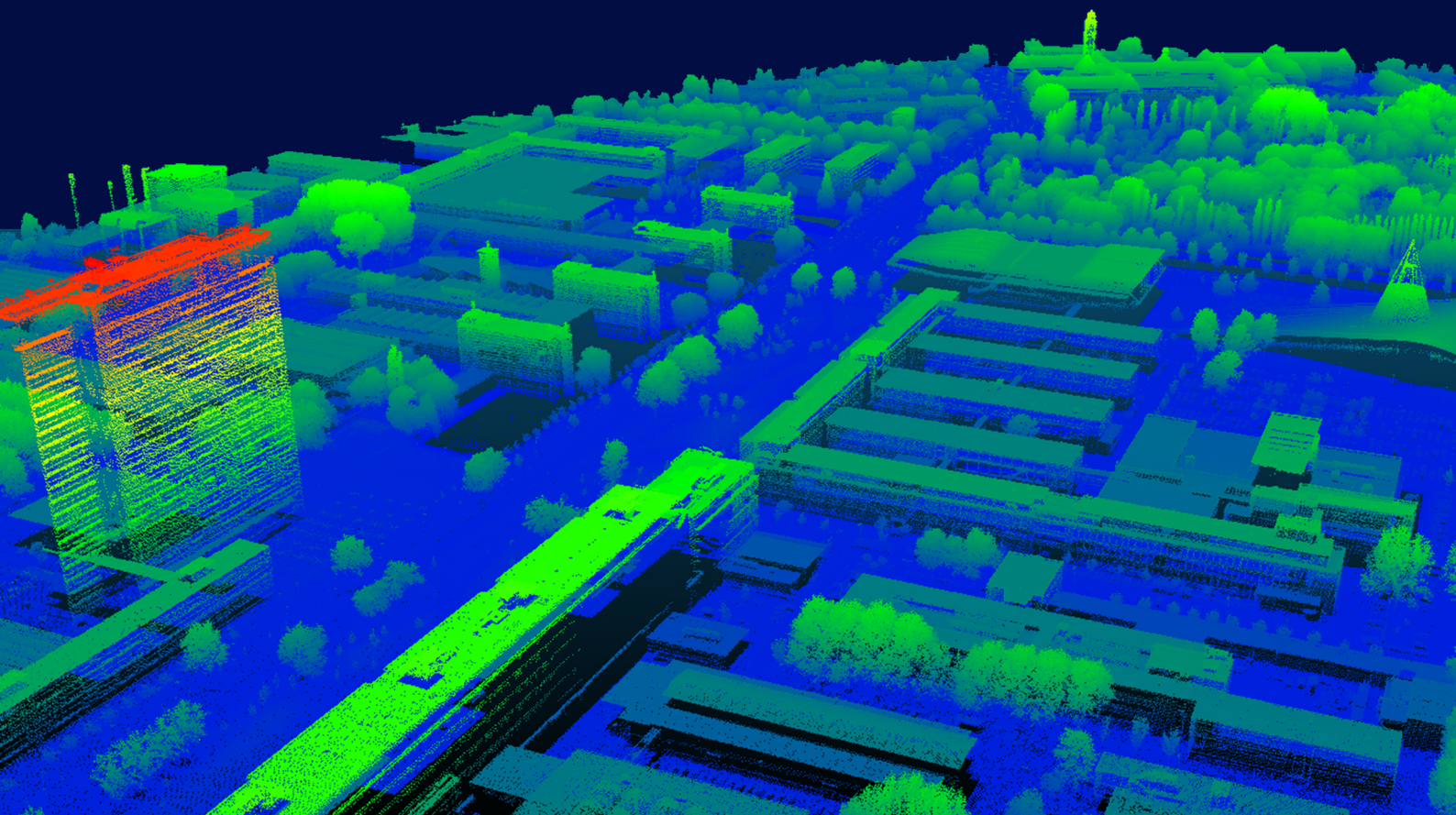
TU Delft

Delft University of Technology

MSc Thesis in Geomatics
Faculty of Architecture and the Built Environment

Visibility Analysis in a Point Cloud based on the Medial Axis Transform

Teng Wu
October 2019



VISIBILITY ANALYSIS IN A POINT CLOUD BASED ON THE MEDIAL AXIS TRANSFORM

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of

Master of Science in Geomatics for the Built Environment

by

Teng Wu

October 2019

The work in this thesis was carried out in the:

3D geoinformation group
Department of Urbanism
Faculty of the Built Environment & Architecture
Delft University of Technology

Supervisors:	Dr. Ravi Peters
	Dr. Hugo Ledoux
Co-reader:	Dr. Martijn Meijers

ABSTRACT

This thesis proposes a novel medial axis transform (MAT) based method to achieve visibility analysis in a point cloud. There are several advantages of this MAT based method. This method avoids surface reconstruction from a point cloud. It also works for the situation when there is surface missing in the input point cloud. For different point cloud datasets such as point cloud generated from meshes, AHN3 point cloud and point cloud generated from dense image matching, this method successfully deliver decent visibility result for all of them.

The main challenge overcome in this thesis is the interior and exterior MAT separation. Two approaches, normal reorientation approach and bisector based approach are experimented in the thesis to separate MAT. The normal reorientation approach only works for point cloud generated from meshes. The bisector based approaches works for all the datasets testes. It successfully separates the interior and exterior MAT when there is surfacing missing.

To speed up of the query process, spatial index is generated for interior MAT. In this thesis, two spatial indices are implemented, KD-Tree and R-Tree. Due to the limitation of my KD-Tree implementation, the KD-Tree does not improve the running speed obvious. There is a room to improve the KD-Tree implementation. The R-Tree achieves sharply improvement on running time of the queries. For 51567 points, the query based on R-Tree finished in 1880 ms.

In a word, this thesis proposed an efficient MAT based method of visibility analysis in a point cloud.

