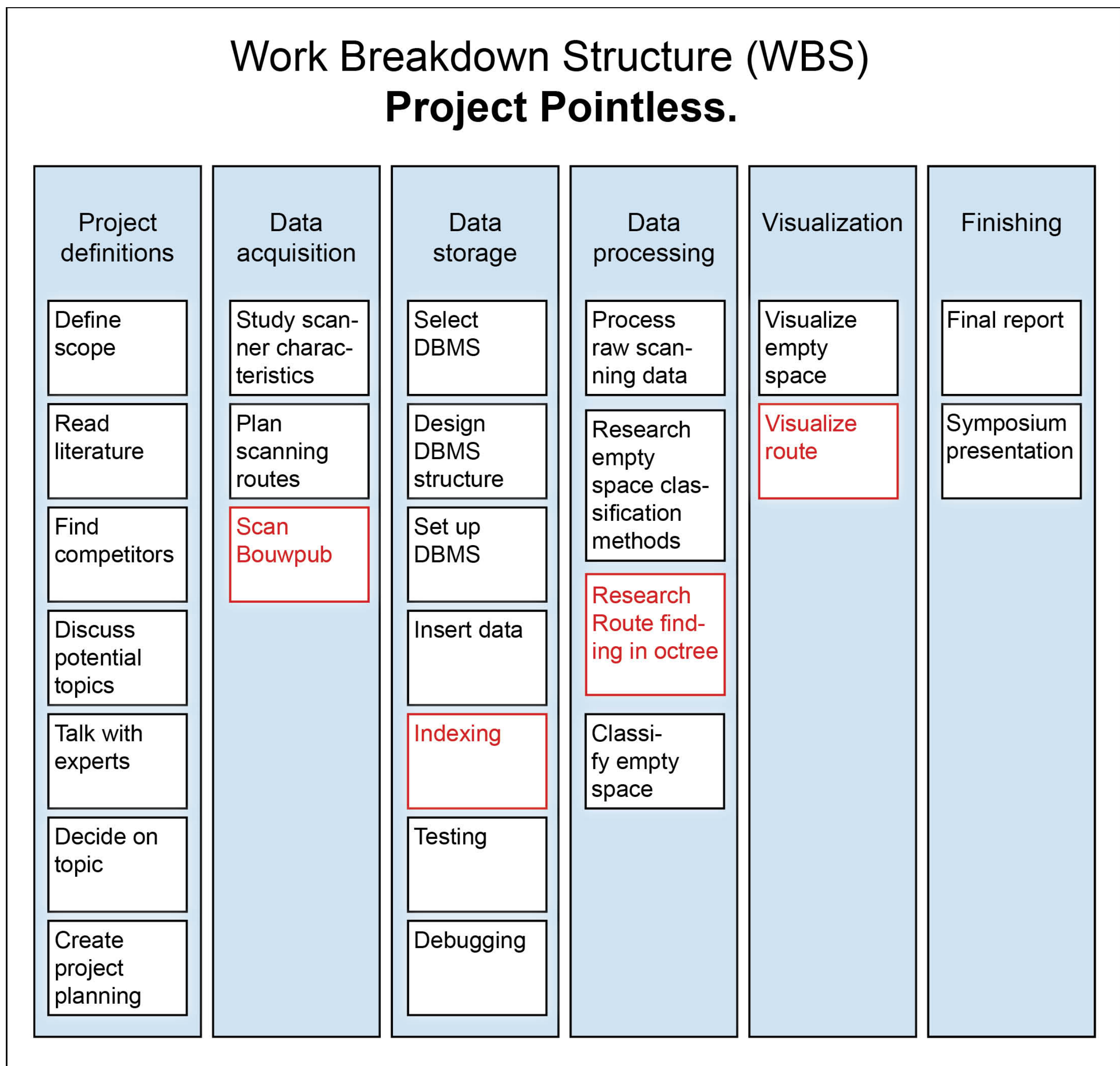

Figure 11: Work Breakdown Structure

Figure 12: Project Pointless' twitter page

or even retweeted us. The goal was to not only promote the team's project, but also the study. The Website was used to add more content for interested people we contacted throughout the project. After a couple weeks the time was found to implement Google analytics into the website to get more information about the visitors. Figure 13 shows that throughout the project there is steady stream of visitors with occasional spikes. In total the website has been requested 235 times over a period of 5 weeks. This means that on average the website has been requested 47 times per week. Table 1 also shows that the website has been requested from many different countries all over the world. Besides the Netherlands, viewers were regularly recorded from Austria and the United States. In total people from 12 different countries have viewed the website. Many of the international visits can be explained through contacts the group members had with others in the field. Every time one of the members contacted other researchers they also included a link to the website to show the project and the background of the questions. The contacted persons were then able to follow the process, which usually happened as more visits of the website from those locations took place.

]

Table 1: Top 5 measured viewer locations of Projectpointless.bitballoon.com since September 22nd 2015

Location	Views
Netherlands	92
Austria	20
United State	10
Germany	4
Spain	3

Besides maintaining the website and tweeting about progress, some videos have also been published

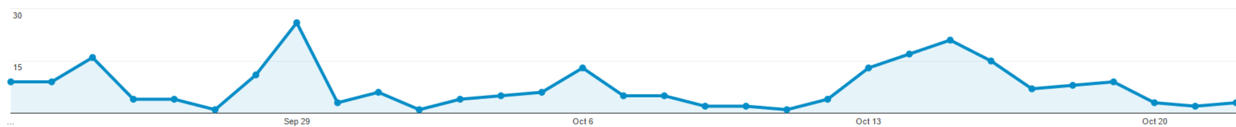

Figure 13: Graph showing the amount of views of Projectpointless.bitballoon.com since September 22nd 2015

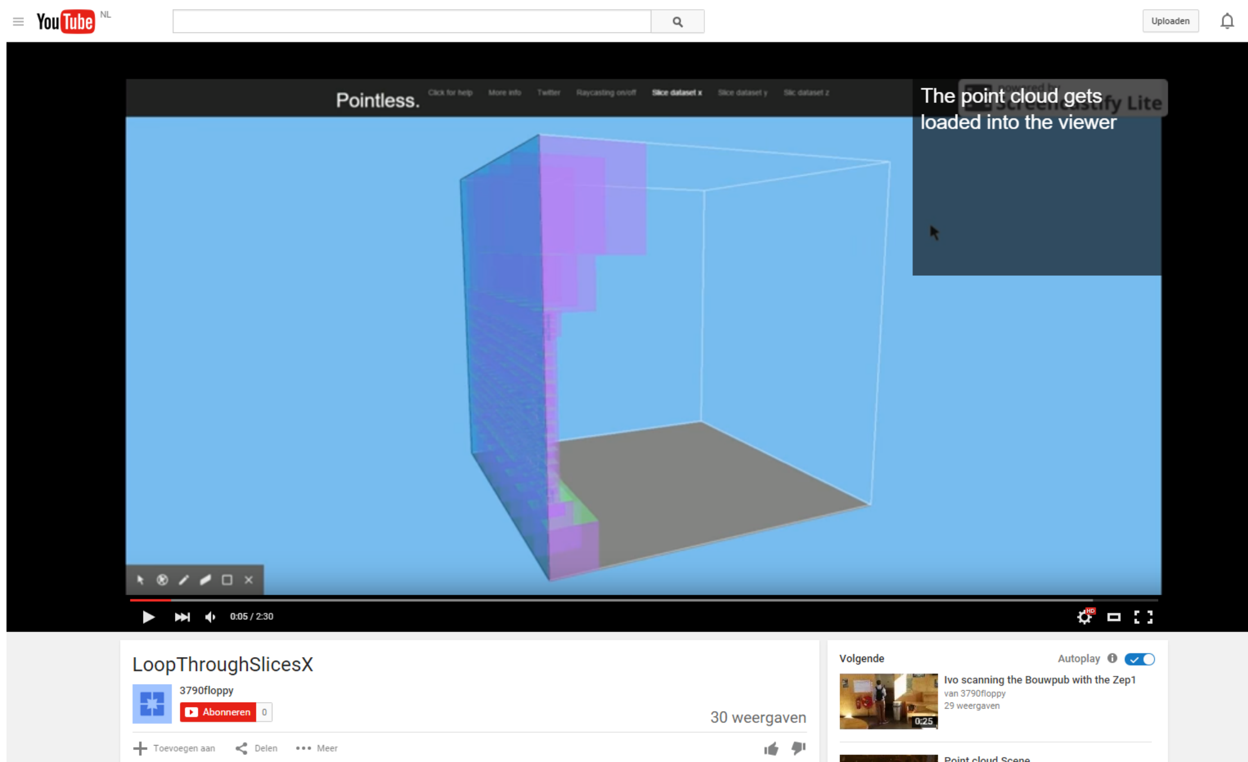

Figure 14: Videos posted on YouTube showing progress of Project Pointless

on YouTube. The kind of videos range from presentation recordings over a video of the team scanning the Bouwpub to a preview of the preliminary results (figure 14). At the time of writing this report the videos had around 30 views on average even though they were not listed and could only be accessed by people knowing the link.

The media outreach strategy of Project Pointless has also attracted the attention of the news channel of the faculty of architecture at TU Delft. They invited the team for an interview in which the team explained what the project is about and some results were presented. This interview led to the publication of a small article in the BK City newsletter (figure 15) with a link to the point cloud that Project Pointless has published online.

The code of Project Pointless is freely available online on GitHub (https://github.com/ivodeliefde/ProjectPointless) (figure 16).

TU Delft Delft University of Technology

Webmail | Webdata | Weblogin | A-Z index | Login | English | Contact

zoek

onderwerp persoon

Home | 3mE | BK | CiTG | EWI | IO | LR | TBM | TNW

Nieuws BK

Nieuwsbrief

Archief

Home > BK > Nieuws BK > Elke dag de hele dag in de Bouwpub? Het kan.

Elke dag de hele dag in de Bouwpub? Het kan.

01 oktober 2015 door *Communication BK*

Geomatics projectgroep Pointless heeft met behulp van een 3D-laserscanner het interieur van de Bouwpub gemodelleerd met een tof 3D-model als resultaat. Neem dus een kijkje!

Betreedt de virtuele wereld van de Bouwpub door Ivo de Liefde, Florian Fichtner, Olivier Rodenberg, Erik Heeres & Tom Broersen hier.
Let op: zoomen en draaien!

Vragen, suggesties of opmerkingen: Mail de webredactie

© 2015 TU Delft

Figure 15: Short publication in BK City News of the preliminary results of Project Pointless

32 commits | 2 branches | 0 releases | 1 contributor

Branch: master ▾ **ProjectPointless** / +

ivodeliefde resize images using html — Latest commit 6d7aa33 2 days ago

images	added load filtered points	3 days ago
static	added load filtered points	3 days ago
templates	added load filtered points	3 days ago
uploads	remove point cloud	11 days ago
ConverterInterface.py	added load filtered points	3 days ago
PointlessConverter.py	added load filtered points	3 days ago
ViewerInterface.py	added load filtered points	3 days ago
__init__.py	first commit!	11 days ago
astar.py	debugged routefinding and neighbour finding	4 days ago
config.py	added neighbourfinding and route calculations	8 days ago
neighbourFinding.py	debugged routefinding and neighbour finding	4 days ago
readme.md	resize images using html	2 days ago
scalingStatistics.txt	debugged volume calculations	8 days ago
volumeCalc.py	debugged volume calculations	8 days ago
volumeStats.txt	debugged volume calculations	8 days ago

Code

Issues 0

Pull requests 0

Pulse

Graphs

HTTPS clone URL

https://github.cor

You can clone with HTTPS or Subversion.

Clone in Desktop

Download ZIP

readme.md

Project Pointless

- What is Project Pointless?
- How does it work?
- How to use the Pointless application
- Emtpy Space Octree
- Pointlessconverter
- Example Use Cases

Figure 16: GitHub repository of Project Pointless

5 Concepts

This chapter will describe the concept of the octree structure created in this project (chapter 5). Additionally, it describes the methods that have been considered for the extraction of the empty space from this structure (chapter 5.1.1). The different options for database management systems required to efficiently retrieve the data are described in chapter 5.3. At the end the concept for visualising the results are discussed in section 5.4.

5.1 Octree Structure of Point Cloud

The octree data structure was chosen to be used for storing the empty space and the point cloud of Project Pointless in a database management system. In the following paragraphs the background, alternative options and the setup of this structure will be explained in detail.

5.1.1 Empty space classification and clustering method

The project team researched different options to find the empty space: besides the octree approach also the use of scanner path and voxel (3D grid cell) based classification have been reviewed. All methods will be explained here and also the arguments for choosing the octree structure over the other approaches will be presented before the octree will be explained in further detail.

The voxel approach takes the bounding box of the point cloud and voxelises the entire space. The points in the point cloud are then intersected with the voxels to find the points inside each voxel. The voxels which do not contain any points after the procedure make up the empty space and thus get this as an attribute. A similar approach was used by Bienert et al. (2010) or by Nourian and Zlatanova (2015) to voxelise a point cloud.

Another method is to make use of the path the scanner travelled while scanning the point cloud. The direct connection between the scanner route, so the point from where the beam was sent, to the point in the cloud, so the point of reflection, can be seen as empty space. This method is also called ray tracing. Turner and Zakhor (2013) are using this approach stating that "if a laser passes through a voxel, that voxel is considered interior space". Carving means that a voxel is labelled empty space or occupied space accordingly. The following figure (17) describes the process in three steps:

An octree is a three dimensional extension of a region quad tree data structure. It consist out of a cubical volume and is recursively subdivided "into eight congruent disjoint cubes (called octants) until blocks of a uniform colour are obtained, or a predetermined level of decomposition is reached" (Samet, 1989). Octrees for point clouds are based on the recursive and non-uniform subdivision of space and are a common approach to structure and segment 3D point clouds. They are used for the partitioning of space and result in a hierarchical structure. Earlier research, like Wang and Tseng

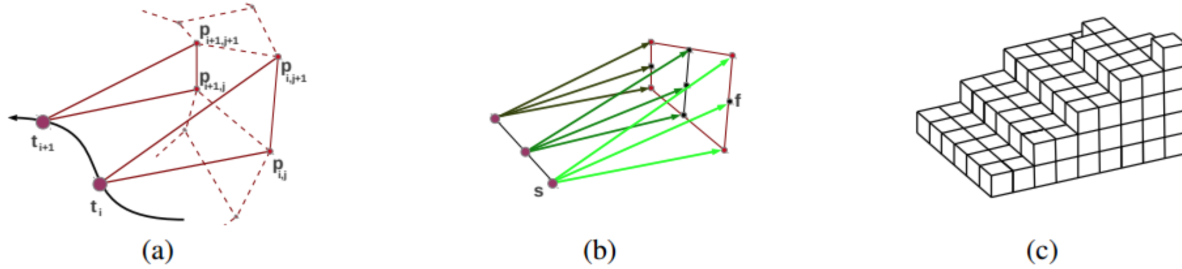

Figure 17: (a) The solid red lines show the scan lines and the dashed lines the spatial and temporal interpolations between them, the purple circles (t) stand for the track samples at the scanner's locations; (b) With ray-tracing carving is performed; (c) The result is a set of voxels labelled as empty space. Image courtesy to Turner and Zakhor (2013)

(2011), Zhou et al. (2011) or Schön et al. (2013) used an octree structure to segment point clouds.

All three options have advantages and disadvantages depending on the application and the available data. The goal of Project Pointless is being able to find the empty space of any point cloud. Not all point clouds, however, include the scanner's route or location or they do this in an insufficient way, being 2D only. This makes the use of the path of scanner method unfavourable to calculate the empty space. Furthermore it is estimated that intersections of a line and a voxel are more expensive than with a point only. On top of that a regular voxel grid is significantly disadvantageous in terms of quality and storage space compared to a flexible octree structure, which also rules out the voxel based classification option. The octree approach on the other hand includes efficient indexing and are effective for neighbourhood operations (Burrough, 1986). The representation of curved objects can be done more cheaply with this approach as well. Additionally it leaves the possibility open to use different levels of details, which is advantageous for rendering (Koo and Shin, 2005) or even calculations. However, the path of the scanner might be a useful validation method and could separate interior from exterior empty space inside the octree structure. This is further discussed as future research topic in chapter 9.8.

To create the octree, it is not necessary to store the entire octree structure explicitly. It is generally sufficient to store only the so-called materialised path or location code for every leaf node. Two methods for creating location codes have been reviewed: the approach where location codes are based on the tree depth and the approach where the location code is based on space filling curves.

Koo and Shin (2005) present a number of methods in which space filling curves are used to store the location of a three dimensional leaf node in a one dimensional array. In this approach each leaf node that contains a cluster of points is stored using a space filling curve. Space filling curves "graphically express a mapping between one-dimensional values and the coordinates of points, regarded as multi-dimensional values, since the points are placed in a sequence according to the order in which the curve passes through them" (Lawder, 2000). There are many different space filling curves, but a few of them have a special feature called 'bitwise interleaving'. This means that the binary numbers of three dimensional coordinates can be interleaved to form the binary number that represents the

Figure 18: Morton order space filling curve

position of a particular coordinate in the one dimensional sequence (figure 18). Campbell et al. (2003) have researched the efficiency of three space filling curves: the Morton curve, the Hilbert curve and the Gray curve. The Morton curve also allows for bitwise interleaving, which not only ensures that “coordinates which are close to each other in the N-dimensional space have Morton numbers which are close to each other too” (Baert, 2013), but also that the N-dimensional order as well as the exact location are stored inside a single Morton number. This makes the Morton curve (figure 19) very powerful. However, Campbell et al. (2003) has performed a number of benchmark tests in which their Hilbert curve implementation (figure 20) scored significantly better than the Morton curve and the Gray curve. The outcome of this research has to do with the fact that each space filling curve has a number of geometrical properties which influence its efficiency in range queries: “Hilbert ordering generally achieves the best resulting partitions, likely due to the continuous nature of the Hilbert curve” (Campbell et al., 2003).

However, the second method for storing the location code has been selected because it allows for easy neighbour finding (Burrough, 1986, Vörös, 2000). This is relevant for the use case of Project Pointless, which is to calculate the shortest path in an octree. The second method will be briefly described here.