ACKNOWLEDGEMENTS

First of all, special thanks to my supervisors Ravi Peters and Hugo Ledoux for their guidance and help during my whole graduation. We had many discussions about the topic where I learnt a lot. Especial gratitude to Ravi, who never hesitates to help me whenever I met problems and always guides me into a correct direction. Thanks Martijn Meijers for being my co-reader and provide me much valuable feedback.

I would like to thanks Jantien Stoter and Hugo Ledoux for providing me a part-time job as a student assistant. During this assistant job, I learnt many skills about programming and compiling, which is the technique support for my graduation. Thanks Hugo for teaching me the rigorous attitude to my work and study.

Many thanks to all the teachers in Geomatics. They teach me how to find out issues and solve them in multiple fields. Special thanks to the staffs in 3D geoinformation group. Thanks Liangliang Nan, Weixiao Gao for their generous help on the technique part.

Thanks all my fellow classmates and friends in Geomatics. Thanks Erik, Pim, Nikos, Vasileious and Davey for their support and accompany in the entire two-year Master program of TU Delft.

I would like to show me gratitude to all my friends in TUD, thanks Qu Wang, Shenglan Du, Xin Wang, Yixin Xu. Thanks Zhenheng Kong, Yijun Wang, Hongjie Huang, Qianwen Tang, Meiqi Liu and Yu Zhang.

Thanks the all the guys in top level design group in Wuhan University. Thanks to Weizhou Li.

Special thanks to Farah for her consistent support and making the cover for me. And to Erik for correcting this thesis.

Finally, I would like to thanks my parents and friends, who always stand besides me, support me and encourage me without hesitation. Thanks to all the challenges I met during these two years.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

1 INTRODUCTION

Visibility analysis is to figure out which objects or parts of object can be clearly seen from a specific viewpoint. It describes spatial relationships between the viewpoint and targets. In other word, it describes if there are obstacles between the studied objects. For instance, when one stands in front of a house and the target is the house. The front facade of the house is visible. But if there is a tree or a car located in between of the person and the house. Only part of the facade can be seen by the person. For this scenario, the goal of visibility analysis is to analyse which parts of the house are visible.

Visibility analysis plays an important role in multiple fields and application. For example, it contributes to calculating the visibility of landmark features [Bartie et al., 2010], providing performance characterization of multi-camera systems [Mittal and Davis, 2004] and driver support systems [Brookhuis et al., 2008]. In 3D GIS fields, visibility analysis is one of the prominent use cases [Peters et al., 2015]. For example, it benefits to calculate the sky view factor [Watson and Johnson, 1987] of the city and measure the view from a window.

Point clouds are one of the most common data type when it comes to 3D GIS data, since its efficiency of providing geoinformation. For example, one can generate a point cloud of a house by laser scanning. A large-scale point cloud can also be obtained by using airborne laser. The popularity of point cloud has been increased in the last decade[Levoy, 2000]. With the advance of the point clouds acquisition technology, point clouds has been widely obtained and used in different applications.

Visibility analysis plays an important role in point cloud applications, such as 3D city modelling, trajectory planning for micro air vehicle and indoor navigation [Peters et al., 2015; Ramon et al., 2014; Biswas, 2012]. However, it is not an easy task due to that points are zero-dimension objects mathematically, which means points do not have size. Therefore, a point will not block the view on the rest of point cloud theoretically.

In order to achieve the visibility analysis, methods have been developed in the past two decades. The first kind of method is voxel-based method. It first generates 3D voxels from a point cloud. Then ray-box intersection is used to separate visible boxes and obstructing ones. Williams et al. [2005] proposed an efficient box-ray intersection method to improve this process. The main limitation is that the result are strongly related to the voxel size [Alsadik et al., 2014]. The second type of method is mesh-based method. The idea is to generate 3D triangle meshes from a point cloud first. Then ray-triangle intersection is used to check if one mesh is blocked by the rest of meshes[Greene et al., 2005]. After this intersection query, visible meshes can be extracted. In practise, it is difficult to create a perfect 3D mesh generation from a point cloud due to noises and points missing of the input. Besides, mesh generation often requires additional information, such as normals and sufficient density of the input point cloud [Alsadik et al., 2014]. The third kind of method is splatting-based methods. Instead of creating triangle mesh surfaces, this method creates splatting surfaces. Then ray-splat intersection is used to detect obstructing splats. The splats are depended on radius size which is often a user-defined value for the whole dataset. It is difficult to handle surface missing of the input. The fourth method is the hidden point removal method [Katz et al., 2007]. The method first transforms input points by a spherical flipping. Then this method generates a convex hull which contains the centre of the transform sphere

and transform points. At the end the visible points can be extracted from the convex hull. The limitation of this method is that it is difficult to handle noises and the performance is depended on the radius of flipping sphere.

The medial axis transform is first proposed by Blum [1967] to describe biological shapes. It can generate the skeleton from the input. From a point cloud, the medial axis transform is used to generate the skeleton which consists of faces theoretically. In this thesis, the skeleton of point cloud is represented as groups of points due to the generation algorithms. More details of MAT are explained in Section 2.1.

Peters et al. [2015] proposed a medial axis transform based approach to achieve visibility analysis of a point cloud. However, in this approach, an apparent limitation is that a depth map needs to be generated of each view point. Due to the limitations of the methods mentioned above, it is significant to find a new solution. This thesis presents a new approach based on the medial axis transform to achieve visibility analysis. Instead of the depthmap, ray tracing is used for visibility query in this thesis. To achieve efficient performance, KD-Tree and R-Tree spatial indices are used.

1.1 RESEARCH QUESTIONS

This thesis explores a medial axis transform based approach, to achieve visibility analysis on a point cloud. This is a new solution compared to these methods mentioned above. The main research question is the following:

How can the Medial Axis Transform (MAT) be used for visibility analysis in a point cloud?

Efficiency of this method can be reflected in a program running speed and computation cost of the whole process. This research is an experiment to check if this method works well for visibility analysis. Besides, it is also significant to find out the performance and properties of this method. More details and interesting points will involve in the following sub-questions.
Visibility query

- What is an efficient spatial index for storing MAT ball?
- How to handle multiple viewpoints and different directions?

MAT properties

- How to separate interior and exterior MAT?
- What are the advantages and disadvantages of the MAT based method?

Validation

- How good are the results of this method compare to existing methods?
- Does this method work for different datasets (e.g. AHN3, EMC-Rotterdam and Dense point cloud from image matching)?
- Does this method work for different objects types of point cloud?

1.2 RESEARCH SCOPE AND OBJECTIVES

This thesis focuses on the MAT based method itself. The properties and performance of the MAT based method are discussed in detail. Other methods have less importance in this thesis. The performance of the MAT based method is reflected on

two aspects, efficiency and accuracy. Efficiency of this method is reflected in the program running speed and computation cost of the whole process. For the accuracy, comparisons between the MAT methods and the hidden point remover method are made. Advantages and limitations of the MAT method are deeply discussed in this thesis.

MAT created from a point cloud can be classified into two classes, exterior and interior MAT. The exterior MAT will not contribute to this thesis, since it will block the view from viewpoint to the input. Rather, the exterior MAT has negative contributions to visibility analysis. Therefore, how to separate the interior and exterior part is a key task in this research. Different approaches of the MAT separation are proposed in this thesis. Their performance and limitations are included.

For experimental dataset, multiple point cloud datasets are tested in this research. Noises and surface missing exist in the input datasets. Whether the MAT method is able to handle this situation is discussed. The influence of different quality input data is covered. But the correction of the input point cloud is not involved in this topic.

1.3 THESIS OUTLINE

This thesis is structured as the following chapters:

- Chapter 2 introduces related theories of the point cloud visibility research. It contains theory of the Medial Axis Transform, KD-Tree, R-Tree and the normal of a point cloud. These theories give fundamental supports of occlusions analysis and efficient visible points query of a point cloud.
- Chapter 3 gives an overview of current approaches of point cloud visibility research. Limitations of these methods are discussed.
- Chapter 4 explains the methodology of this research. It illustrates the architecture of this MAT based method. The challenge in this research and potential solutions are discussed.
- Chapter 5 focuses on the implementation of the MAT based method. It transfers ideas to codes. Tools and datasets are included in this chapter.
- Chapter 6 shows results, discussions and validations of the MAT based method. Besides the visibility results of different datasets, it is discussed how to evaluate the approach.
- Chapter 7 answers all the research questions. Conclusions and ideas for future work are given.

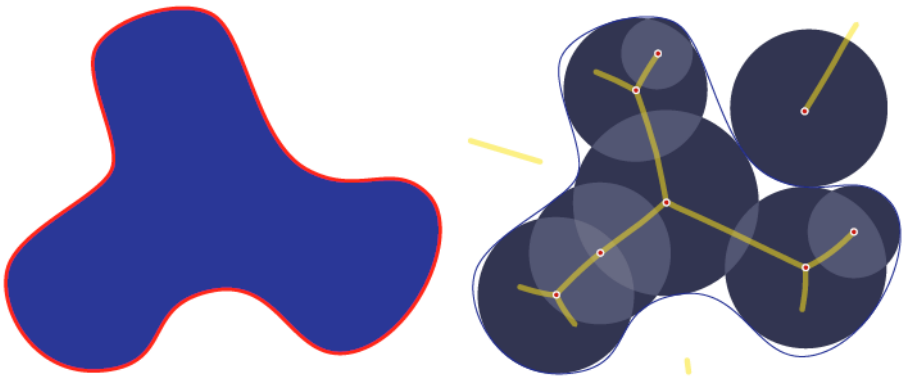
2 THEORY

2.1 MEDIAL AXIS TRANSFORM (MAT)

Blum [1967] proposed the Medial Axis Transform theory to describe shapes in biology. This method extracts skeleton of an input shape both inside and outside. It divides the space of the input into partitions. By applying this method to point cloud, it is able to extract 3D volume information from zero-dimension points. The 3D volume information contributes to visibility analysis, since occlusions caused by 3D volume can be used to determine which subdivision spaces are visible.

2.1.1 2D MAT

The 2D MAT of a 2D point set consists of groups of surfaces which have certain distances to the boundary. Figure 2.1 illustrates what is the 2D MAT result of an input 2D shape. The 2D MAT result is groups of surfaces which make up the yellow lines in the Figure b. Medial circles are drawn in Figure b in black. The element of MAT can be considered as the centres of the medial circles which tough the boundary at one or more points. There are two kinds of MAT, interior MAT and exterior MAT. As shown in figure 2.1, the difference is that whether they are located inside the input shape or not.



(a) A shape

(b) Corresponding medial axis and some medial circles

Figure 2.1: 2D MAT of a 2D shape
[Peters et al., 2015]

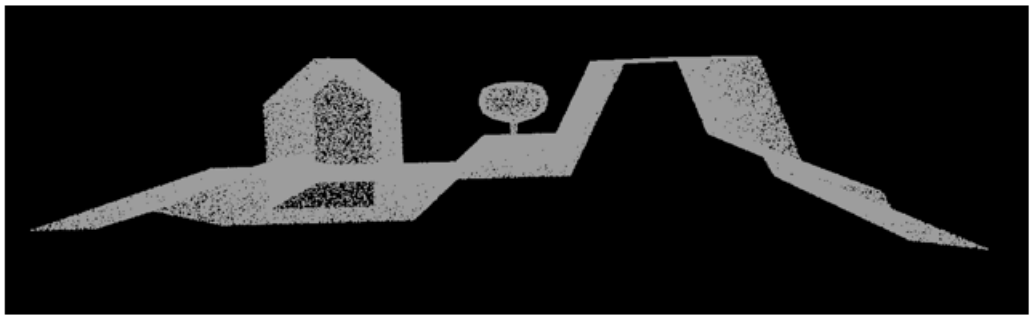
2.1.2 3D MAT

Similar to 2D MAT, 3D MAT is groups of surfaces as well. For the points in the surfaces, medial balls can be drawn with certain radius which equals the distance from MAT point to the boundary. The centres of these medial balls can be considered as 3D MAT result of the input. These centres make up the MAT surfaces.