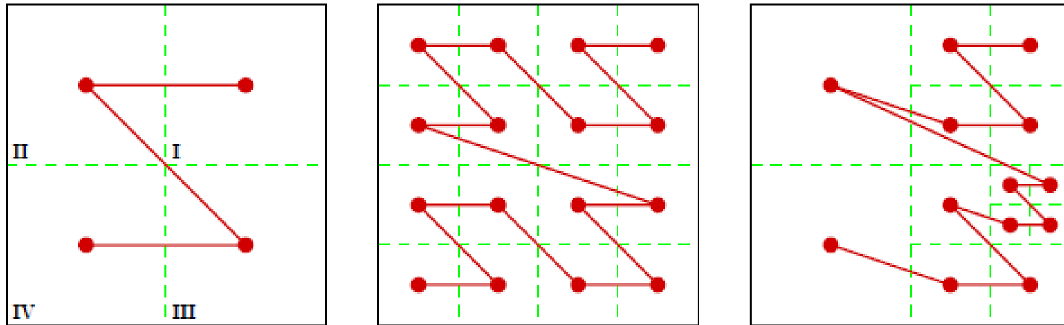

Figure 19: Morton order space filling curve in a quad tree structure

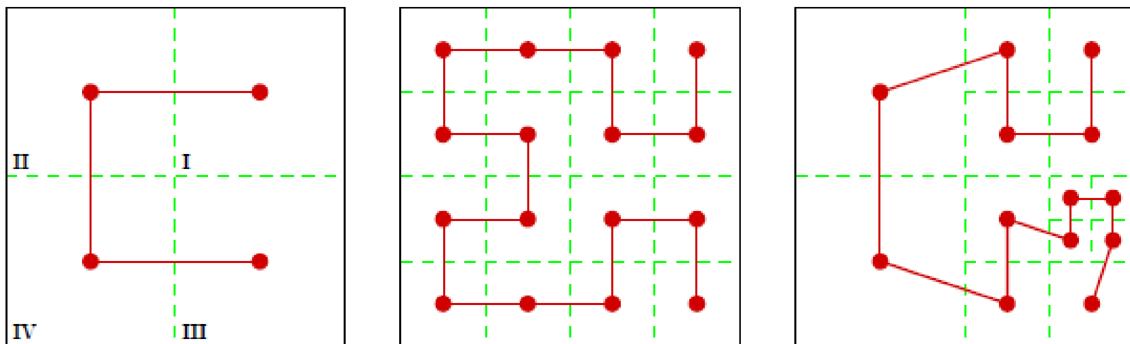

Figure 20: Hilbert order space filling curve a quad tree structure

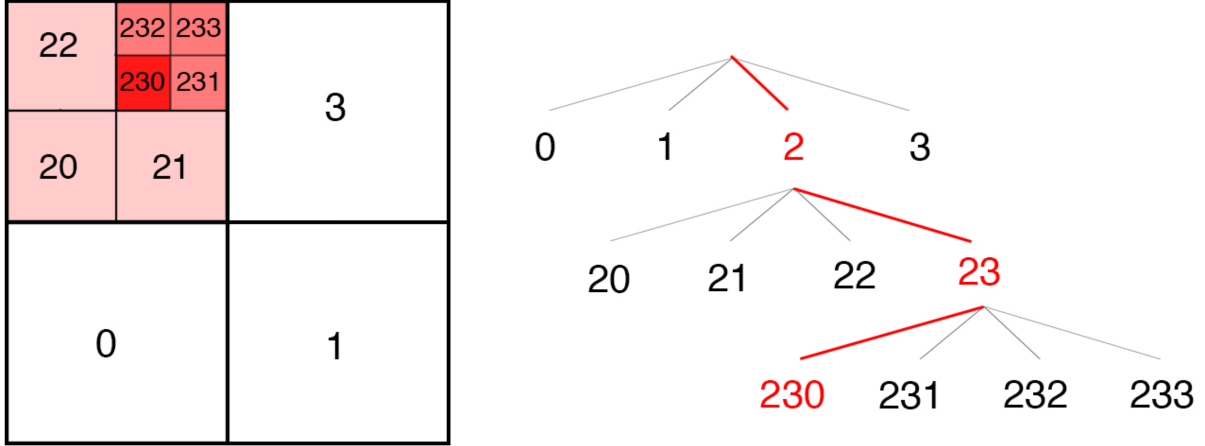

Figure 21: Quad tree structure

The preferred method uses the tree depth to store the leaf nodes location. Every point in the point cloud will have one materialised path assigned to it. This materialised path refers to the corresponding leaf node it is located in. This octree structure is thus a linear octree. Materialised paths are built by traversing the tree from root node to leaf node, thereby adding the name of the node where the point is located to a string for every level. This means that every level in the octree will be represented by a single digit in the eventual materialised path. This path shows which steps needs to be followed through the tree from root to leaf node in order to find the leaf node containing a specific point. Figure 21 shows this principle of using materialised paths for a quad tree structure. For an exemplary node (0220), it follows from this that the materialised path goes from 0 to 02, and 022 which are the ancestor nodes of this particular leaf node. Since the root node is always the same (0), this can be exempted from the resulting materialised path. The principle is the same for an octree, except that the cells will be enumerated from 0 to 7. Enumeration of the cells in the octree is done in a z-order, as displayed in figure 22. To obtain maximum detail of the empty cells in our octree, full leaf nodes are forced to always split to the maximum level. There are thus no non-empty leaf nodes in the tree except for the maximum level.

5.1.2 Binary Masking

To locate leaf nodes inside the octree structure, a materialised path needs to be created. There are three kinds of nodes in an octree structure: white nodes which are empty nodes and therefore don not require further splitting and black nodes which contain at least one point but are at the maximum level and therefore also don't require further splitting. These are the leaf nodes (they are at the end of the tree). The third kind of node is the grey node, which is node that contains at least one point but is not yet at the maximum level and therefore needs to be split into eight octants. Grey nodes are so called non leaf nodes (Samet, 2006). Project pointless aims to derive the materialised path of the black nodes and uses them to find the empty white nodes.

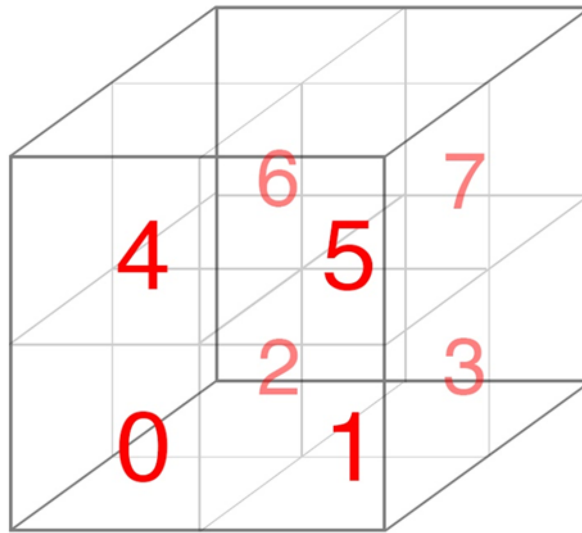

Figure 22: Enumeration principle

y

10

10 = 2	11 = 3
00 = 0	01 = 1

00

00 01 x

Figure 23: Binary conversion

Finding the materialised paths is done using a method called binary masking. The location of a cell can be encoded with only 3 bits for every level in the tree. For visualisation purposes a two dimensional quad tree will be used as running example in this report, however for the project a three dimensional octree was implemented. The same principles apply for the quad tree example as for the octree implementation.

Every cell containing points in the quad tree is split into four smaller equal cells. Thus, in this cell, along every dimension a leaf node can be located at one of the two sides of the split line (either to the left or to the right), which can be indicated by either a 0 (true, above/right of the split line) or a 1 (false, below/left of the split line). In three-dimensional space you can thus indicate the cell you are in (at that level in the octree) by a combination of three bits; a 0 or 1 for x, 0 or 1 for y, and 0 or 1 for z-direction. This an efficient way to determine quickly at every level in the octree in which (gray) node a point is located. The procedure followed to determine the materialised path of the leaf node for every point is displayed in the following chapter (5.1.3) and figures. First a conversion of the coordinates of the point into binary numbers for x,y, and z takes place. Then every digit in these binary numbers will be iteratively compared to a binary mask, the form of which depends on the level of the position in the octree. For level 1 (and binary x, y coordinates of [101,011]), the binary mask will be 100 (so only the first digit is evaluated). Iteratively comparing the digits in the (x,y) coordinates to the binary mask using a Boolean AND operator, will result in the following outcomes: True for x, and False for y. Then, if x is true on a specific level, a 1 is added to the path number for that level. If y is true on a specific level, a 2 is added to the path number of that level. The addition of these will lead to the number of the cell the point is located in at the current level in the tree. In a quad tree this leads to 4 combinations:

- When both x and y are false: $0 + 0 = 0$
- When only x is true: $1 + 0 = 1$
- When only y is true: $0 + 2 = 2$
- When both x and y are true: $1 + 2 = 3$

This procedure is repeated for every level in the tree, every time concatenating the cell numbers to form the materialised path.

5.1.3 Example

In the following example the path of a single point in a quad tree will be calculated. First the box coordinates are converted to a binary number. X has a value of 5 which is 101 in binary, y has value of 3 which is 011 in binary. These x and y values will be used to compare the binary mask to for each level.

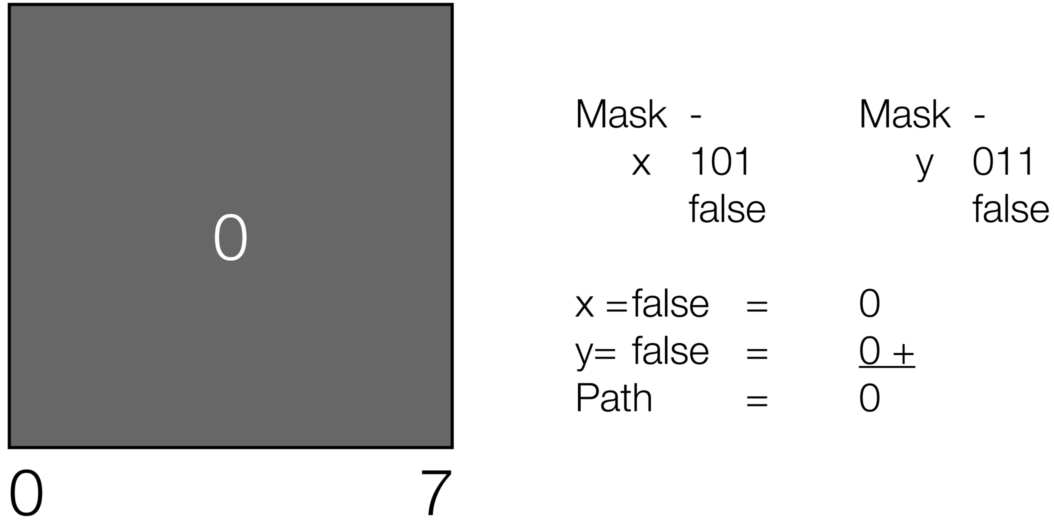

Figure 24: Root level of quadtree, no digit in materialised path yet

Depth 0 In the root level (depth 0) there is no split, so masking the coordinates is not possible (figure 24). This results in a false for both for x and y for the Boolean comparison with the mask and the coordinate, a false result will add zero to the path number. So the result of the 0th depth is 0. To reduce storage space the path number of the root will not be stored.

Depth 1 In depth 1 (figure 25) the root level is split into four equal boxes. A mask with the value of the split line is used to check if the coordinates are right or left (for the x coordinate) or above or under (for the y coordinate) the split line. In this example this is four, which is 100 in binary. So if the x or y coordinate has a value of more than 4 than it is to the right or above of the split line. The mask for a depth of 1 results in a true for x and false for y. X is true, therefore 1 is added to the path. Y is false so 0 is added. This results in path number of 1 for the first level.

Depth 2 In depth 2 (figure 26), the box with path number 1 is split into four equal boxes. In the previous step, the coordinates are checked if they are below or above the centre of the domain of the octree. In the next depth the coordinates are checked if they are higher or lower than half of the domain of the box that was selected in the previous step. Since half the domain of this box is a quarter of the entire domain the number that needs to be checked will be the half of the number checked in depth 1. In this example, the domain of 8 is used. Because binary numbers are used it is easy to find the number half of an even integer, this is the number that is situated one digit to the right of it. For example, half of 4 is 2, the binary number of 4 is 100 and the binary number of 2 is 010. In this case the split line in depth 1 is 4 so the mask with a binary number of 010 (decimal

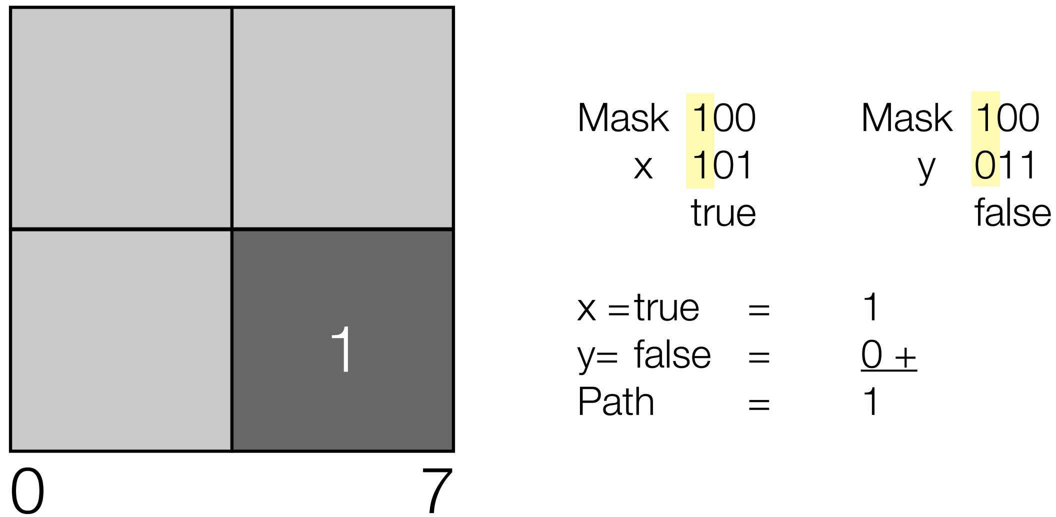

Figure 25: First digit of materialised path

number 2) is used. The mask for a depth of 2 results in a false for x and true for y. X is false, therefore 0 is added to the path, y is true thus 2 is added, this results in path number of 2 (0 + 2). The concatenated path number so far is 12.

Depth 3 In depth 3 (figure 27) the box is split, with path number 12, into four equal boxes. Next step is to check for half the mask number of depth 2, this number is a decimal number of 2/1=1. So the mask in depth 3 is 001. The mask for a depth of 3 results in a true for x and true for y. X is true so 1 is added to the path and y is true thus 2 is added. This results in path number of 3. The concatenated path number is 123.

5.1.4 Translation and Scaling

For the above procedure to work, the point cloud first needs to be translated and scaled. The translation is necessary to define the lower left corner of the front of the bounding box as the origin of the point cloud. The scaling is needed so that our code fits to any kind and size of point cloud, and so that every cell which we define has coordinates represented by integers between 0 and 2n. Then by removing the decimals from every point's coordinates, we can snap these points to the particular cell it is located in.

The translation happens for every dimension separately by taking the smallest coordinate of the bounding box in a given dimension. If this coordinate is lower than zero, the scaling factor is the

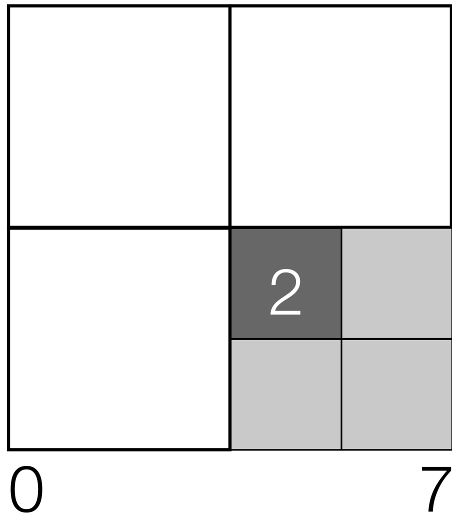

Figure 26: Second digit of materialised path

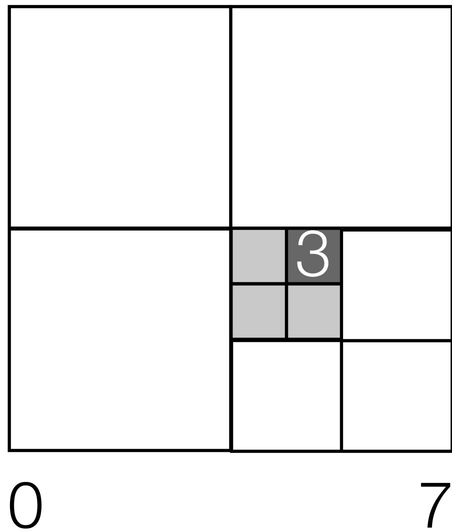

Figure 27: Third digit of materialised path

absolute value of the coordinate. For example, if the minimum x coordinate is -20 the entire point cloud needs to be translated +20 in the x direction to have its minimum coordinate positioned at the origin of the axis. If the minimum coordinate is above 0 the translation factor has to be 0 - minimum coordinate to have its minimum coordinate positioned at the origin of the axis. If the minimum coordinate equals 0 no translation is necessary and the translation factor is set to 0. Translation should take place before scaling of the point cloud. If not, the result will be distorted.

The scaling is done by defining a scaling factor. This scaling factor is calculated by comparing the dimensions of bounding box to the required dimensions. The length of the bounding box is calculated for every dimension first. Then it is checked if all of these dimensions equal 2^n, where n represents the maximum number of levels in the octree. If this is the case the scaling is not necessary and the scaling factor is set to 1. However, if this is not the case it is checked what the maximum domain of the bounding box is. This length is used to calculate the scaling factor by dividing the preferred dimension of 2n by the maximum domain. This scaling factor can then be applied to all dimensions to obtain the scaled point cloud.

For the proposed octree procedure to work, it is necessary for the bounding box of the point cloud to be a cube, however this is not always the case after the scaling and translation has been applied. To create a perfect cube of the bounding box, we simply take the largest domain, and use this length for every domain. In essence, we thus add empty space to the dataset in all dimensions smaller than the largest dimension.

The order of computations should be as follows:

1. Compute translation and scaling factors for the entire point cloud
2. Apply translation factors to the points
3. Apply scaling factor to the points
4. Remove decimals from the points to snap to a cell
5. Use binary masking to determine the materialised path

5.2 Empty Space Identification

Section 5.1 explains how the point cloud is structured into an octree, thereby obtaining a list of materialised paths of all the non-empty nodes. However, the octree does not explicitly store the empty nodes. This section will accordingly explain how the empty nodes can be generated from the list of materialised paths of non-empty nodes, and how the geometry of these nodes can subsequently be derived.

5.2.1 Finding the Empty Nodes

To reconstruct the empty nodes in the octree structure, the following method was used. Empty nodes can occur in every level in the octree. To find all the empty nodes at a certain level in the octree, we take the list of all non-empty nodes (for example [0123, 1231, 4215, 6432] in case of an octree which goes down to level 4). Then, to identify the empty nodes at a certain level n, we take the following steps:

1. Generate a list of all parent nodes at level n-1
2. Generate a list of all non-empty child nodes at level n
3. Concatenate to all parent nodes at level n-1, one by one every number in range 0 to 7, thus generating a list of all potential empty and non-empty child nodes at level n.
4. Test for all nodes in the list of potential empty and non-empty child nodes at level n, whether these child nodes are present in the list of non-empty child nodes at level n. If they are not present in this list, then this means the child node is an empty node.

These steps have to be performed for every level of nodes in the octree. Generating the lists of parent and non-empty child nodes at a certain level is simple by just removing parts of the string of every leaf node until the required level is reached. Duplicates have to be removed from the resulting lists. To find all empty nodes at level 0, simply compare the list of nodes at level 0 to a list of numbers in range 0 to 7.

The above described procedure has the advantage that combinations are only generated for gray nodes. Gray nodes include black (full) and white (empty) nodes in deeper levels of the octree. This avoids also making combinations for white nodes, which would be obsolete, since white nodes will never have children in our approach, and are thus leaf nodes. The disadvantage is that the method also generates combinations which are in fact non-empty nodes, thus a sizable part of the generated nodes are not used in the end.

5.2.2 Finding the Geometry of Empty Nodes

Size of the nodes It is known that all sides of the nodes have the same length, and that this length is dependent on the level of the node in the tree, and is halved for every deeper level in the tree. Since it is also known that the minimum size of the leaf nodes equals 1, it can be said that the size of the nodes at each level, except for the deepest level, is equal to 2^n (where n is computed by taking the difference between the current level and the maximum number of levels in the tree).

Origin point of the nodes Section 5.1.2 of this report explains how to use binary masking to convert from coordinates to materialised paths. The same binary masking technique can also be

used to find the origin point of the nodes. Instead of converting from coordinates to materialised paths, it is done the other way around. Materialised paths are converted to coordinates. This can be done by iterating over the numbers in the materialised path, and for every number adding the corresponding binary numbers to separate x, y, and z strings. At the end of the iteration, these binary numbers can be converted to obtain the (x,y,z) coordinates in decimal numbers.

5.3 Database management system

For the decision which database management system (DBMS) is most suitable for Project Pointless a number of systems have been analysed. Besides PostgreSQL, also Oracle, MonetDB and MySQL were taken into consideration. All these DBMSs use the Structured Query Language (SQL). In conclusion PostgreSQL was chosen because it was taught in GEO1006: GEO-DBMS and because of the extension PostGIS, the team was expecting to use. In the end this extension has not yet been applied as the 3D implementations did not give the desired results. Future research could look into the use of PostGIS for creating point cloud octree structures, which allow for easy integration into GIS (see chapter 9.7).

5.4 Visualisation

After the points and the empty spaces are stored in the database it is possible to use the data for many kinds of applications. However, it is useful to have a tool to visually inspect the data before using it to verify whether the conversion was successful. To do this a basic viewer has been implemented that can retrieve data from the octree in the database and visualise it. For this part of the application WebGL has been used to make sure that no additional plugins are necessary. The different parts of the application will be explained in the following paragraphs.

5.4.1 Three.js

Different available methods for the visualisation of the empty space have been discussed: Potree (`http://potree.org/`), PCL point cloud viewer (`http://pointclouds.org/documentation/tutorials/cloud_viewer.php`) and mirage online PointCloudViz (`http://www.pointcloudviz.com/`). However, in the end it was decided to create a custom point cloud viewer using Three.js. This decision was made because there was only limited time available in the planning to make the researched viewers suit the functionality of Project Pointless. The examples in Three.js (Threejs, 2015) showed that it was relatively easy and fast to create a basic viewer. This was faster to implement than to adapting the existing code from the other viewers.

Three.js is an open source JavaScript library for visualising three dimensional scenes in the browser. Its goal is to make WebGL in the browser easy to use (ThreeJS, 2015). WebGL “is a cross-platform, royalty-free API used to create 3D graphics in a Web browser” (WebGL, 2011). It runs inside

HTML5 canvas element and does not require any additional plugins.

Visualisation with Three.js is done in two steps. First a scene needs to be created. In this scene a camera is placed together with objects. The camera represents the perspective of the user inside the scene. The objects are all the visible geometries inside the scene. Additionally lights can be added, for example to create directional light (a light shining in a certain direction and therefore casting shadows) or to manually change parameters like colour or intensity.

The second step is to render the scene. This is the process of visualising the defined scene in the HTML canvas. The rendering function is usually a loop that updates the visualised scene when the user interacts with the controls. How often the scene is being 're-drawn' can be defined here, together with other rendering parameters.

5.4.2 FLASK

Three.js is a method for visualising three dimensional objects in the browser. Therefore, a website had to be created which runs Three.js and connects it to the database with the empty space and the point cloud. Since the scripts to create, store and retrieve the octree structure have been written in Python the Flask framework has been selected for the backend. Flask "is a micro framework for Python" (Flask, 2014). It allows to render HTML templates which connect to Python scripts and static files such as CSS stylesheets and JavaScript.

Flask also offers a routing system to direct user to different parts of the website. Inside Flask dependencies can be added. This can be used to establish a connection to a database management system. With the Flask-SQLAlchemy package the backend can create a connection with the PostgreSQL database. The Three.js script retrieves data from the PostgreSQL database by making an AJAX Post request to the Flask backend, the routing system then calls a function to query the database. The resulting table is returned to the script as a JSON object.

5.4.3 Web Workers

JavaScript in the browser runs synchronously. This means that when one part of the code is running the other parts have to wait for it to finish before they can continue. This can influence the performance of a website, especially when certain functions are often (or continuously) running in the background. Web workers are a JavaScript that runs in the background "independently of other scripts, without affecting the performance of the page" (W3Schools, 2015).

Loading large amounts of geometries in the browser has a big influence on the performance. While the request has been sent to the database the scene rendered in Three.js stops updating. Only when all the data is returned the scene can be updated again. Depending on the amount of data that needs to be retrieved this can take a couple of seconds up to a number of minutes. During this time the controls are not responding. To overcome this problem all the geometries are loaded from the

database using a web worker script. This script is always running in the background and listening for a button 'click'-event. While the web worker is waiting for the backend to return the database table the controls are still responding, because it runs independently from the main JavaScript.

6 Implementation

So far the concept of the octree based structuring of empty space and its requirements were explained. But nevertheless the only way to really proof a concept is to actually implement it and show that it works in practice as it was planned in theory. The Geomatics Synthesis Project's goal was not only to connect the subjects and knowledges learned throughout the MSc program, but more than that also the possibility for students to experience a whole workflow when working with point clouds. After research this also includes practical work and implementation. First of all a point cloud has to be collected, then it needs to be processed and the results have to be presented and published. The following chapters will focus on the latter two and give an insight of the design of workflow, interface and engineering and will then highlight the system characteristics and the exploitation plan of the results.

6.1 Workflow Design

The workflow diagram (figure 28) shows the place of the different components within the project. The input is the data which the user has to provide and the result is the visualisation of the empty space. In paragraph 6.4 the different components will be explained in more detail.

The input consists out of the point cloud, the number of levels of the octree the user wants to have and the PostgreSQL credentials. The levels of the octree are limited to 8 at this moment, but this can easily be changed in the code. The reason for this limitations is the performance of the visualisation which decreases with more details. Next the input gets validated, so an error gets raised as soon as the file format of the point cloud cannot be read for example. Right now the format las-file is required. Subsequently the point cloud gets automatically translated and scaled to the coordinated system used before a linear octree and each point along with its materialised path gets written into the database. Afterwards the empty leaf nodes can be derived from the octree which will be written to the database as well. They will also be the output of the workflow and can be visualised.

6.2 Data Acquisition

The original plan was to scan the interior of the entire faculty of architecture with the Zeb1 laser scanner of Leap3D. However, due to miscommunication about the tariffs, the scanning cost estimation has significantly exceeded the budgets of €3000,-. It has therefore been decided to scan only the Bouwpub, which has a much smaller area than the entire faculty. The Bouwpub is the faculty's pub and located in a rectangular house just next to the main building. This way the variable costs, to be paid for every meter scanned, went down significantly. The cost of renting the Zeb1 was not lower, even though the scan took much less time, the device still had to be rented for the whole day. To make use of that time additional rooms, like the Geolab and the Berlagezaal were scanned as well. The actual scanning was easy and took only 15 minutes for the Bouwpub with a one hour

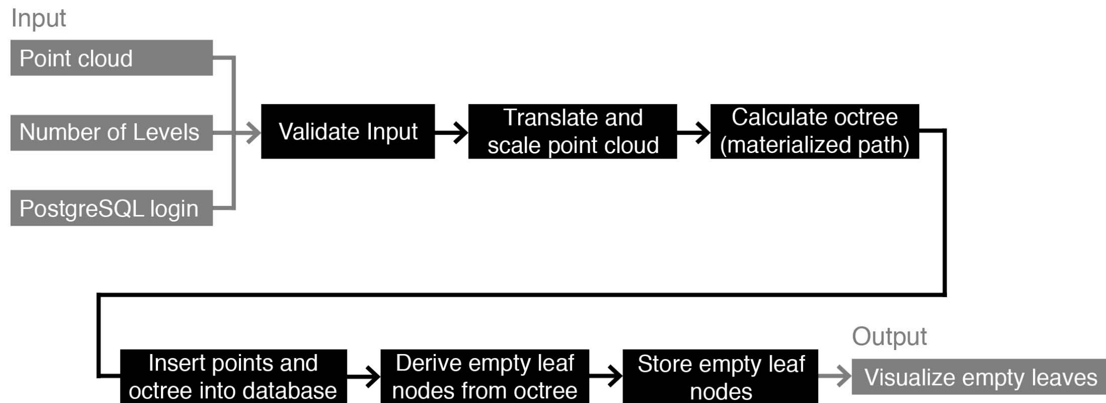

Figure 28: Workflow diagram

preparation time to get acquainted with the Zeb1 laser scanner when the user does it for the first time. The results of the scanning were good. However, due to open windows some noise had to be removed (objects outside the building were scanned as well). The unfiltered point cloud included more than 25 million points and the total cost of using this scanner was €363,-. For most of the further processing a filtered version of this point cloud was used.

Because the costs for using the scanner of Leap3D are significantly less than first anticipated (after downscaling to just the Bouwpub) there was the possibility to also use an alternative scanner as well, which was able to produce a coloured point cloud. The team used the Leica C10 Laser Scanner provided by the faculty of Civil Engineering. The only costs related for using this scanner is the loan of the student who helped with the scanning. Scanning the Bouwpub took approximately four hours and the costs were €120,-. With the Leica C10 laser scanner it was possible to capture coloured points, but it took more time to scan and therefore it was not possible to scan all the smaller rooms, which were scanned with the Zeb1.

Before scanning a total cost of €540,73 was estimated (see costs overview in appendix 11.2), so the actual costs of €483,- were lower than expected.

6.3 Interface Design

During the development of the Pointless converter and viewer it has been taken into account that there might be other potential users than just the project team itself. Therefore, a basic user interface has been created (figure 29). In this viewer the user can enter all parameters in a HTML form. When the submit button is clicked the octree structure will be automatically generated and stored in the database.

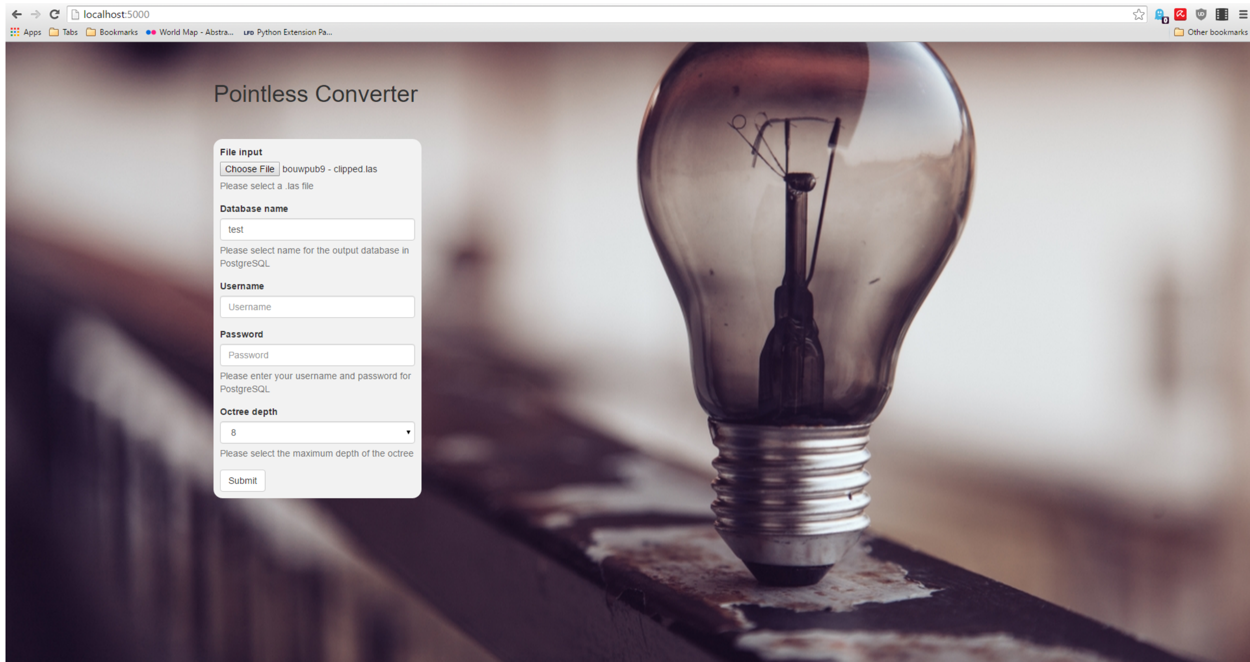

Figure 29: Pointless Converter interface

After the script has successfully finished, the viewer automatically opens. In this viewer a number of predefined queries can be performed in order to view the data in the database. The user can load slices of the dataset in any dimension (x,y,z) or calculate a path through the empty space (figure 30).

When opening pgAdmin the database the user can find the newly created database (figure 31). The table 'emptyspace' exists of all the coordinates, materialised paths and sizes of the empty leaf nodes. The table 'pointcloud' contains all the points and their coordinates and optionally also (r,g,b) values.

After the point cloud has been converted and stored in the database the viewer can also be opened by itself. For this a special interface has been made in which only the database name has to be entered together with the PostgreSQL login credentials (figure 32).

6.4 Engineering Design

Following the steps explained in the interface design a whole sequence of operations are triggered. This paragraph will explain the most important functions and reflect on the way they are implemented.