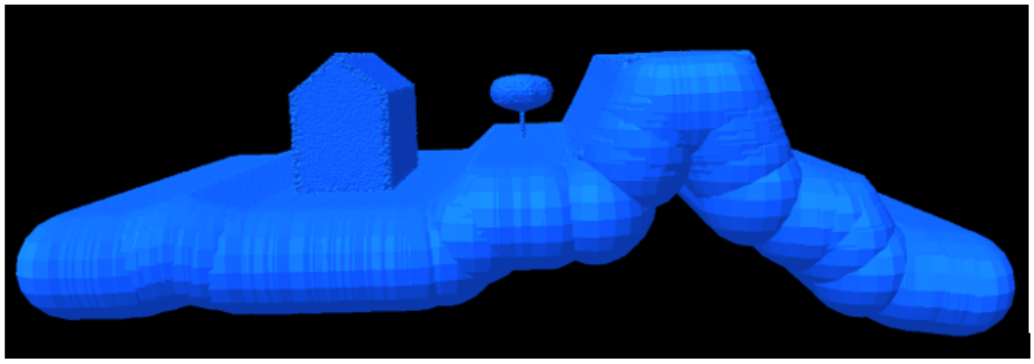
Figure 2.2 illustrates the 3D MAT of a surface model of a hand. The 3D MAT can be considered as the skeleton of the input model.

Figure 2.2: 3D MAT of a surface model of a hand
[Amenta and Kolluri, 2001]

An example of 3D MAT of a point cloud is given in Figure 2.3. The input point cloud is an artificial point cloud which contains a house, a tree and a terrain (see Figure 2.3 a). The 3D interior MAT is shown in Figure 2.3 c (in red). It is represented by a number of points. And these points are the centres of the medial balls shown in Figure 2.3 b. The 3D medial balls are shown in Figure 2.3 b (in blue). The balls perform 3D volume information of the input point cloud. And they are able to block the sight of view from a viewpoint, which contributes to visibility analysis. The key information of visibility analysis is to determine if the target is visible or blocked.



(a) An artificial point cloud



(b) 3D medial balls with initial Radius 1.0 m

(c) Interior MAT (in red) and the input point cloud

Figure 2.3: 3D MAT of a point cloud

2.1.3 Interior MAT and exterior MAT

There are two kinds of MAT of a point cloud, interior MAT and exterior MAT. Their difference is the relevant positions to the boundary of input. The interior MAT is located inside the boundary and exterior MAT is outside. When the input is a closed object (see Figure 2.1 b), it is obvious to distinct the interior MAT and the exterior MAT. However, point cloud data is not closed, so there is not clear border of inside and outside . But the interior MAT and exterior MAT can still be separated by their position to the input point cloud. The interior MAT and exterior MAT always located on the different sides of the input point cloud (see Figure2.4). In this thesis, the MAT located on the side of lower elevation is defined as interior, and the MAT on the side of higher elevation is defined as exterior. In short, the interior MAT (Figure 2.4 in red) is under the input. The exterior MAT (Figure 2.4 in blue) is above the input.

When there is a surface missing of the input point cloud, the interior MAT can have connection with exterior parts (see Figure 2.5 in orange). The connected MAT

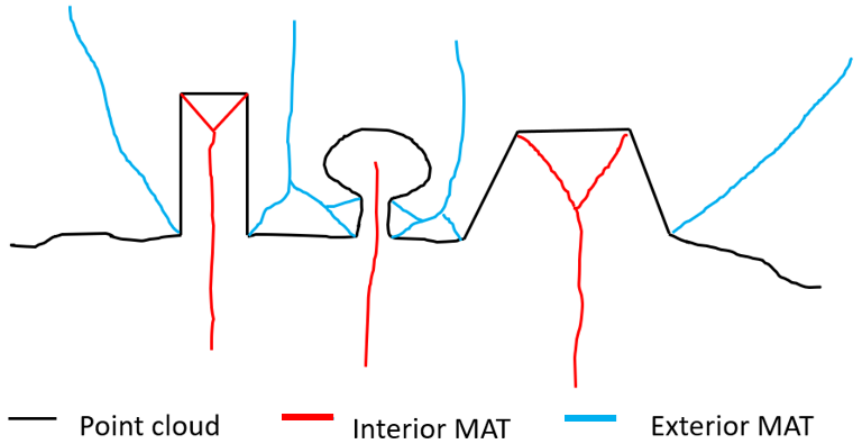


Figure 2.4: Interior and exterior MAT of a point cloud

exists in the situation of holes or surface missing or low density point cloud. In practice, the quality of the input point cloud is often not perfect, so the ambiguity of interior MAT and exterior MAT exists. The exterior MAT can block the input point cloud, which will result in wrong visibility results. Only interior MAT benefits visibility analysis. It is a significant task to separate the interior MAT and exterior MAT properly.