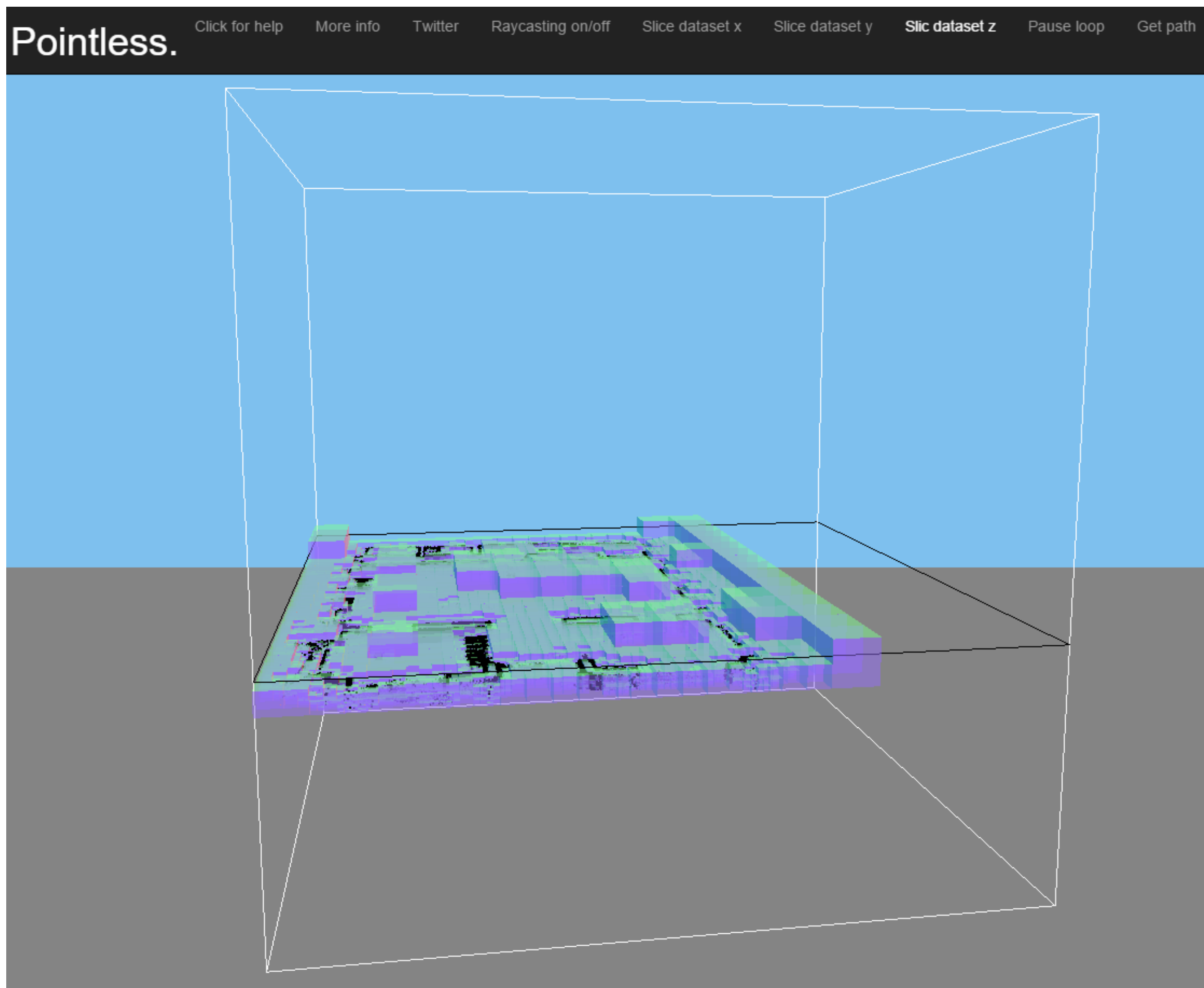

Figure 30: User interface with slice of dataset

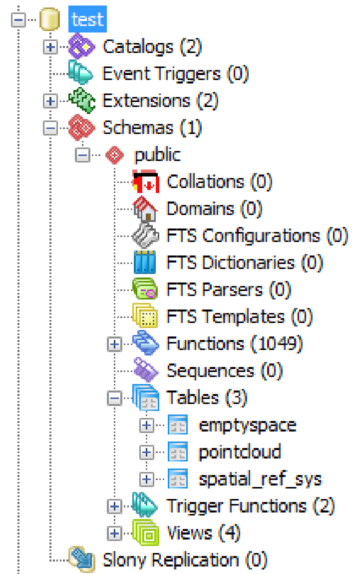

Figure 31: Newly created database in pgAdmin

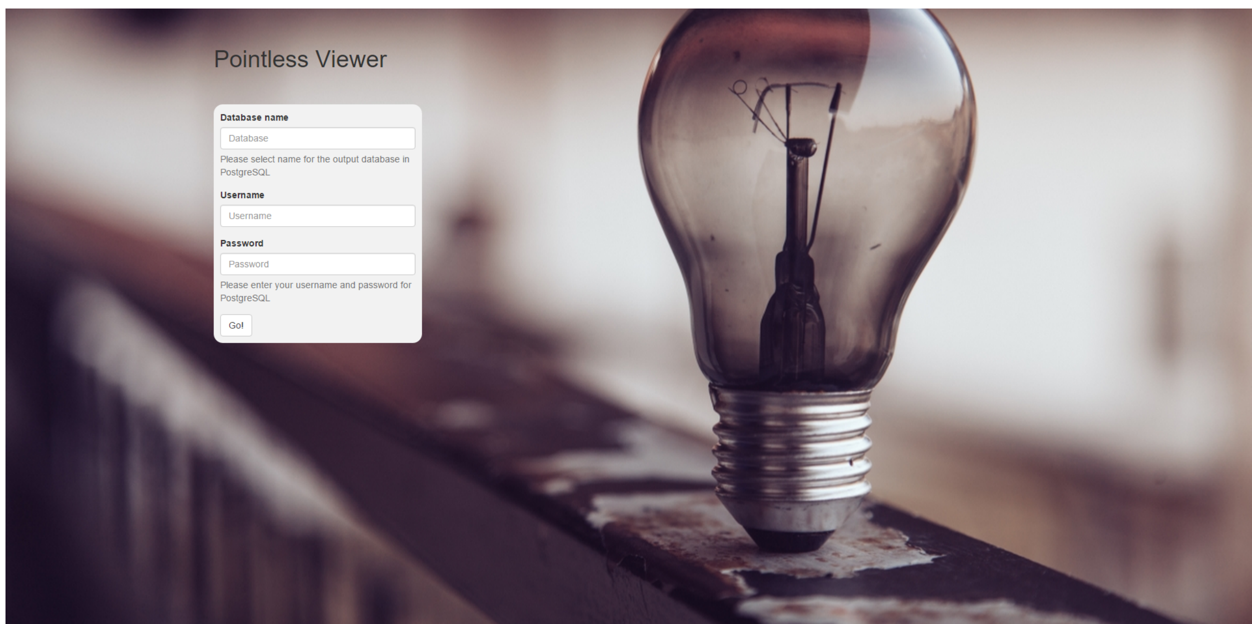

Figure 32: Pointless viewer interface

Table 2: convert data function

```
@app.route('/ConvertLas/', methods=['POST'])
def convertData():
if request.method == 'POST':
    filename = ""
    file = request.files['InputFile']
    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
dbms_name = request.form['InputDBMSname']
user = request.form['InputUsername']
password = request.form['InputPassword']
maximumLevels = int(request.form['InputDepth'])
if (user == "" and password == ""):
    Pointless(os.path.join(app.config['UPLOAD_FOLDER'], filename),
"1", dbms_name, maximumLevels)
elif (user == "" and password != ""):
    Pointless(os.path.join(app.config['UPLOAD_FOLDER'], filename),
"1", dbms_name, maximumLevels, "postgres", password)
elif (user != "" and password == ""):
    Pointless(os.path.join(app.config['UPLOAD_FOLDER'], filename),
"1", dbms_name, maximumLevels, user)
elif (user != "" and password != ""):
    Pointless(os.path.join(app.config['UPLOAD_FOLDER'], filename),
"1", dbms_name, maximumLevels, user, password)
return render_template('Viewer.html')
```

6.4.1 Major Components

The code of the Project Pointless implementation aims to put the concepts described in chapter 5 into practice in a compact and effective way. The three major components - user interface, calculating full nodes and calculating empty nodes - will be explained into detail.

Converter interface When the converter interface is opened in the browser an HTML form is visible. The submit button triggers the function convertData (table 2). In this function the point cloud is copied to the localhost and after that the parameters of the HTML form is retrieved and passed on to the Pointless script. After the script has finished it returns the viewer in which the data can be visually inspected.

Calculate the full leaf nodes The Pointless function is a function that calls a number of other functions (table 3). First it checks whether the input values of the viewer interface are valid in the function CheckInput. This function checks if it is possible to create a connection with the

Table 3: main function Pointless

```
def Pointless(lasFile, threshold="", dbms_name="", maximumLevels=8,
user="postgres", password=""):
    # This is the master file calling all the other functions based
on the user's input
    # Check if the user input is valid
    dbms_name, error = CheckInput(lasFile, dbms_name, user, password,
maximumLevels)
    if error == False:
        print "Start loading "+lasFile+" file:  " +
str(datetime.datetime.now())
    else:
        return
    # Call the different functions with the user's input parameters
    LasToOctree(maximumLevels, lasFile)
    create_dbms(dbms_name, user, password)
    write_dbms(dbms_name, "pointcloud", user, password)
    EmptyMaterialPaths = find_empty(call_dbms(dbms_name, user,
password, threshold), maximumLevels)
    emptyLeaf2DBMS(EmptyMaterialPaths, dbms_name, user, password,
maximumLevels)
    print "Finished writing '"+lasFile+"' to octree database
'"+dbms_name+"':  " +
str(datetime.datetime.now())
```

PostgreSQL login, it checks if the new database name already exists and if the point cloud is in the .las format.

If all the user's parameters have been approved the LasToOctree function is called. This function translates the point cloud to the origin and scales it to the appropriate length (table 4). Then it calculates the materialised paths and writes all the points to a temporary .csv file together with their corresponding materialised path. The function create_dbms creates a new database and the function write_dbms takes the temporary point cloud .csv file and copies it to the database. EmptyMaterialPaths takes the point cloud database table and uses it to derive the empty leaf nodes. The function emptyLeaf2DBMS stores the empty leaf nodes in the database as well.

The materialised path is calculated by the function getMaterialPath (table 5). This function iterates over every level of the octree and checks for every dimension in which part of the domain the coordinate is located. If a coordinate is in the lower half of the domain it gets a value of 0 in the binary coordinate. If the x coordinate is over half of the x domain it gets +1 in the binary coordinate, if the y coordinate is over half of the y domain it gets +2 in the binary coordinate and if the z coordinate is over half of the z domain it gets +4 in the binary coordinate. In other words, for every level 3 binary digits are added to the materialised path: one for x, one for y and one for z, corresponding to the binary values of 1 (=001), 2 (=010) and 4 (=100). Together 8 combinations

Table 4: las to octree converter function

```
def LasToOctree(depth, lasFile):
    # Open .Las file
    f = file.File(lasFile, mode='r')
    h = f.header
    print "Calculate BBox translation and scaling"
    # check bounding to calculate scaling factor
    lenX = h.max[0] - h.min[0] # length in x direction
    lenY = h.max[1] - h.min[1] # length in y direction
    lenZ = h.max[2] - h.min[2] # length in Z direction
    # Putting the BBox min coordinates at 0
    if (h.min[0] < 0):
        translateX = abs(h.min[0])
    elif (h.min[0] > 0):
        translateX = 0 - h.min[0]
    else:
        translateX = 0
    if (h.min[1] < 0):
        translateY = abs(h.min[1])
    elif (h.min[1] > 0):
        translateY = 0 - h.min[1]
    else:
        translateY = 0
    if (h.min[2] < 0):
        translateZ = abs(h.min[2])
    elif (h.min[0] > 0):
        translateZ = 0 - h.min[2]
    else:
        translateZ = 0
    if (lenX != 2**depth) or (lenY != 2**depth) or (lenZ !=
2**depth):
    # if scaling is required we take the maximum domain of the point
cloud for scaling
        maxDomain = max([lenX, lenY, lenZ])
        scale = 2**depth / maxDomain
    else:
        # if no scaling is required we set the scale to 1
        scale = 1
```

Table 5: function to calculate the materialised path

```
def getMaterialPath(depth, x, y, z):
    # define the binary function to get the material path of a leaf
node in an octree
    ocKey = ""
    for i in range(depth, 0, -1):
        digit = 0
        mask = 1 « (i-1)
        if (x & mask) != 0:
            digit += 1
        if (y & mask) != 0:
            digit += 2
        if (z & mask) != 0:
            digit += 4
        ocKey += str(digit)
    return ocKey
```

are possible (ranging from 000 to 111). The method is explained in more detail in chapter 4.

On top of that the method can also filter noise, which is a major cause of errors when finding the empty space. In the current implementation this is very basic though, there need to be at least two or more points (this can easily be adapted) in a box in order to be classified as full and used to calculate the empty leaf nodes.

Calculate the empty leaf nodes To find the empty leaf nodes the function find_empty is called (table 6). This function iterates through all levels of the octree and stores the materialised paths of the nonempty (full) leaf nodes in a set. This set is then used to calculate which children are missing. On every level all the children of full nodes are created. Then it is checked whether these children are a member of the nonempty set. If this is not the case they become member of the empty set.

To go from the materialised paths of the empty leaf nodes back to coordinates the function getCoord is called (table 7). This function iterates over all the digits of a materialised path and derives the coordinates from it. It can be seen as the reverse of the getMaterialPath function (table 5). The output of this function is stored in the emptyspace database table.

6.4.2 Method Analysis

The methods that have been used in the Project Pointless application have been conceptually described in chapter 5 and more technically in paragraph 6.4. This paragraph reflects on the methods used.

Table 6: function to find the octree of empty space

```
def find_empty(table, maximumLevels):
    # This function finds the empty nodes for every level in the tree
    print "Calculate the empty leaf nodes in the point cloud"
    tree = ['0', '1', '2', '3', '4', '5', '6', '7']
    empty = set()
    nonempty = [str()]
    # Iterates over the depth of the tree
    for level in range(maximumLevels):
        # Stores all the nonempty nodes of the current tree level in a
set, and adds the set to a list
        nonempty_cur_level = set()
        for entry in table:
            nonempty_cur_level.add(entry[0][0:level+1])
        nonempty.append(nonempty_cur_level)
        # Finds all the empty nodes in the current level of the tree,
and stores these in a set
        for node in nonempty[level]:
            for number in tree:
                child = node + number
                if child not in nonempty[level+1]:
                    empty.add(child)
    return empty
```

Table 7: Function to get the coordinate of the nodes in the octree of empty space

```
def getCoord(depth, materialPath):
    #This function finds the coordinates of the empty space
    #checks the size of the voxel according to the lenght of the path
    if len(materialPath) < depth:
        leafSize = 2 ** (depth - len(materialPath))
    else:
        leafSize = 1
    x, y, z = '','',''
    check digit per digit
    for digit in range(len(materialPath)):
         # check which 0s/1s should be added to the binary coordinate
strings
        if int(materialPath[digit]) > 3:
            if int(materialPath[digit]) > 5:
                if int(materialPath[digit]) > 6:  #7
                    x,y,z = x+'1',y+'1',z+'1'
                else:  #6
                    x,y,z = x+'0',y+'1',z+'1'
            elif int(materialPath[digit]) > 4:  #5
                x,y,z = x+'1',y+'0',z+'1'
        else:  #4
            x,y,z = x+'0',y+'0',z+'1'
        elif int(materialPath[digit]) > 1:
            if int(materialPath[digit]) > 2:  #3
                x,y,z = x+'1',y+'1',z+'0'
            else:  #2
                x,y,z = x+'0',y+'1',z+'0'
        else:
            if int(materialPath[digit]) > 0:  #1
                x,y,z = x+'1',y+'0',z+'0'
            else:  #0
                x,y,z = x+'0',y+'0',z+'0'
    remainingLevels = "0"*(depth - len(materialPath))
    x += remainingLevels
    y += remainingLevels
    z += remainingLevels
    # convert the binary string to decimal integers
    x = int(x, 2)
    y = int(y, 2)
    z = int(z, 2)
    return str(materialPath)+" "+str(x)+" "+str(y)+" "+str(z)+"
"+str(leafSize)
```

Scaling, translating and snapping Snapping is used to identify in which leaf node of the octree a point is located. In order to make this work the point cloud is scaled to always have the smallest leaf node on a 1x1x1 resolution in the local coordinate system. The point cloud is also translated to the origin of the three dimensional Cartesian space. This way the points always lie between 0 to 2^n, where n equals the amount of levels in the octree.

This method for locating points in leaf nodes is very effective. Because the point cloud is scaled to a grid with a 1x1x1 resolution the coordinates of the lower left front corner of the leaf node (the corner closest to the origin) can be found by simply cutting of the decimal digits behind the comma and truncating those digits towards zero. In Python this can be done by simply calling the int() function on the scaled coordinate of a point. The entire process of going from a raw coordinate to a leaf node contains three steps: add the translation factor to a coordinate, multiply it with the scaling factor and call the int() function on it. This is a very simple, elegant and efficient way to snap a point to a leaf node.

Finding full leaf nodes: binary masking After the points are located in leaf nodes the materialised paths of these nodes need to be calculated (their location code in the octree structure). This is done by binary masking, in which the coordinates in binary are compared to a mask for each dimension (001 for x, 010 for y and 100 for z). Every time the octree splits to a new level three digits are added to the binary materialised path (the domain is 2^n, with n as the maximum amount of levels). Therefore, checking a coordinate three binary digits at a time can reconstruct the materialised path from the root to the leaf node.

Using binary masking for calculating the materialised path is conceptually complicated, but very powerful. In only ten lines of code a location code can be calculated from the input coordinate. Since it uses only binary comparisons it is also very fast (for 2.3 million points it only takes 11 seconds, more details about performance can be found in chapter 6.6).

Finding empty leaf nodes: set operations Once the full leaf nodes have been found the empty ones can be derived from this. By creating a set that contains all potential nodes on a specific level and checking whether or not they are also member of the set with full leaf nodes the empty ones are identified. Checking for membership in sets is very fast in Python due to the use of hash tables. Testing whether a variable is a member of a set is a very common usage of sets. Therefore, this part of the script is very efficient in terms of speed for finding the empty leaf nodes in the octree structure.

Storing octree in Postgres While the octree structure is being created the script stores the data in a temporary .csv file. After the entire point cloud is processed the .csv file is copied into the Postgres database. Writing the octree to a file and then to a database is by far the most time-consuming part of the Pointless converter script. For 2.3 million points it takes about 1 minute and 17 seconds (out of 1 minute 40 seconds in total, again, more details about performance can be found in chapter 5.6). This is one of the bottlenecks of the current implementation.

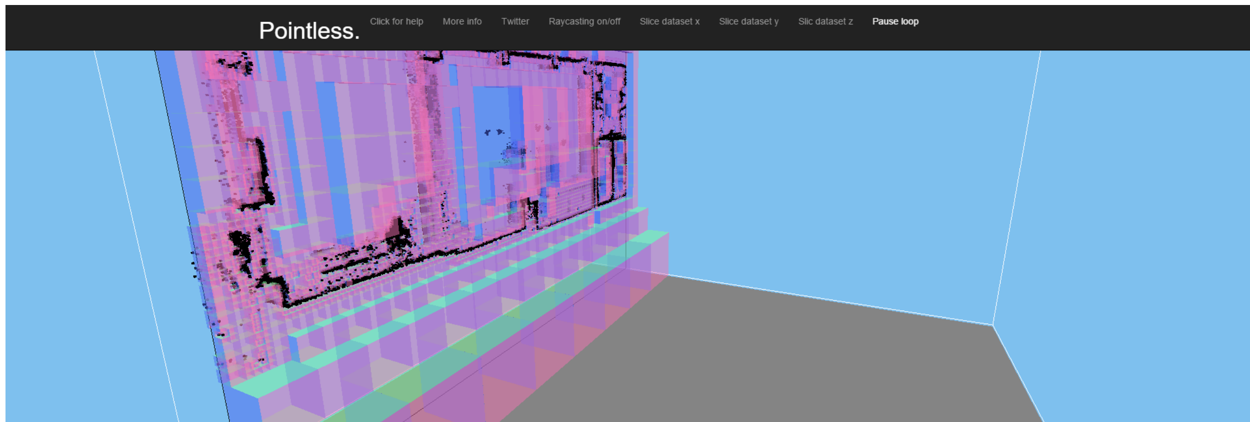

Figure 33: Visualisation of empty space in a slice of a point cloud

View octree in browser Once the octree structure is stored in the database it can be used for many applications. To visually inspect the octree a simple viewer has been created using the Three.js JavaScript library for WebGL. This viewer can show parts of the dataset and visualise a route inside the point cloud. However, creating the octree structure was the main focus of this research and the viewer just a validation tool. Therefore, also the viewer in its current state can be considered a bottleneck of the application, since it is not able to render large amounts of data at the same time.