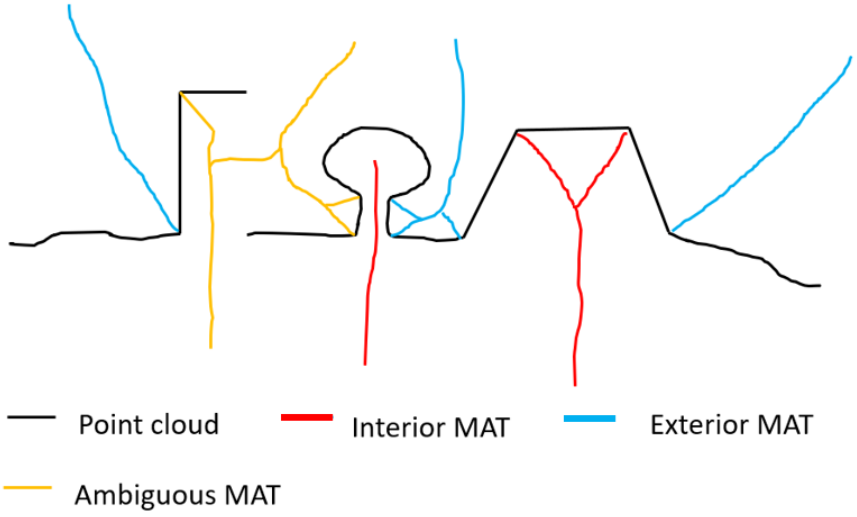


Figure 2.5: Interior and exterior MAT of a point cloud with a surface missing

2.2 KD-TREE

KD-Tree, first proposed by Bentley in 1975, is a data structure aiming at associative searching. It divides 3D space into subdivisions. The elements of KD-Tree consists of nodes and leafs. Figure 2.6 is an example of the subdivisions process of a 2D space. The process is described as following. First find a middle point along the K_0 axis (horizontal direction). Then divide the space into two halves perpendicular to K_0 axis at point A. For each subdivision, find a middle point along the K_1 axis (vertical direction). Divide the subdivision into halves at that middle point perpendicular to K_1 axis. Repeat the steps above, until no partition space can be divided. Figure 2.7 gives a planar tree graph representation of the 2D KD-Tree. It

contains the connection relationship between nodes and leafs. Points which belongs to different subdivisions are stored in the corresponding levels of the KD-tree.

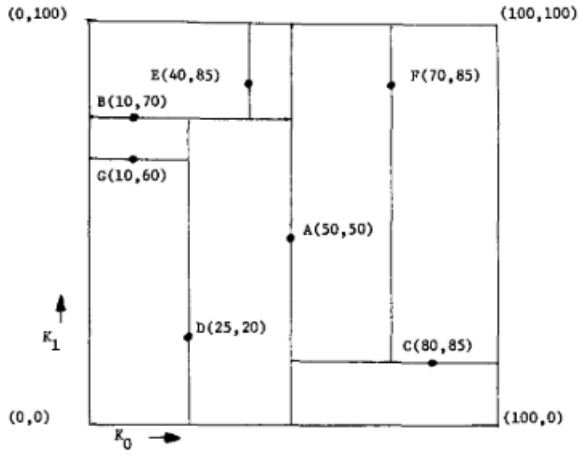


Figure 2.6: A 2D example of a KD-Tree
Bentley [1975]

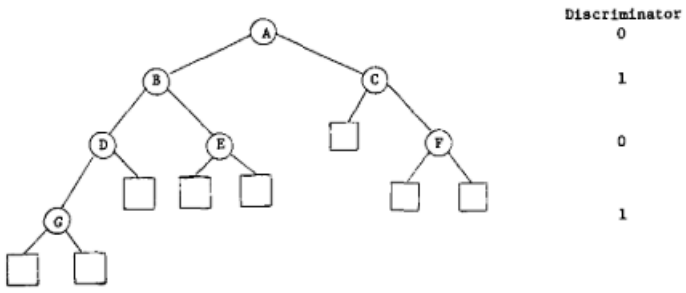


Figure 2.7: Planar graph representation of the 2D KD-Tree
Bentley [1975]

KD-Tree is a useful approach to find the nearest neighbour. Each node or leaf has a nearest partition space. This information is stored in the KD-Tree structure. Since the visible point is always the nearest one from the viewpoint along that view direction, KD-Tree is suitable data structure to create spatial index for the point cloud.

2.2.1 Process of finding the nearest node in the KD-Tree

The process of finding the nearest node of a KD-Tree is described in this Section. Figure 2.8 shows a 2D point set, viewpoint and the KD-Tree of the point set. Now the task is to find out which point in the point set is the nearest to the viewpoint, (-2, 12). First, start with the root of the KD-Tree, (7, 2).

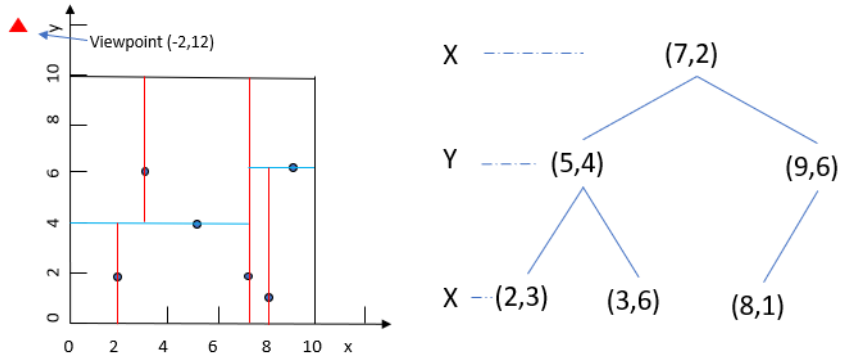


Figure 2.8: A 2D points set and its KD-Tree

Now the nearest node should lie on either the right partition of the root or the left. After comparing node (5, 4) and (9, 6), we know that the left node (5, 4) of the root is the closer to the viewpoint. So right partition of the root can be eliminated (see Figure 2.9). Then we continue doing the same comparison between node (2,