Discussion In conclusion it is clear that the methods for creating an octree structure that stores not only points, but also empty spaces are very efficient. To put points in leaf nodes, calculate their materialised path and then deriving the empty leaf nodes and their materialised paths is very well implemented. Therefore, the application can already be used for a number of applications as described in the introduction and in the use case chapter (an actual use case can be found in chapter 6). Ways to improve the current implementation even further have been identified as well. Further elaboration on these improvements can be found in chapter 5.8 and 8.

6.5 System Characteristics

The characteristics of the system developed by Project Pointless during the Geomatics Synthesis Project of TU Delft will be explained in the following subchapters. It will discuss the dependencies, trade-offs, user conflicts and costs of the system.

6.5.1 Dependability

The dependability is composed by the measures of reliability and maintainability.

The octree converter is designed to work in a reliable manner. After the script starts it first checks all input parameters to prevent catching invalid input. It will make sure that the input point cloud is of the right file format (.las) and whether the database connection can be established with the username and password and if the database already exist (and will therefore be overwritten).

Another design decision in the application to make it more reliable is that the .las file is first copied to the localhost and then converted from there. This allows the user to continue working on the original point cloud while the conversion takes place, without affecting the script. Also when the leaf nodes are calculated they are first all stored in a .csv file before writing them to the database. This method ensures that the duration of the connection with the database is minimized and therefore it is less likely that the connection with the database is (temporarily) lost. Losing the connection with the database while writing data to it can cause the script to crash.

The viewer is less reliable than the converter script. Since the viewer works with WebGL in the browser and can only render a limited amount of points it is possible that when loading a large point cloud the viewer crashes. With the test point cloud used for the project this does not happen, but the test point cloud is by no means a very large point cloud with only 2.3 million points. Therefore in its current state it cannot be expected to work for every point cloud.

The application is safe to use even when it crashes or returns unexpected results. If the script crashes it does not affect the system as it can simply be stopped in the task manager in the worst case scenario. Also the dataset is copied to the localhost before running the script and only uses the copied .las file. Even if in some very unlikely situation the converter script would harm the input file and make it unusable, it would not affect the original point cloud but only a copy of it.

On top of that is the programming language very scalable and compared to other languages easy to understand for readers that can because of this relatively easy improve or change the implementation for their needs. The workflow is therefore easy to maintain also for people not directly involved with the project.

6.5.2 Trade-offs

A number of potential solutions have been reviewed before deciding on the methods to use for the Project Pointless application. Some of the solutions that have been discussed were of very high quality, like mainly Potree (Schütz, 2015), but were still not selected to be implemented. They have not been selected because there was either not enough time to implement them or the project team did not have the required skills or time to implement them. The results is a trade-off between the quality of the final product and the available resources. The Potree viewer and PCL are examples of software that have a very high performance, but the Project Pointless team was unable to implement them because no team member new the programming languages used for these programs. In other words, creating the octree structure of empty space had a higher priority and therefore not enough time was available for diving into the Potree viewer and PCL to connect them to our application. The major trade-offs of the current implementation are therefore performance of the visualisation

and the lack of usage of already existing applications.

6.5.3 User Conflicts

Since the application works locally the script can only run once at a time. This means that the converter script does not allow to be run by multiple users on the same machine. However, the database can be used by different users simultaneously. Viewing the octree structure or using it for an application can happen for multiple applications or multiple users of the same application at the same time on the same machine. However, it would be preferable for the future to be able to run to whole application server based to allow more scalable user access.

6.5.4 Cost Estimate for Operational System

There a number of costs involved in the process of creating a point cloud and storing it in an octree structure. First of all, the acquisition of the data set costs both money and time (see chapter 6.2). The acquisition requires hardware (a laser scanner or photogrammetric equipment) which is costly. Also having someone scanning a building can easily take up multiple days and requires knowledge and skills about the setup of the scanner and the scanning itself. The scanning equipment if not borrowed (€20.000,- for a Zeb1 scanner and up to more than €100.000,- for the Leica scanner) is an investment that has to be made once in order to create professional point clouds. The labour or processing cost for creation of the point cloud will also cost money and this is a cost that is recurring for every new point cloud that is being scanned. An estimation of the cost breakdown for the team's scanning can be found in the appendix 11.2.

All software used for the application is open source and is available for download, use and reuse free of charge. The software should run on a high performance computer or server for the best results. This hardware should be available as well, but requires a significantly smaller investment. New hardware should be purchased relatively often to keep up with the rapid technological developments. Also the open source software is being updated for free, as long as it has an active community. Developing applications that use the Project Pointless octree structure are not considered in this cost estimate, but would require a significant investment, too. At the moment though, the biggest investments for software development are of temporal nature rather than monetary.

6.6 Performance and Scalability

To test the performance of the application the different functions have been timed with different input parameters. First the performance of the script is compared for different resolutions of the octree. The performance and scalability of the standard resolution of 8 tree levels is being compared with the ones of 7 and 6 levels. Additionally, the performance is compared for different dataset sizes. The test dataset used for this project contains 2.3 million points. To test the scalability, the script

has also been timed for datasets of 4.6 and 9.2 million points.

6.6.1 Influence of Resolution on Performance

For the octree script of the Project Pointless application a standard resolution of 8 levels in the octree was used. A resolution of 8 levels means that the smallest leaf nodes are approximately 10 cm3 for the dataset used. However, to test the performance 7 and 6 levels have also been tested. A resolution of 7 levels comes down to smallest leaf nodes of 20 cm3 and for 6 levels this is 40 cm3.

Figure 34 shows the performance of the different resolutions and compares the change in speed per component of the script. It shows that doubling the smallest resolution by going one level deeper does not take twice as much time. In fact the difference in total run time between 6 levels and 7 levels is larger than the difference between 7 levels and 8 levels. This indicates that the amount of levels scale very well. Looking at the components it is visible that mainly the performance of writing the empty leaf nodes to the database is taking increasingly more time. This is logical since not only more empty leaf nodes need to be written to the database since the point cloud automatically splits to the smallest level. Also the amount of empty leaf nodes grow exponentially with every level resulting in exponentially more rows to store in the database. This is actually already a lot more efficient than writing to the database directly, as this requires to keep the database connection open for a longer time which makes the application less robust as a sudden loss of the connection could cause the script to crash. On top of that this would mean that for every point written to the database a single SQL query has to be performed. Writing to a file first and then loading the file to the database is already efficient in this respect. The performance of writing the points to a file and the points file to the database remains stable when creating different amounts of levels since the same dataset (the same amount of points) were used in all three test cases.

When the relative time distribution of the scripts with different octree resolutions are being considered (figures 35, 36 & 37) it is visible that 65% to 74% of the time is spent on writing the points to a temporary file before storing it in the database. Storing the files into the database shows a remarkable pattern which cannot be easily explained: for six levels it took up 12% of the total running time, for seven levels this increases to 14%, but for eight levels it somehow only takes up 10%. This is probably due to the exponential increase of time for storing the empty leaf nodes in the database: this goes from less than 1% for six levels, to 2% for seven levels and to 8% for eight levels. Calculating the materialised paths is very well scalable, it only increased with 1% when increasing the amount of levels from six to eight.

6.6.2 Influence of Dataset Size on Performance

The dataset that was used for creating the Project Pointless octree implementation contained 2.3 million points. This can be considered to be a very small point cloud dataset. To retrieve insights in how the performance of the converter script scales with datasets that contain more points three cases have been tested: first with the original dataset of 2.3 million points, then with a dataset containing

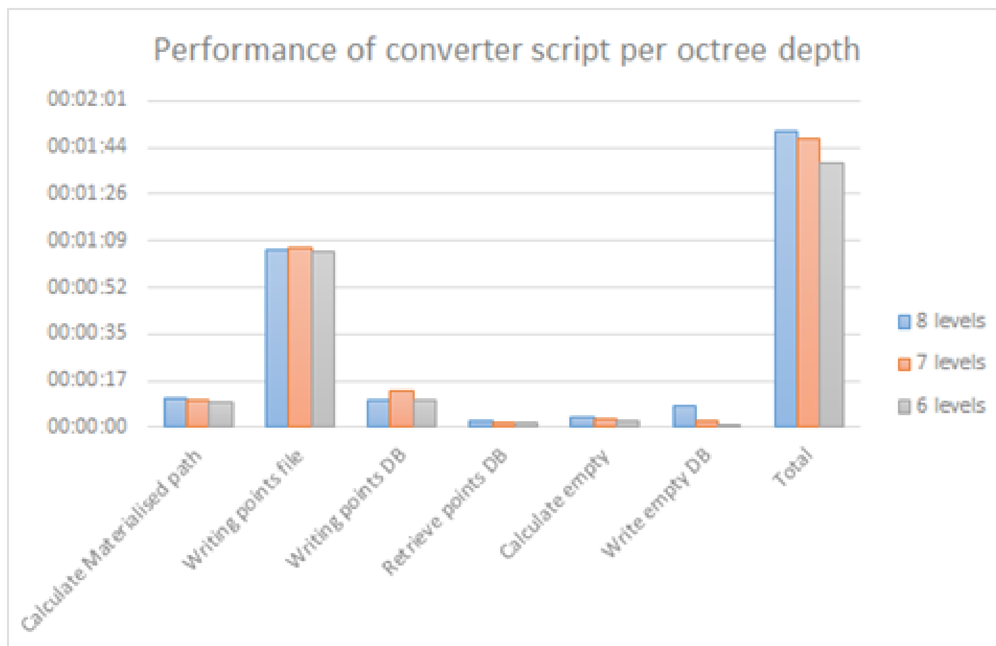

Figure 34: Performance of octree generation script per octree depth

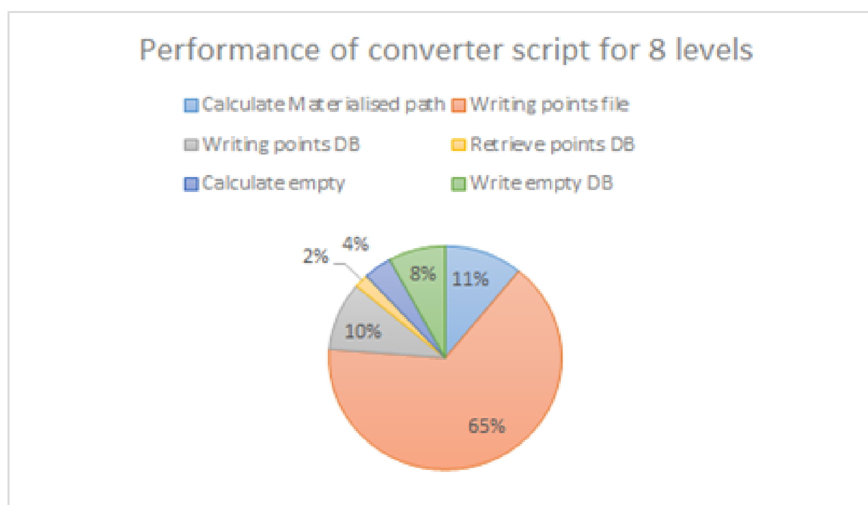

Figure 35: Performance of converter script for 8 levels

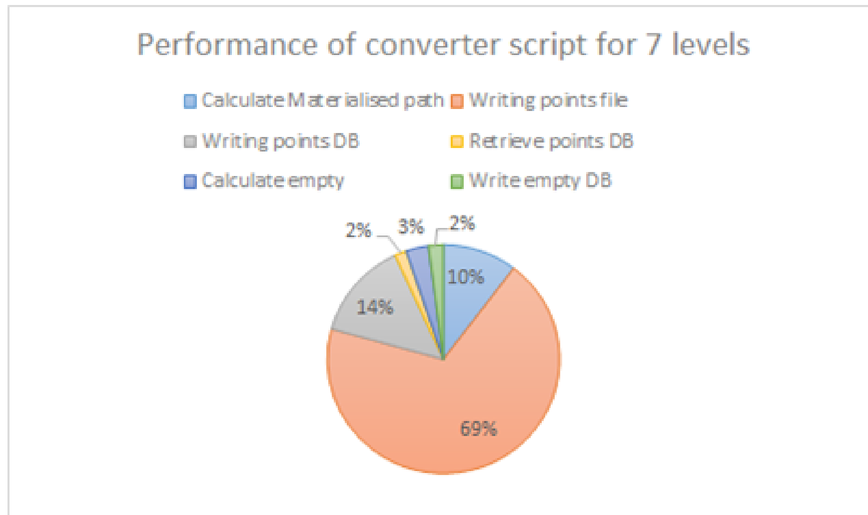

Figure 36: Performance of converter script for 7 levels

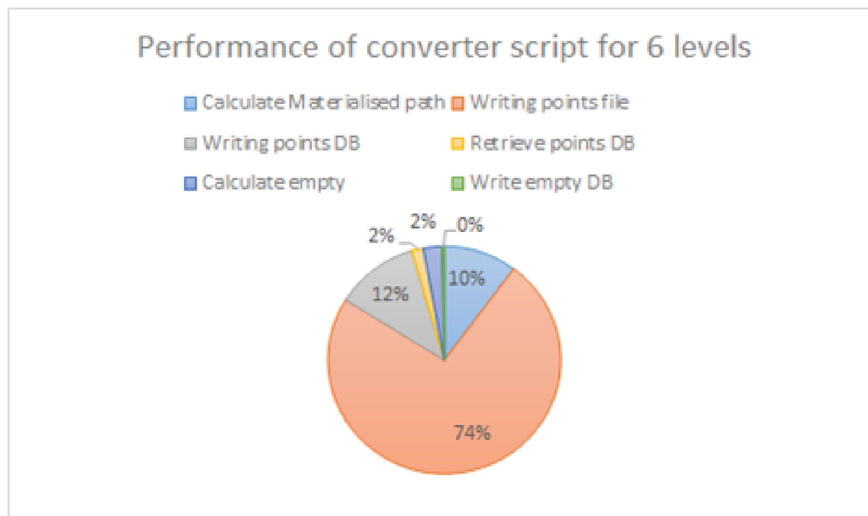

Figure 37: Performance of converter script for 6 levels

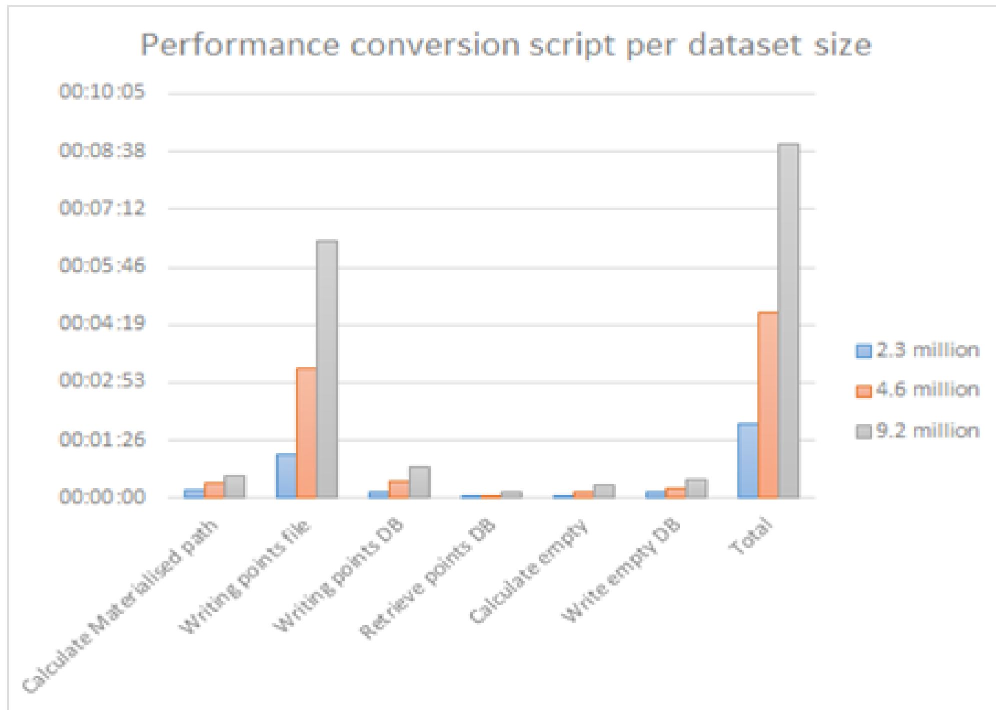

Figure 38: Performance conversion script per dataset size

4.6 million points and finally with a dataset of 9.2 million points. Comparing the performance of these three test cases should give an indication of the scalability, but other factors also play a role in scalability that are not being tested here (for example: can the hardware that is being used to run the script handle the amount storage that is required in local memory).

Figure 5.6.5 shows the differences in performance for the major components of the converter script. First of all, it is visible that writing the points to a file takes increasingly more time. This is logical since it needs to write twice as many lines to a text file when a point cloud of twice as many points is used as input. This is by far the most expensive and least scalable component of the script. Still, as mentioned in the previous paragraph this is the most efficient way of writing points to the database from what was tested by Project Pointless so far. It can process a point cloud of 2.3 million points in less than 1.5 minutes.

Looking at the relative distribution of time spent on different components of the script a pattern is again visible (figures 39, 40 & 41): for 2.3 million points 65% of the time is spent on writing points to a file, for 4.6 million points it takes up 73% of the running time and for 9.2 million points 74%. Not only does it take more time to convert more points, but it also takes up a larger share of the total time spent when more points are converted. This means as soon as future improvements are implemented that make writing points to a file and to the database more efficient incredible increases in performance can be achieved, since it is only this part of the process that is currently slowing down the already very fast script. The calculations for finding the materialised paths scales well: the more points are being converted the smaller the share of the total spent time it has.