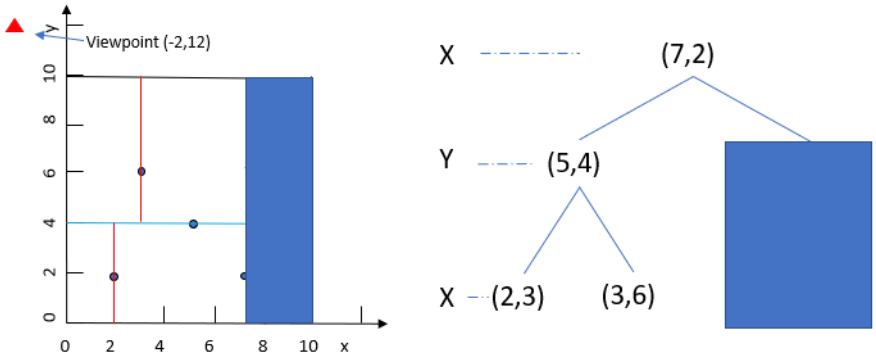


Figure 2.9: Nearest node query of the KD-Tree, step 2

3) and (3, 6). Another partition in the bottom can be eliminated (see Figure 2.10). Finally the nearest node (3, 6) is found.

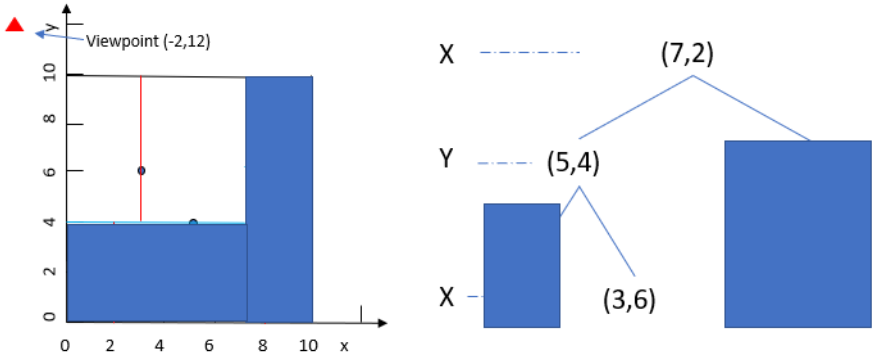


Figure 2.10: Nearest node query of the KD-Tree, step 3

2.2.2 KD-Tree with buckets

In the past two decades, KD-Tree theory has been exploring deeply. Arya and Mount [1993] proposed an adapted version of KD-Tree which makes the nearest neighbour query even faster by involving the concept of bucket size (see Figure 2.11). In the adapted version, there are two kinds of KD-Tree nodes, leaf nodes and intermediate nodes. Data is only stored in the leaf nodes. A rectangle associated with leaf nodes are called buckets which defined subdivision of space into rectangle [Arya and Mount, 1993]. Data located in the rectangle is stored in the corresponding leaf nodes. There is no data stored in the intermediate nodes. Arya and Mount [1993] stated that this version of KD-Tree achieved improvements in running time compared to a KD-Tree without bucket, due to the total depth of the tree is reduced.

In this thesis the KD-Tree is implemented with a bucket size 16. (see Section 4.4)

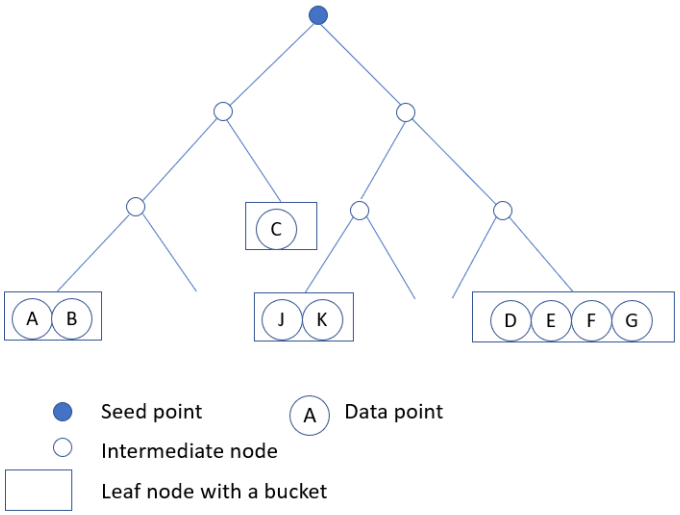


Figure 2.11: Planar graph representation of the a KD-Tree with buckets

2.3 R-TREE

Besides KD-Tree, R-Tree is another data structure for spatial queries. It was first invented by Guttman [1984]. R-Tree is able to divide the space into several smaller subdivisions. The items stored in R-Tree are represented by their minimum bounding rectangles (boxes in 3D). These minimum bounding rectangle containing position information benefits spatial queries. The number of items need to be checked can be reduced during spatial queries, since the uninterested subspaces corresponding to uninterested rectangles can be skipped. There are some specific requirements for an R-Tree.

- "Each leaf node contains m to M records unless it is the root.
- For each record (I, *tuple-identifier*) in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data.
- Every non-leaf node has between m and M children unless it is the root.
- For each entry (I, *child-pointer*) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level." Guttman [1984]

Figure 2.12 shows a 2D R-Tree example. The total level size of the R-Tree is 3. Each leaf node contains 1 to 3 elements. The data (red rectangles in Figure 2.12) are only stored in the leaf nodes. The process of finding the intersected data with query rectangle (blue in Figure 2.12) is described as follows. First, check the nodes in the top level. Only $R1$ is intersected with query rectangle. Second, check the child nodes of $R1$. $R3$ is the only child node intersected. That means result is in the child nodes of $R3$. Finally, check all the child nodes of $R3$. The result is $R8$.