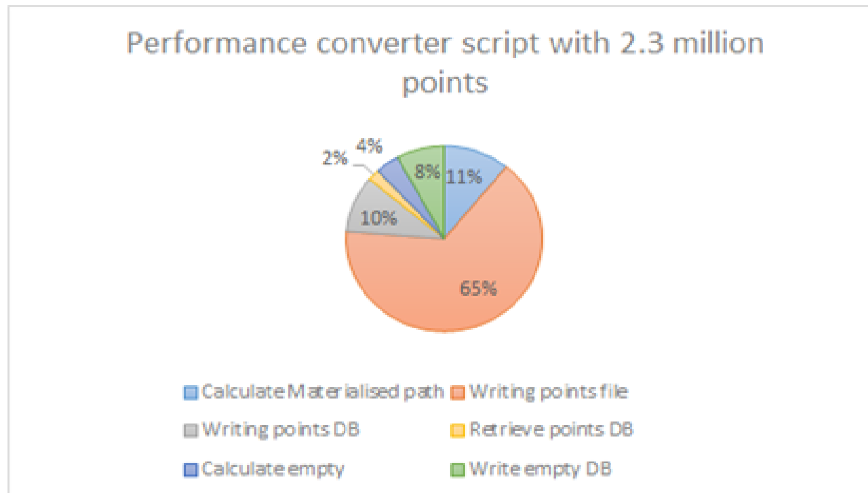

Figure 39: Performance converter script with 2.3 million points

Calculating the empty space in point cloud scales well, too. The relative amount of time it takes is a stable 5%.

6.6.3 Performance of Visualisation Results

The performance of the visualisation of both, point cloud and empty space decreased significantly with the level of detail. Rendering the whole dataset in the browser at once was impossible also with 2.3 million points only as the browser always crashed or froze before everything was loaded. Therefore the FME Inspector was used to make screenshots and validation analysis of the results, which had the advantage that point clouds and the geometry of the boxes could be rendered at the same time. However, it is still not fast and flexible enough to show any results live, even taking a screenshot only can take several minutes or up to one hour. A worksheet to create the geometries can be found in appendix 11.3. However, the Potree viewer is very efficient and is very well capable of viewing large amounts of data using Three.js in the browser. Therefore, Project Pointless could be improved by connecting the octree structure presented in this research to the Potree viewer. There was not enough time to create this connection in the limited time available. This is could be a very interesting next step to improve performance of viewing the Project Pointless implementation.

6.7 Operations and Exploitation plan

The goal of Project Pointless was to research new methods for structuring point clouds and creating new possibilities for applications based on using the empty space inside of them. There was no commercial motivation behind the project. Therefore, the team was extremely transparent about the process as well as about the obtained results. The code is online on GitHub for everyone to use, reuse and continue working on under an open source license. This does not mean that nothing

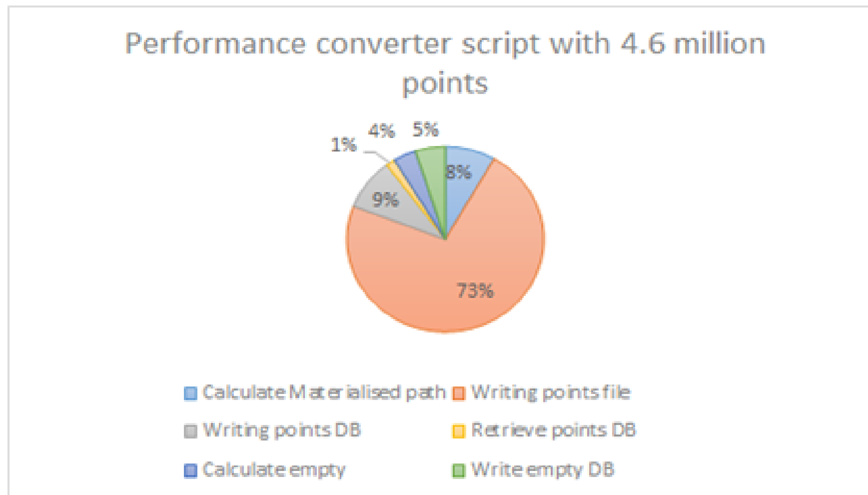

Figure 40: Performance converter script with 4.6 million points

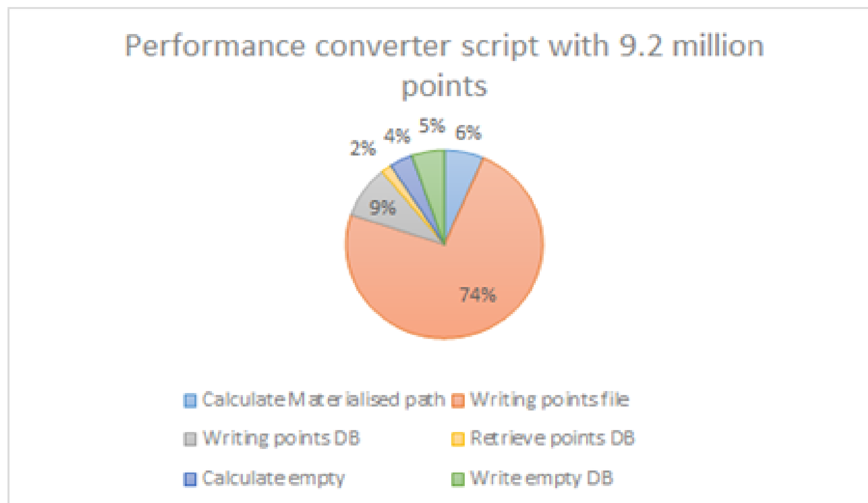

Figure 41: Performance converter script with 9.2 million points

is gained from this project. Having new knowledge on structuring and using point clouds and offering a new method that is freely available online has an added value for both science as well as for developers creating new applications with point clouds. This innovation is valuable for the community of Geomatics engineers. The members of team Pointless also gain from becoming more visible inside the field.

It is possible to exploit the idea of structuring point clouds in an octree using the open source software. If someone builds a good application on top of the octree structure that can be created using our open source software, then that is a legit way of creating revenue with this innovation.

6.8 Discussion

One of the aims of Project Pointless was to create an application as a proof of concept. This application has been built using the methods described in the previous paragraph and exists of three main parts:

- the script that converts the point cloud to a local coordinate systems, finds the empty space and calculates the octree structure,
- a database that contains the octree structure and the point cloud,
- and a basic viewer to visually inspect the dataset.

The script to generate the octree is the core product of this project and most of the effort was put into the developing this. The implementation was based on a large variety of research papers and this has resulted in a script which produces the correct output and which has a good overall performance. As the results are good and fulfill all requirements of the project, mainly the speed of the process is left to be improved and discussed. Speed is a major drawback of point clouds in general, the large amount of data makes them difficult to use. The test dataset of 2.3 million points takes about two minutes to convert the input point cloud, create the octree structure, calculate the empty space and store everything in a database. It is outstanding that over half of the run time is spent on reading and writing, the calculations take only 35% of the time to perform, which is an incredible achievement for point cloud processing which is usually really slow. Reading and writing are operations that are becoming increasingly faster with the development of flash storage as this kind of storage is not limited by the maximum speed of a spinning disk (Oracle, 2014). Therefore, there is still a large potential for improving the performance even further. The scalability has also been tested. The results indicate that the calculations in the script scales well. With four times more points it takes only 3.6 times more time to calculate the empty space.

The Postgres database that is created by the converter script exists of two tables. The first table contains the entire point cloud. This table has columns for the materialised path, (x, y, z) coordinates and (r, g, b) colour values. Using a database has a number of advantages over using files. First of all, the multiple users can use the same data without interfering with each other. This also means

that there are no problems regarding everyone in an organisation having an up-to-date version of a dataset or users creating conflicting changes. Database storage also scales well for storing and querying large datasets. The tests performed using our database showed that writing large amounts of data to the database is expensive, albeit less expensive than writing to a file. On the other hand, querying the database is efficient and scales well: two times more points take little more than twice the time to query.

The current implementation can also be improved by using the path of the scanner, when this is available. In the octree there is currently no distinction between empty space usable for routing or for example empty space found inside a wall. The scanner's path can be used to validate the empty space and filter out nodes of the octree not connected to an intersection of the path. This could take shape as a region growing algorithm that moves through the octree. The implementation of such an approach is a topic of future research (see chapter 9.8).

Another method to structure empty space of a point cloud, not using the grid system used in this project, but using sphere leaf nodes can increase the quality of the representation of the reality. Peters et al. (2015) show that also a solution using a union of spheres instead of a grid approach is possible to model the environment of a point cloud. Their research could be extended with a higher focus on empty space.

7 Use case: 3D path finding in empty space

The core of this project is the identification of empty space inside a point cloud. The previous chapters have explained this process in detail. This section will explain one of many possible use cases. The use case Project Pointless developed and focused on is 3D path finding in empty space, a smart 3D path finding method which can be used for many applications. For example finding the shortest route between location A and location B inside a building for a drone. A* is familiar to the team as it was taught and the programming of the algorithm was part of an assignment all group members had to undergo in the GEO1002 course about GIS and cartography in the first quarter of the Geomatics and the built environment MSc programme at TU Delft. In the case of Project Pointless it is not necessary that the route is bounded to the ground as a drone is airborne and the route can because of this be 3 dimensional. However, the size of the drone has to be taken into account when navigating through objects. Another example is finding the shortest path in which a wheelchair is able to move. This example is a bit more complex than the drone example as a wheelchair is bounded to the ground. For this use case we focuses merely on the drone example independent from its size. This is enough to validate the empty space and its opportunities when structured in an octree. In the next subsections it will be explained which path finding algorithm is used, how it works and how the neighbour finding of nodes in the octree is handled.

7.1 Path Finding Algorithm

A path finding algorithm is an algorithm for finding the shortest or fastest route between two nodes in a network of nodes. The first node can be defined as the starting node and the second as the target node. A path is the route between two adjacent nodes. And the cost between two neighbouring nodes is equal to the length of the path. To find the shortest route, so the route with the lowest costs, between the start and the target adjacent nodes are recursively explored (Noto and Sato, 2000). For the algorithm to know his neighbouring nodes the network should have a topology, it should be a graph, or the algorithm should be able to generate the neighbours for each node.

7.1.1 Dijkstra Algorithm

A common path finding algorithm is described by Edsger Dijkstra in 1959 (Nosrati et al., 2012). The Dijkstra algorithm calculates the optimal route between a start and the target by searching for the minimal cost to travel. The algorithm works as follows: It selects the neighbouring nodes of the starting node and calculates the cost. A list is created in which for each node the cost and the route to get there is stored. The node with the lowest accumulated cost is marked in the list. This step is repeated with the marked node. When a node already exists in the list it will only replace the existing one if the cost is lower, else it will be ignored. These steps are repeated until the target node is reached. In general this process will be concentric. Due to this the time to find the shortest path is heavily dependent on the distance between the start node and the target node. With a large distance the search time becomes very long. This makes the Dijkstra algorithm unsuitable for

real-time applications (Noto and Sato, 2000) and thus not usable for the octree structure as either.

7.1.2 A* Algorithm

A solution for the time problem of the Dijkstra algorithm was found by Hart et al. (1968). They extended Dijkstra's solution by implementing a heuristic cost to the algorithm. The heuristic cost estimates the cost from the current node to the goal node. The new cost to mark the node with the lowest cost is now: The cost from the start node to the current node (path cost) + the heuristic cost (Nosrati et al., 2012). Because the heuristic is now added to the cost of the route already travelled all nodes which are not between the start node and the goal node get a higher heuristic cost compared to the nodes which are. Due to this, the speed of the algorithm is not related to the network size, but only to the length of the route.

In Xu et al. (2015) a method is described for using the A* algorithm in combination with an octree structure. In an octree structure the centre points of the octants can be seen as nodes. And the length between the centre points of two adjacent octants can be seen as the cost. Xu et al. (2015) find the neighbours by checking if the length between two centre points does not exceed half of the length of the sides of the two nodes. So to find the neighbours of each node they have to make this comparison between the current node and every other possible node. As they state themselves this is kind of a 'naive method' for finding the neighbours (Xu et al., 2015). In the following chapter smarter alternative methods for finding neighbours will be explained. The method used by Project Pointless is based on a paper by Vörös (2000) and explained here in detail.

7.2 Neighbour Finding

To find a route in an octree the connectivity of the nodes has to be known, so whether the nodes are neighbours. A node is another node's neighbour if they share a common face.

There are many different solutions to find adjacent nodes in an octree. Samet (1989) suggested the classical method that finds neighbours based on common ancestors of the nodes. However, this method does not take into consideration, that in certain situations no adjacent node exists. Gargantini (1982) uses locational codes which refer to the path of the node, similar to the material path approach used in this project. So it uses two different methods depending on whether the neighbour belongs to the same octant or not. The drawback of this solution is that it is unknown if the found adjacent node is part of the object or not which makes another search procedure necessary. Vörös (2000) benefits from both method and avoids their drawbacks. Furthermore the approach holds implementation advantages over other methods like the ones presented by Kim and Lee (2009), Namdari et al. (2015) and Payeur (2006). Vörös (2000) suits Project Pointless very well as its setup uses a similar octree and binary matrix structure.

The method of Vörös (2000) and its implementation will be further described here. In the route finding application of Project Pointless the adjacent nodes of the current position are constantly

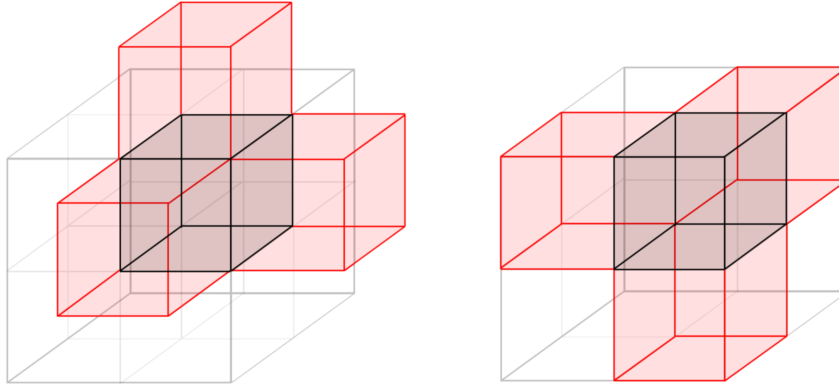

Figure 42: Equal sized inner neighbours (left), equal sized outer neighbours (right)

being calculated on the fly. With this method any previous calculations using the whole dataset and a constant storage of neighbours is avoided. The algorithm finds all possible neighbours of the current node and then checks whether they exist in the octree of empty space (so is neither FULL nor GRAY, but WHITE) using a fast hash table lookup. The method only finds face neighbours, which is sufficient for this application because line or point neighbours do not guarantee enough empty space for an object to move through.

Vörös (2000) differentiates between inner and outer neighbours, where an inner neighbour B is located in the same octant as node A. Moreover a neighbour B can be of equal, smaller or larger size than node A. The location codes of the nodes in a linear octree can be derived using binary codes (see 6).

7.2.1 Equal Sized Neighbours

There are two possibilities for the equal sized neighbours of a node (figure 42). Node A can be in the same octant as neighbour B in direction D (so direction Dx, Dy or Dz depending on which possible neighbour is getting checked), then B is a inner neighbour of A, or it is not and B is an outer neighbour of A. For the inner neighbours only the last digit of the materialised path has to be changed to the complement in direction D. If it is an outer neighbour the last (k + 1) digits of the materialised path have to be changed to the complement in D where k is the number of steps until A and B share a common ancestor.

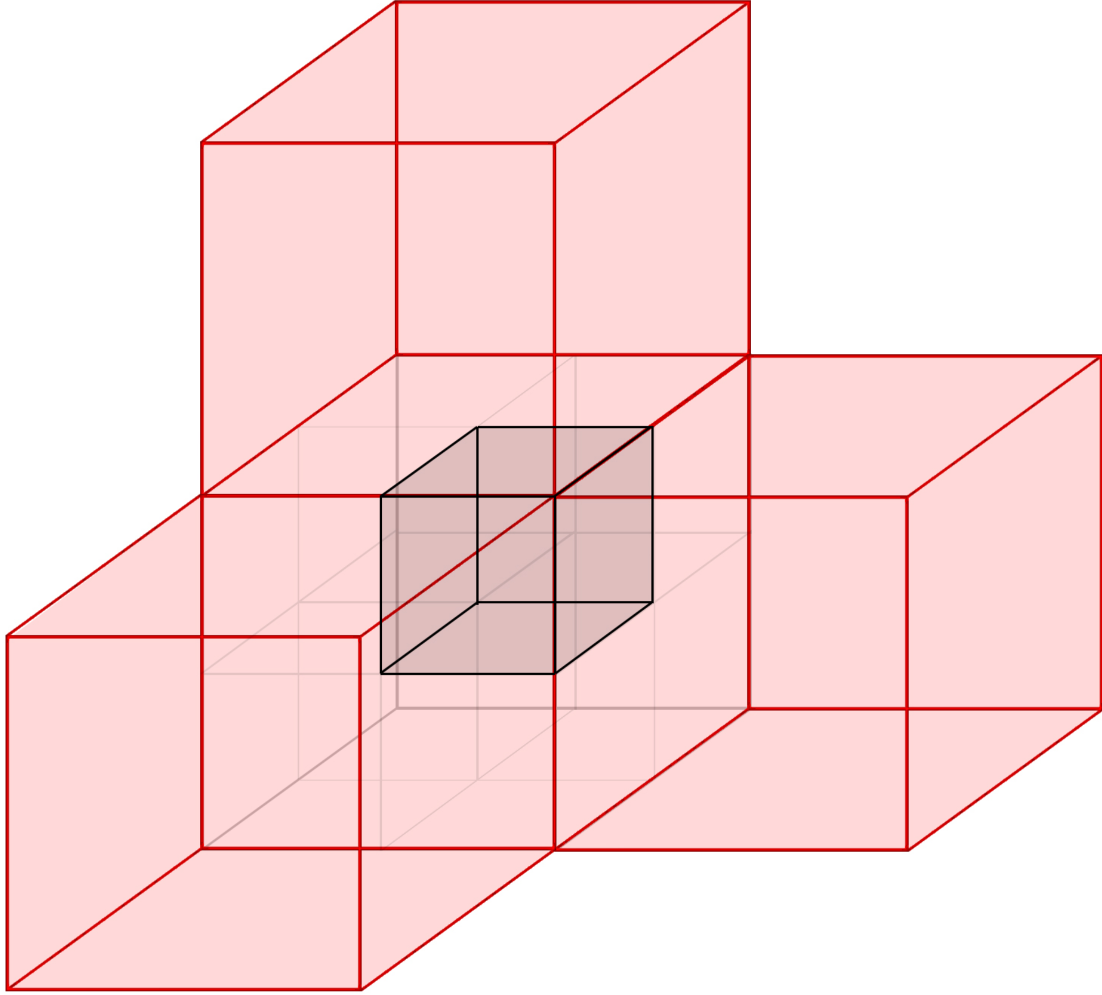

Figure 43: Larger sized outer neighbours

If k is bigger than the node level in the tree, there is no outer neighbour (the shared ancestor is on a level higher than the root of the tree).

7.2.2 Larger Sized Neighbours

Larger sized inner neighbours cannot exist since they share the two neighbours share the same parent node. The larger sized outer neighbour B (figure 43) is found by successively deleting the last digit of the materialised path of A. There are is a number of k larger sized neighbours. Hence when there is an equal sized outer neighbour, there is k larger ones also, as the equal side's parents will also be neighbours of A.

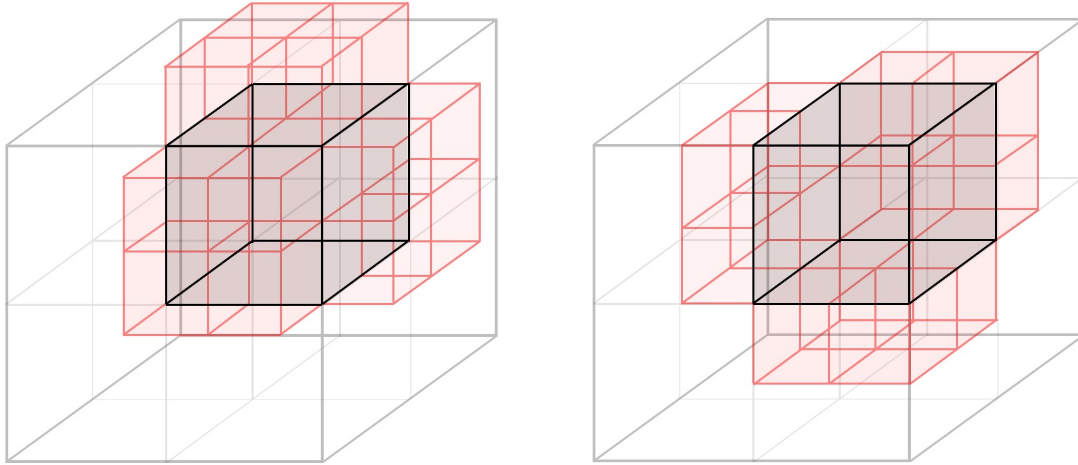

Figure 44: Smaller sized inner neighbours (left), smaller sized outer neighbours (right)

7.2.3 Smaller Sized Neighbours

All smaller sized neighbours of A (figure 44) are the “quadruples of adjacent octants, suboctants, ... of its equal sized neighbours” (Vörös, 2000). So again, they are derived from the materialised path of the equal sized neighbours, but added with another level, so another digit for the materialised path. For the inner neighbours the binary materialised path stays the same for the direction D which is getting checked and the others are getting flipped to all 0 and 1 combinations to reflect all 4 possibilities. For the outer neighbours the same holds, but the composite is taken for direction D. For deeper levels the same processes can be repeated.

7.3 Results of the Use Case

The case study demonstrates the concept of storing empty space developed by Project Pointless during the Geomatics Synthesis Project for developing useful applications in a short period of time. The algorithm automatically finds a route through the empty space of the point cloud. An example of the three dimensional route through the empty space of point cloud of the Bouwpub can be seen in the figure 45. This chapter reflects on the results of the implemented use case and the potential for its practical application.

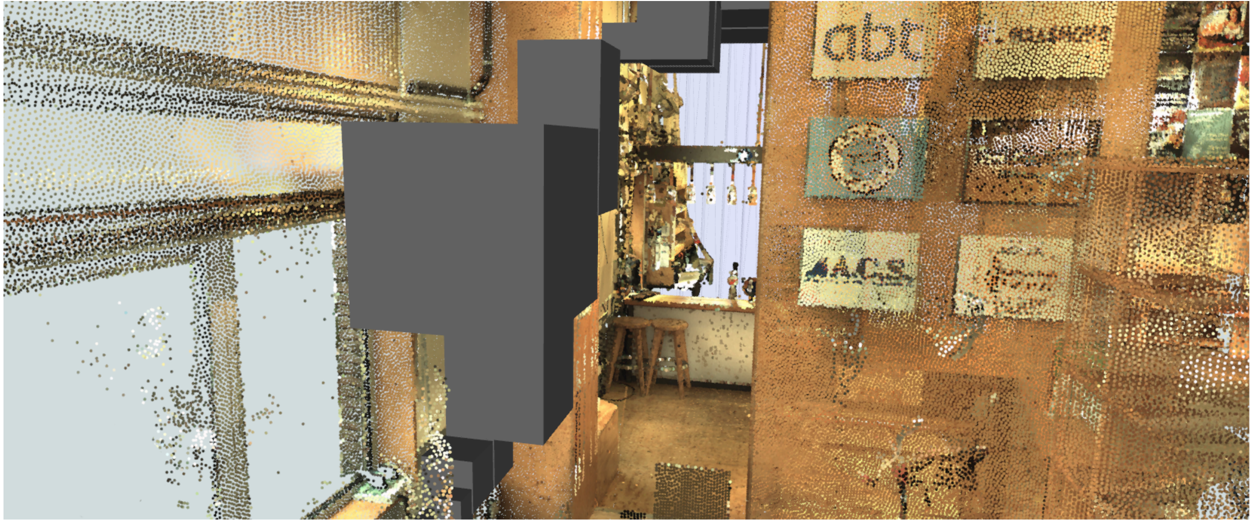

Figure 45: Visualisation of a route through the empty space in a point cloud

7.3.1 Performance Estimate

The routing application results in a set of nodes that represent the shortest route between two points in the empty space of a point cloud. The algorithm takes any start or target node as input and finds the shortest route between them, avoiding nonempty space. Visual inspection shows that it does not visit any full nodes; nodes which contain parts of the point clouds. It can therefore be seen as a proof of concept for the research results of Project Pointless.

There are some good results for the routing algorithm that moves through empty space (figure 45). It can also be seen as an intuitive approach to pathfinding, as it is not very relevant to know all the obstacles on the way, but it's more important to know all the spaces that connect two places together. The current implementation of the A* is still basic and calculates the route in a brute force way. It returns very accurate results and does not require any pre-processing. However, in its current state it takes a lot of computation, so the performance is not yet optimized. The algorithm has been timed while computing routes in octrees with different amount of levels. These numbers show that there is still room for improvement of the performance: for six levels the algorithm takes less than 20 seconds to find a path. For seven levels it takes two minutes. However for eight levels it suddenly takes almost half an hour to compute a route between the same nodes. This is because the amount of neighbours that should be checked in the A* algorithm increases exponentially with the addition of an extra level. A graph, so pre-processed neighbours, could circumvent this, but would need a lot of preparation and requires additional storage space. Another possibility, especially if on the fly calculations are preferred, is a solution which finds a route that initially uses gray nodes. It only increases the level of detail later on once a route with grey nodes is found. This way it is less likely that a complete route is being calculated into detail to only run into a dead end right before the route reaches the destination.

There exist very good possibilities to visualise a point cloud, for example the browser based appli-

cation Potree (Schütz, 2015) or the desktop application CloudCompare (Girardeau-Montaut, 2015). Currently they are not yet connected to the routing application and cannot render a route inside the point cloud yet. However, the integration of the route into one of those applications is possible and would increase the rendering performance significantly. This was outside of the time scope of this project, but has a tremendous potential. The own implementation for visualisation, which was easier to implement within the limited available time, showing a route together with the point cloud could not render all the millions of point and box geometries in the browser at once. This viewer is more useful for visually inspecting slices (subsets) of the entire dataset. Therefore the FME Inspector was used, which allowed to get a visually appealing impression of the results (Figure 45), but also did not offer the performance of CloudCompare or the Potree viewer. However, visualising point clouds and empty space is only done to showcase the implementation, as most real world applications (like the navigation of robots or drones through buildings) would be able to use the calculated route without having a visualisation of it. Just the order of boxes through which the object is allowed to move - which is already provided by the current implementation - is sufficient in these cases.

7.3.2 Discussion

The use case shows, that applications of the octree structure of empty space can be created relatively easy. The routing application was built in less than a week's time and already calculates high quality routes. Also the neighbour finding algorithm on its own is really fast and can find any possible neighbour. For example, finding all neighbours of an empty leaf node at a given location takes less than 0.1 seconds. This is already very fast without having spent any time on improving the performance just yet. This kind of algorithms can be used for other potential applications as well and shows that the usage of the octree structure made in this project has a lot of potential. It shows that a lot can be achieved with it in a short amount of time. In fact, there was even a substantial delay of one and a half (out of five) days that were spent on building the application which was due to minor typos in formulas in the paper that was used for calculating the neighbours. Otherwise the application could have been developed even faster.

In conclusion, for the time spent on building this application the performance is good and the results of the route are flawless. The A* star algorithm can still be tweaked for improved performance (see chapter 9.4), but it already is fully functional. This means there is a lot of potential for rapid development of new applications based on Project Pointless' database structure.

The size of the object that is meant to use the route does not play a role for the integration at this moment. However, this could be a really useful next step to implement in the routing algorithm: only allowing routes that go through a connected open space that is large enough to fit an object of specific dimensions. Because the octree structure by Project Pointless is completely three dimensional a solution can be implemented that only allows a route in the point cloud through which objects of specific shapes and sizes fit. This also suits the application of Rijkswaterstaat for measuring whether trucks fit under bridges and tunnels or whether boats fitting through a river. A laser scan and the empty space calculation of a bridge only has to be done once and then, if the shape of the truck is known, it can always check whether it fits through or should go via another way.

There is no longer a need for a scanner that constantly scans and calculates whether an approaching vehicle fits through a construction.

Additionally the route calculated in the current state of the implementation can move through the space fully independent from rules of physics like gravity, so is applicable to flying object like drones only. As drones for indoor applications are not the only possible use case research should take place focusing more on objects that are not able to fly (e.g. by applying physics rules to restrict movement to just the floor area). Objects that can make use of this improvement range from wheelchairs to trucks to robotics.

Generally, the use case is only an example application showing the potential of Project Pointless' octree structure. The application developed here is meant to inspire other people to extend and improve the results for other potential use cases.

8 Conclusions

Project Pointless aimed to create a smart and universally reusable workflow which efficiently identifies the empty space in a point cloud using an octree structure, stores it in an efficient way and visualises it. This report provides a documentation of all the aspects necessary to achieve these goals: the project design and development logic was described, which has been the backbone of the project management. Furthermore, it includes the methods and concepts of the implementation as well all the technical details. The performance and scalability have been tested and finally a use case was developed to demonstrate the technological concept suggested by Project Pointless in a real world application.
The method of Project Pointless does not use intermediate geometric models and therefore is explorative. The raw point cloud data is merely indexed and stored in a database management system. The procedure is fast and automated, enriches point clouds with a structure that can be understood by computers, while keeping all advantages of the raw data set.

8.1 Octree Generation and Space Finding

The project successfully created a Python script, which takes a point cloud from any .las file, and efficiently generates a linear octree using binary masking. All points in the original point cloud, together with the materialised path of the nonempty leaf nodes they are located in, were stored in a PostgreSQL database. From these materialised paths of the nonempty leaf nodes, the empty nodes and their geometry can be quickly extracted. The current octree implementation was written such that an octree of 8 depth levels is generated, which equates to a +/- 10 cm resolution for the dataset used here. The speed of the octree generation script was measured by the comparison of the generation of octrees of 6, 7, and 8 levels deep. This comparison showed that doubling the octree resolution from 6 to 7 depth levels, causes only a small increase in total processing time +/- 10%), and the increase from 7 to 8 levels causes an increase of only 2%. Moreover, almost all of this increase in processing time can be accounted to the growing time needed to write the results to the database. The generation of the octree itself, and derivation of the empty space from this octree, only takes 1% longer when increasing the number of levels in the octree from 6 to 8. This indicates that the octree generation script scales very well when the resolution is increased. Moreover, 65% to 74% (depending on the depth of the octree) of the total processing time is spent on writing points to a file, thus indicating that the script can become even faster if a method is found to avoid this step. The influence of a change in point cloud size was also tested. The script was tested with point cloud sizes of 2.3 million, 4.6 million, and 9.2 million points. This comparison shows that the increase in the amount of points in the point cloud has an impact on performance, the time spent on writing points to a file more than doubles when doubling the amount of points. Also, the relative amount of time spent on this procedure increasing from 65% for 2.3 million points, to 74% for 9.2 million points. Writing points to a text file is by far the least scalable component of the script. However, here again the generation of the octree and derivation of the empty space shows to scale well, with the amount of processing time increasing 3.6 times when including 4 times more points. It is presumably the combination of fast snapping of points to the leaf nodes, binary masking, and membership testing using sets in Python, which makes the octree generation and empty space

finding so efficient.

From a performance viewpoint, the octree generation and empty space finding can thus be considered successful, and possible further optimisations have already been identified. Furthermore, the script is flexible since it is able to use any .las file containing an interior point cloud as input, due to the translation and scaling operations carried out on the points before further processing. The script is also free for anyone to use and completely open source. The implementation is also reliable since the script checks all input parameters on start-up to prevent getting invalid input, it makes sure that the input point cloud is in a valid file format, and whether a database connection can be established. Furthermore, the .las file containing the point cloud is first copied to ensure the original stays intact and can still be worked with during processing. Finally, before writing results to a database, they are first written to a .csv, which minimizes the connection to the DBMS and thus failure when this connection is lost.

The current implementation still knows some limitations. The performance and scalability of the implementation is limited by the writing of points to a file, and subsequently copying this file into the DBMS. Also the octree generation script is not yet scalable to multiple users since it can locally run only once at a time. Finally, visualisation of the points and empty space using the currently implemented combination of WebGL, FLASK, and three.js, is not sufficient for larger point clouds, which can cause crashes. This could be solved in the future by using Potree, or an adaptation of Potree for visualisation purposes.

The project successfully fulfilled its goals set out at the start. The media campaign utilised by Project Pointless attracted significant attention, which shows that the topic and idea of Project Pointless meets the state of the art and that people are interested in the result as well as its process.

8.2 Use case: 3D path finding in empty space

To do the 3D path finding through the identified empty space, the A* algorithm was used in combination with a neighbour finding method by Vörös (2000). This neighbour finding method is able to find all smaller and larger, outer and inner neighbours of a certain node in an octree. The implementation was found to work well and does not require any intermediate geometric models or graph network to find a route in 3D. All calculations are performed directly using the original point cloud source and generated octree structure. All neighbours for a specific node are found in less than 0.1 seconds.

The current implementation of the 3D path finding algorithm is already able to successfully find a route through the empty space, but knows some limitations in its performance. Neighbours are currently checked on the fly, which means that they have to be recomputed every time the script is executed. Also the script does not scale well. However, these are limitations that can be overcome, and the use case effectively showed that the octree structure and empty space generated here has great potential for real-world use cases, and allows to quickly develop useful applications on top of it. The idea of finding and structuring the empty space and making direct use of an explorative

point cloud was successfully implemented and also proved to be fruitful for future applications.

9 Future Work and Recommendations

The research for storing empty space in an octree structure has delivered many interesting insights on how to use a point cloud in alternative and direct ways. However, it has not only answered questions, but also raised new ones. This chapter will briefly discuss the areas for future research and recommendations based on the findings of Project Pointless.

9.1 Rotate Point Cloud

One of the first steps of the suggested workflow is to translate the point cloud to the origin, scale it and fit a bounding box around it. Because the point cloud itself may not be aligned with the x,y,z-axes, the resulting bounding box can result in a lot of empty space surrounding the point cloud. To minimise this empty space, the point cloud should be automatically rotated in advance as the first step of the Pointless converter. Ideally, the point cloud is rotated in such a way that the bounding box is as small as possible. Aligning the point cloud to be perpendicular to the axis could make the implementation more efficient. When the walls of the building are diagonal to the axis the same wall will take up more leaf nodes compared to when it is parallel to the axis. Future research could find ways to optimise and automise the rotation of the point cloud to use the least leaf nodes to store the structure in the octree.

9.2 Improve Octree Visualisation

In the current implementation of the octree generation and empty space finding script, the full nodes are always split until the maximum specified level in the octree. This ensures that the maximum resolution is always reached for the empty space, but does not have the benefits of storing points in an octree based on their density. An example of this is the Potree Viewer (Schütz, 2015) that can render points of a point cloud based on the level in the tree the points are stored in. The result is a great and fluent visualisation of even huge point clouds where points closer to the root are rendered first (Koo and Shin, 2005). In the approach of Project Pointless this is not used and rendering the results was therefore slow (see chapter 6.6). A future extension of this research could include a combination of both approaches which would enable a better visualisation of the results.

9.3 Merging Coordinates with Materialised Path

Another area of future research is about the possibilities of storing the coordinates of points as (part of) the materialised path. This can be done in two ways. First of all, the resolution can be increased by such a degree that only one point can be inside a single leaf node. The materialised path of the leaf node is then equal to the coordinate of the point. This approach can be very efficient as the point cloud is not generalized and remains all of its detail, but it is stored in a

smart way. Using the materialised path as coordinates of points the coordinates contain much more information than just the location; it also contains the tree structure. A second approach for merging the materialised path and the coordinate of a point is by concatenating the coordinates to the end of the materialised path. When points are always stored in the maximum resolution it is known how many digits represent the materialised path: for 8 levels there are 8 digits, for 7 levels 7 digits and so on. Therefore, it can be an option to attach the coordinates as additional digits of the code. These additional coordinates can be stored as the distance from the centre point in every dimension. This way multiple points can be clustered in leaf nodes, while the point cloud keeps all of its detail.

The advantages of combining the materialised path with the coordinates is that the storage of the point cloud in the database can be made more efficient. Also, the amount of required storage can be reduced, since less individual columns are being stored (the tables become smaller). Another advantage will also be that less data needs to be queried to retrieve the same amount of information from the database: a query returns a single value that contains the coordinates of a point and its position in the octree structure.