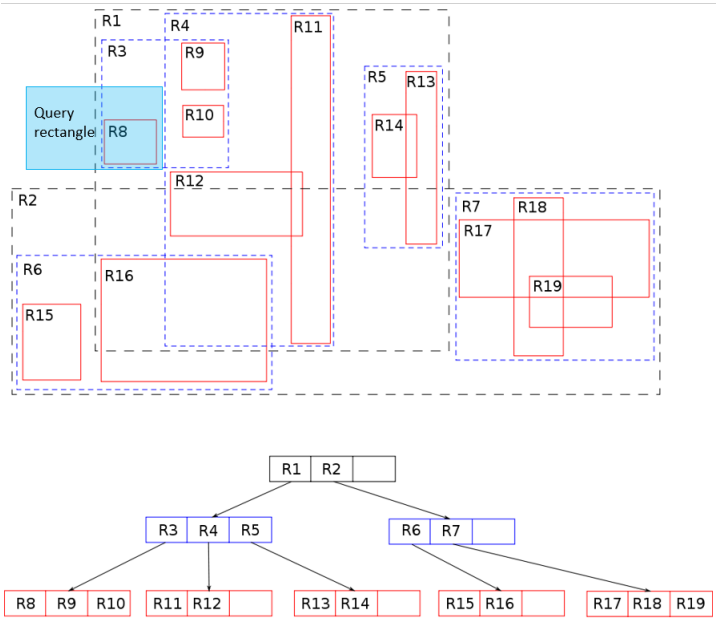


Figure 2.12: Simple example of an R-tree for 2D rectangles
Skinkie [2010]

Figure 2.13 shows a 3D R-Tree of 3D points. Instead of 2D rectangles, the bounding boxes are represented by 3D boxes. The bounding boxes divide the space into smaller partitions.

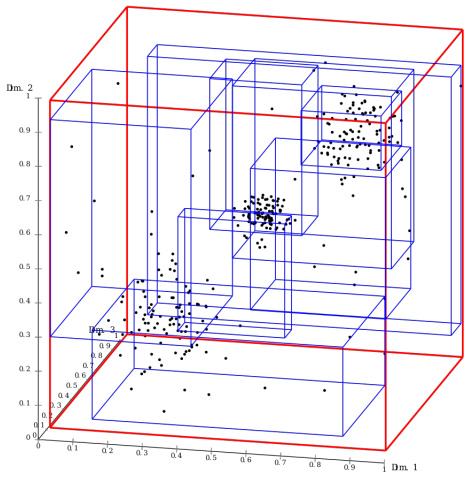


Figure 2.13: Visualization of an R-tree for 3D points
Chire [2010]

2.4 NORMAL ESTIMATION OF POINT CLOUD BY USING PRINCIPAL COMPONENT ANALYSIS (PCA)

When it comes to normal estimation, Principal Component Analysis (PCA) [Jolliffe, 2002] is one of the most popular method. It is available for example in the PCL library [PCL_Developing_Group, 2019]. The steps of normal estimation with PCA method are described as follows. For each point p in a point cloud, select k nearest neighbours and these points make up a point set S. Then compute the covariance matrix of the set S. Next, calculate the eigenvector of the covariance matrix. The normal n of point p is the smallest eigenvalue of the eigenvector (see Figure 2.14).

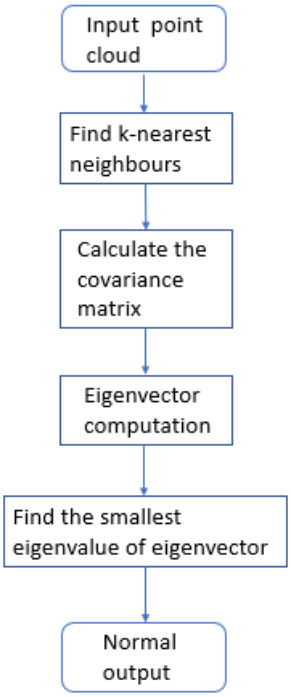


Figure 2.14: The workflow of PCA normal estimation

The limitation of the PCA method is that the result is highly influenced by the k neighbours. Different k values result in different normal results (see Figure 2.15). When the input point cloud is not sufficient dense or there are missing surfaces, the parts of the result might point to the wrong direction. Figure 2.16 example why the normal calculated by PCA can be wrong due to a surface missing. In Figure 2.16, the k nearest neighbours parameter is set to 10. In the Figure 2.16 a, the normal of point p is pointing upward. When there is a surface missing around point p (see Figure 2.16 b), the nearest k neighbours of point p is changed. And the normal calculated by PCA is pointing to the top-left direction which is wrong.

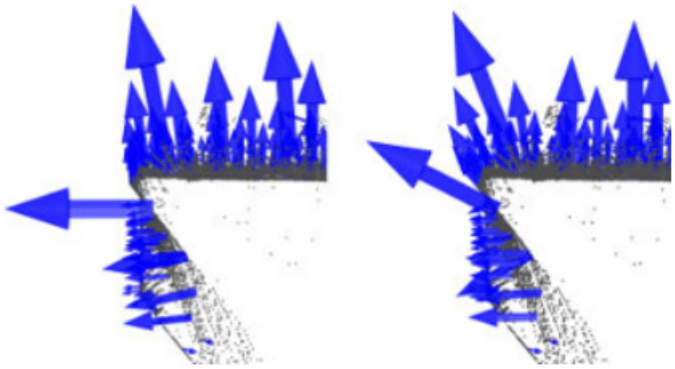


Figure 2.15: Different normal results of a small k values (left) and a large k values (right) PCL_Developing_Group [2019]