9.4 Path Finding

The 3D path finding algorithm described in chapter 6 has shown how easy and fast it can be to build applications on top of the octree structure. Two ideas to improve the path finding algorithm are defined in the following.

First of all, it could be interesting to experiment with the use of different levels of the octree to calculate a path in a more efficient way. A solution which finds a route that also uses grey and high level nodes (thus multiple LOD's) for the route could greatly improve the performance of the pathfinding algorithm. First the route will be calculated on a lower resolution and the level of detail is increased to a lower resolution only after a route has been found. This way it is less likely that a complete route is being calculated with a high detail to only run into a dead end right before it reaches the destination and then causes the algorithm to start again from a much earlier position. What could also be taken into consideration with this approach is whether it could calculate the routes based on different shades of gray: how full is a node exactly? In other words, can a more fuzzy classification of gray be implemented in this kind of route finding algorithm. If there is no measure of the shade of gray a gray node which is only 1% full would be considered to be the same as a gray node which is 99% full. With this knowledge routes could be calculated in a more efficient manner. The approach can be inspired by the idea of vario-scale, where important objects are loaded first (Meijers et al., 2015).

The second idea is to investigate whether the algorithm can be used for a real time navigation with object avoidance. This can be interesting for devices that know exactly where their position is, but do not have laser scanning on board and can therefore not make use of trilateration. Speed is the key in this and certainly needs improvements over the current solution, alternatively the path can also be calculated beforehand. On top of that this could include restrictions to routes adapted to

the shape, size and capabilities of the object it is getting calculated for. If the object that needs to be moved through the space is a driving vehicle with wheels the route should always stay connected to the floor and should avoid too big steps like stairs, but most importantly also be restricted to space where it can fit through. The octree structure offers great opportunity especially for the last part as the size of the empty space can be accumulated for the route. That means that a small leaf node might not be enough for an object to move through if only full nodes are round this one.

9.5 Laz File Compression

The large size and amount of data is a major drawback of the use of point clouds. A smart octree structure could possibly help improving the compression rate of las to laz-files. There are a number of techniques for compressing .las files (Isenburg, 2011). Two techniques could potentially benefit from the Pointless octree structure. First of all, the progressive compression scheme allows for "immediately display a lower resolution version of the points while detail is added as the decompression progresses" (Isenburg, 2011). It would be interesting to see if the octree could be used to improve the storage of points based on different levels of resolution. Secondly, random access schemes based on the octree structure allow to search for blocks of points and load them directly instead of decompressing the points in the exact same order in which they were compressed. Research could be done in whether the octree leaf nodes could be queried as blocks. This could potentially allow the octree structure to be transferred with the use of a .las file.

9.6 Sphere Leaf Nodes

The solution to find and structure the empty space researched in this project is based on a grid structure of the octree. When breaking to the last level for the black nodes you could almost compare the result with a voxel approach as all boxes have the same size. Depending on the empty space to be modeled a more realistic representation of the empty space can be reach through a sphere instead of a grid solution. Peters et al. (2015) show that also a solution using a union of balls instead of a grid approach is possible to model the environment of a point cloud. His research could be extended with a higher focus on empty space.

Moreover, it is possible to reconstruct the geometry of the building from the neighbouring black nodes. This was contradictory to the research topic of this GSP and its focus on explorative point clouds and its direct use without doing just that, but it is generally possible when full nodes are being merged. Also the volume of not only the empty space, but also the model of walls can be calculated through this method.

9.7 Connecting Pointless to Geographic Information Systems

In order to combine the Pointless octree structure with other processes in an organisation and therefore improving the usability it would be interesting to look into the integration of Pointless with GIS. At the moment there is not yet any functionality in ArcGIS and QGIS for processing point clouds from .las file like presented in this report. A functionality like presented in this report would add more value for using point clouds in GIS. A database connection for using the structure could already been made with a GIS, but some pre-processing needs to be performed to create geometries out of empty the leaf nodes (see appendix 11.3 for an example in FME). Future research could look into how to seamlessly integrate the methods presented in this report in software packages such as ArcGIS and QGIS. Plugins could be created based on the scripts presented here, which enable GIS users to add the Pointless octree structure to their toolbox.

9.8 Combine Path of Scanner with octree Structure

The octree structure is an efficient way for further analysis and applications of a point cloud and the empty space. However, the path of scanner method that has been discussed in (Verbree and van Oosterom, 2003) and chapter 5.1 might be of added value to the current implementation. In the octree there is currently no distinction between space that is interior and space that is exterior. Therefore the space inside walls or outside of the building is also being labelled as empty space. This means that there is more empty space stored than actually exists in the real word. The path of the scanner could be used to validate which connected empty space should be kept in the database (really usable empty space) and which empty spaces should be filtered out. This could take shape as a region growing algorithm that moves through the octree or can be achieved by simply intersecting the path of the scanner with the empty space. As soon as there is a connection with the merged empty space it cannot be the inside of a wall anymore. However, how to implement this exactly is still a topic of future research.

9.9 Noise removal

Erroneous measurements can lead to noise being present in the point cloud, e.g. points in areas where there is empty space in reality. This can cause false classification results when the empty space classification method proposed in this report is applied to the particular point cloud. Areas which are empty in reality, may then be wrongfully classified as occupied space.

A noise removal procedure applied to the input point cloud could prevent these classification errors from occurring. In the current implementation the noise is already removed to some extent; when leafnodes in the octree contain only 1 point, then they are reclassified as empty cells. Additionally, disconnected nonempty cells could be classified as empty by checking its neighbors. If all surrounding neighbors of the box are classified as empty, then the box should most likely also be labeled as empty. This would probably mean that the cell represents noise, or that an object moved through the space

while the measurements took place.

9.10 Use octree for volume calculation

Having the empty space of the point cloud stored in a database automatically raised the question whether this can be used for calculating the volume of the interior of a point cloud. A quick implementation has been created to distinguish the interior of the point cloud from its exterior and can be found at https://github.com/ivodeliefde/ProjectPointless/blob/master/volumeCalc.py. It currently is able to find the interior empty space of a rectangular building, but there is still lots of research to be done in order to perfect it. First of all, a method should be found which can define the interior space of a building with complex shapes (instead of a rectangular shape). Also the empty space inside walls are currently classified as interior empty space. This is not preferred as it should not be taken into consideration for the volume calculations.

9.11 Dynamic octree

A temporal aspect can be added to extend the three dimensions of a point cloud and its empty space to four dimensions. As soon as a point cloud changes the empty space changes as well. This means that new routes have to be calculated. The current implementation is based on a static model. The octree should be updatable and dynamic to changes without having to recalculate the whole structure. This allows for a lot more application like for example for robots in changing environments or for point clouds made by drones in silos with changing filling levels. The remaining empty space in the silo could easily calculated using the octree. The structure, also when calculated from the scratch would not necessarily have to be stored in the database to safe time.

9.12 Improving performance

Chapter 6.6 describes the performance of the current implementation. Although it is already fast and scalable there are still ways to improve it. Some experiments are currently being implemented on writing the points directly to the database without writing to a temporary file, but by writing to a file-like object inside the script. The first tests with this have shown promising results. Still there is a lot of work to be done regarding the optimisation of the scripts. For example rewriting the code to a lower level language like C++ could also significantly speed up the process.

9.13 Additional Areas for Future Research

More research should be carried out to find more advantages of the octree approach. The possibility to use it for routing was the only one which was deeper researched throughout the GSP. Originally

however, smarter rendering of only the visible points was one of the use cases for the empty space and the octree structure. A future task could be to find and develop more applications.

Currently the octree approach takes only .las files as input. To improve the functionality and interoperability of the implementation this should be extended with other point cloud files such as .xyz files or .ptx files. This should be relatively easy to implement, as the implementation as it is now uses the bounding box of the point cloud and all individual coordinates, values that are also included in the other file formats.

Finally, connect to the outdoors generally, but specifically to the projects from other groups of the GSP which all acquired a point cloud from the same building. One group explored the possibilities to better relate in an intuitive way the aerial imagery and their resulting point cloud. The other group used a street level point cloud acquired from a scanner mounted on a car for land use recognition. An interesting use case would be a combination of all point clouds and applications into one environment.

10 Acknowledgments

We would like to thank our supervisor, Edward Verbree, for all the invaluable conversations, discussions and feedback. A special thanks also goes to our project owners Martijn Meijers for his essential help and Dick ten Napel for his feedback. Also, we would like to thank Peter van Oosterom for his feedback and Leap3D for giving us to possibility to scan with the Zeb1. Finally, we would like to show our appreciation for the Bouwpub for being so kind to let us scan their building and to provide us with lovely beers.

References

Baert, J. (2013). Morton encoding/decoding through bit interleaving: Implementations. http://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/. Accessed: 2015-10-28.

Bienert, A., Queck, R., Schmidt, A., Bernhofer, C., and Maas, H. G. (2010). Voxel space analysis of terrestrial laser scans in forests for wind field modeling. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, XXXVIII.

Biljecki, F. (2015). Level of detail in 3D city modelling PhD research. http://filipbiljecki.com/research/phd.html. Accessed: 2015-10-30.

Burrough, P. A. (1986). Principles of geographical information systems for land resources assessment. *Geocarto Int.*, 1(3):54–54.

Campbell, P. C., Devine, K. D., Flaherty, J. E., Gervasio, L. G., and Teresco, J. D. (2003). Dynamic octree load balancing using space-filling curves. Technical report, Williams College Department of Computer Science.

Eriksson, U. (2012). Functional vs non functional requirements. http://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/. Accessed: 2015-10-12.

Flask (2014). Flask. http://flask.pocoo.org/. Accessed: 2015-11-19.

Gargantini, I. (1982). Linear octrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20(4):365–374.

Girardeau-Montaut, D. (2015). CloudCompare. http://www.danielgm.net/cc/. Accessed: 2015-10-15.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107.

Isenburg, M. (2011). LASzip: lossless compression of LiDAR data. http://laszip.org.

Kim, J. and Lee, S. (2009). Fast neighbor cells finding method for multiple octree representation. In *2009 IEEE International Symposium on Computational Intelligence in Robotics and Automation - (CIRA)*.

Koo, Y. M. and Shin, B. S. (2005). An efficient point rendering using octree and texture lookup. *Proceedings International Conference on Computational Science and Its Applications*.

Lawder, J. K. (2000). Calculation of mappings between one and n-dimensional values using the hilbert space-filling curve. Technical report, School of Computer Science and Information Systems, Birkbeck College, University of London.

Meijers, M., Šuba, R., and van Oosterom, P. (2015). PARALLEL CREATION OF VARIO-SCALE DATA STRUCTURES FOR LARGE DATASETS. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL-4/W7:1–9.

Namdari, M. H., Hejazi, S. R., and Palhang, M. (2015). MCPN, octree neighbor finding during tree model construction using parental neighboring rule. *3D Research*, 6(3).

Nosrati, M., Karimi, R., and Hasanvand, H. A. (2012). Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, pages 251–256.

Noto, M. and Sato, H. (2000). A method for the shortest path search by extended dijkstra algorithm. In *SMC 2000 Conference Proceedings. 2000 IEEE International Conference on Systems, Man and Cybernetics. 'Cybernetics Evolving to Systems, Humans, Organizations, and their Complex Interactions' (Cat. No.00CH37166)*.

Nourian, P. and Zlatanova, S. (2015). Big data analytics in the geospatial domain. https://3d.bk.tudelft.nl/projects/bigvoxels/. Accessed: 2028-9-15.

Oracle (2014). How to improve database performance using database smart flash cache. http://www.oracle.com/technetwork/articles/servers-storage-admin/database-flash-cache-linux-2199720.html. Accessed: 2015-10-26.

Payeur, P. (2006). A computational technique for free space localization in 3-D multiresolution probabilistic environment models. *IEEE Trans. Instrum. Meas.*, 55(5):1734–1746.

Peters, R., Ledoux, H., and Biljecki, F. (2015). Visibility analysis in a point cloud based on the medial axis transform.

Samet, H. (1988). An overview of quadtrees, octrees, and related hierarchical data structures. In *Theoretical Foundations of Computer Graphics and CAD*, pages 51–68.

Samet, H. (1989). Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, 45(3):400.

Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

Schön, B., Mosa, A. S. M., Laefer, D. F., and Bertolotto, M. (2013). Octree-based indexing for 3D pointclouds within an oracle spatial DBMS. *Comput. Geosci.*, 51:430–438.

Schütz, M. (2015). Potree 1.3. http://potree.org/. Accessed: 2015-NA-NA.

ThreeJS (2015). Three.js - javascript 3D library. http://threejs.org/. Accessed: 2015-10-21.

Threejs (2015). Three.js examples. http://threejs.org/examples/. Accessed: 2015-9-NA.

Turner, E. and Zakhor, A. (2013). Watertight planar surface meshing of indoor Point-Clouds with voxel carving. In *2013 International Conference on 3D Vision*.

van der Spek, S. and Verbree, E. (2015). Geomatics synthesis project 1.1.1 project guide fall 2015-16: Direct computing with explorative point clouds. Geomatics for the Built Environment; Faculty of Architecture and the Built Environment.

Verbree, E. and van Oosterom, P. (2014). Explorative point clouds maps for immediate use and analysis.

Verbree, E. and van Oosterom, P. J. M. (2003). The stin method: 3d surface reconstruction by observation lines and delaunay tens. In *Proceedings of ISPRS Workshop on 3D-reconstruction from airborne laserscanner and InSAR data, Dresden, Germany*.

Vörös, J. (2000). A strategy for repetitive neighbor finding in octree representations. *Image Vis. Comput.*, 18(14):1085–1091.

W3Schools (2015). HTML5 web workers.

Wang, M. and Tseng, Y. (2011). Incremental segmentation of lidar point clouds with an octree-structured voxel space. *The Photogrammetric Record*, 26(133):32–57.

WebGL (2011). What is WebGL? https://www.khronos.org/webgl/wiki/Getting_Started. Accessed: 2015-11-19.

Wikipedia (2015). MoSCoW method. https://en.wikipedia.org/wiki/MoSCoW_method. Accessed: 2015-10-9.

Xu, S., Honegger, D., Pollefeys, M., and Heng, L. (2015). Real-Time 3D navigation for autonomous Vision-Guided MAVs. *IROS*.

Zhou, K., Gong, M., Huang, X., and Guo, B. (2011). Data-Parallel octrees for surface reconstruction. *IEEE Trans. Vis. Comput. Graph.*, 17(5):669–681.

11 Appendix

11.1 MoSCoW

The last column is coloured by result; green for fully achieved, orange for not achieved and grey for not relevant.

Must	**Have a usable point cloud dataset**	Must	Acquire an indoor point cloud dataset	
		Should	Acquire a point cloud of the Bouwpub	
		Could	Acquire a point cloud of the interior of the entire faculty of Architecture	
		Would	Acquire a coloured point cloud and a scanning route of the interior of the faculty of Architecture	
	classify empty space	Must	Classify the empty space using **any kind** of point cloud	
		Should	Classify the empty space using **any kind** of point cloud in an automatic way	
		Could	Classify the empty space using the route information of the Zep1 laser scanner	
		Would	-	
	Validation of classified empty space	Must	Validate empty space by hand, comparing classification result with reality	
		Should	Validate empty space automatically	
		Could	Validate classified empty space by comparing to result by other methods, for example with the route information of the zeb1 laser scanner	
		Would	Checking whether scanner route is always within the empty space. Empty space without scanner route in it can be dropped as they are most likely outside of the building	
	Research methods for visualising point cloud	Must	Compare papers of different methods	
		Should	Download and test different software	
		Could	-	
		Would	-	
	visualisation of empty space	Must	visualise the empty leafs of the segmented octree	
		Should	visualise the empty leafs of the segmented octree in a web based application	

		Could	visualise the empty leafs of the segmented octree as a merged geometry
		Would	-
Should	**Media Campaign**	Must	Publish the end results online
		Should	Communicate regular updates on the progress of the project
		Could	Tweet on a daily basis to attract attention
		Would	-
	Store point cloud and empty space in the database	Must	Store the point cloud in a database
		Should	Store the point cloud and the empty space in a database
		Could	Improve performance by spatial indexing
		Would	Have the database on a server (enabling easy access also outside the university)
	Create a route finding algorithm on top of the empty space	Must	create algorithm to find a route through octree of empty space
		Should	visualise route
		Could	-
		Would	create algorithm to find a route through empty space depending on space requirements of movable object
Could	**Object fitting through empty space**	Must	find where an object can fit through octree of empty space
		Should	create a visualisation of the space where the object fits through
		Could	find where an object can fit through a point cloud
		Would	-
	Alternative method to find the empty space of a point cloud using the route of the scanner	Must	finding the empty space with a 2D approach, every space on a direct line between route and point (at a cut through) can be seen as empty space
		Should	-
		Could	do it in 3D
		Would	-
Would	**Volume calculation**	Must	Calculate volume of empty space
		Should	-
		Could	-

		Would	-	
	Research how to visualise only the visible points	Must	Compare papers of different methods	
		Should	Compare and test different methods with test data	
		Could	Compare and test different methods with our own data	
		Would	-	
	visualisation of only visible points	Must	A solution for visualising point clouds in the browser has to be implemented	
		Should	A solution for visualising point clouds in the browser has to be implemented and non-visible points from the user's perspective are hidden	
		Could	A solution for visualising point clouds in the browser has to be implemented and non-visible points from the user's perspective are hidden and can only move through empty space	
		Would	User can upload their own data and visualise it using our application, doing all of the above	
	Oculus Rift walkthrough	Must	make visualisation of only visible points suitable for Hardware	
		Should	-	
		Could	-	
		Would	-	

11.2 Costs Overview

Budget Zeb1

Leap3D	Costs	Amount	Total
Scanner/day	300	1	300
Meter costs	0.225	100	22.5
Insurance/day	50	1	50
Instruction	300	0	0
total ex btw			*372.5*
total incl. 21% btw			450.725

Budget Leica Scanner

Leica Scanner	Costs	Amount	Total
Scanner/day	0	0	0
loan per hour	30	3	90
Total			90

11.3 FMW Workbench Worksheet

This worksheet loads the points and makes the boxes of the path out of the origin and leaf size of the box.