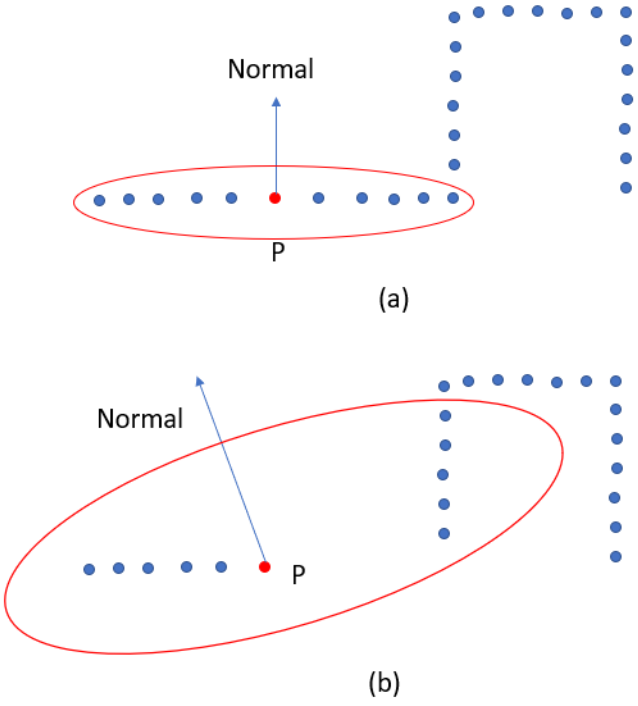


Figure 2.16: Wrong normal results of PCA method [Peters, 2017]

3 RELATED WORK

The visibility analysis of a point cloud has been studied for two decades. There are several ways to achieve that. In general there are two kinds of visibility methods. The first type is the indirect visibility method which is based on an auxiliary geometry extracted from point cloud, such as voxel-based method (section 3.1), mesh-based method (section 3.2), splatting method (section 3.3) and MAT based method (section 3.5). The other type is the direct visibility method which avoids creating intermediate geometry shapes. The direct visibility method is based on mathematical operators or transformation, such as hidden point remover. And it can directly analyse the visibility on the point cloud itself.

3.1 VOXEL-BASED METHOD

The idea of voxel-based method is to use voxels as occlusion objects, then to check which voxels are visible from a viewpoint. Assuming that voxels are already created from input point cloud, rays are emitted from one viewpoint. Each ray represents a direction of view from the viewpoint. Rays have intersections with voxels. The first intersected voxel of that ray is considered as the visible voxel of that direction. Figure 3.1 gives an overview of the visibility check based on voxels. In this figure a voxel is checked from multiple viewpoints.

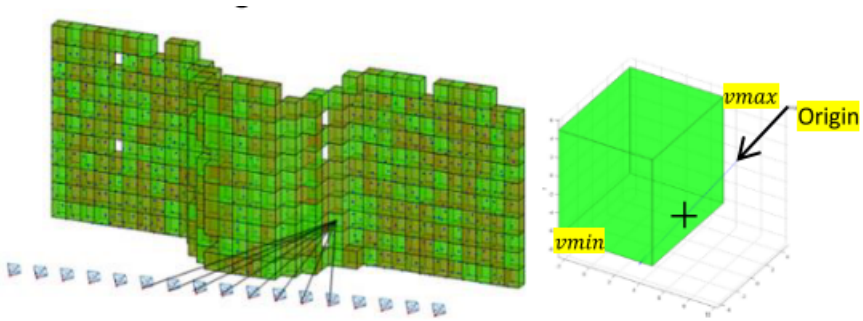


Figure 3.1: Voxel-ray intersection for visibility check [Alsadik et al., 2014]

Except the voxel generation from a point cloud, ray-box intersection is the main part for this method. Williams et al. [2005] developed an efficient Ray-Box intersection algorithm to acquire intersection result between lines and boxes. Later Mena-Chalco [2010] improved this Ray-Box intersection algorithm by adding the distance calculation from the origin to the hit point.

Although this method avoids surface reconstruction from a point cloud ([Alsadik et al., 2014]), the limitations of voxelization are obvious. Alsadik et al. [2014] stated that there were high memory requirements and computation cost when processing massive dataset. Another significant limitation is that the voxel size may have influence on the performance, especially when the input point cloud is not sufficiently dense. Figure 3.2 illustrates the situation that different voxel sizes result in different visibility performances. In Figure 3.2 a, it is the performance of a smaller voxel size. There is a hole in the front raw and the middle voxel of the back raw is visible. In

Figure 3.2 b, there is no hole in the front row and only the voxels of the front row are visible.

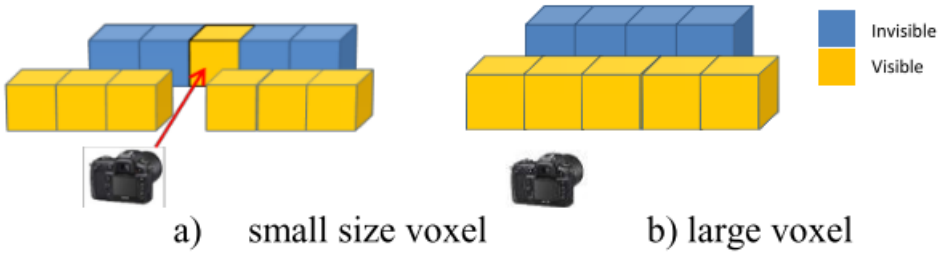


Figure 3.2: Voxel size influence
[Alsadik et al., 2014]

3.2 MESH-BASED METHOD

The concept of the mesh-based method can be described as two steps, mesh generation and visibility check. Mesh generation has already been deeply researched in the last two decades, such as Ball-Pivoting algorithm [Bernardini et al., 1999] and Poisson Surface Reconstruction [Kazhdan et al., 2006]. Therefore, this section focuses on visibility check step. There are two kinds of ways of mesh visibility check, ray-tracing method [Kaplan, 1987] and z-bufferring method [Greene et al., 2005].

The ray-tracing method is described as following. From one viewpoint rays emitted to the meshes. One ray can hit one or more meshes along the ray direction. The first mesh intersected is regarded as the visible mesh of that direction. After checking all the rays from the viewpoint, all the visible meshes of the viewpoint can be obtained. Möller and Trumbore developed a robust and efficient ray-triangle intersection method to find out the intersection result between shooting ray and triangle meshes. It reduces memory storage cost by avoiding storing triangle plane equations [Möller and Trumbore, 2005].

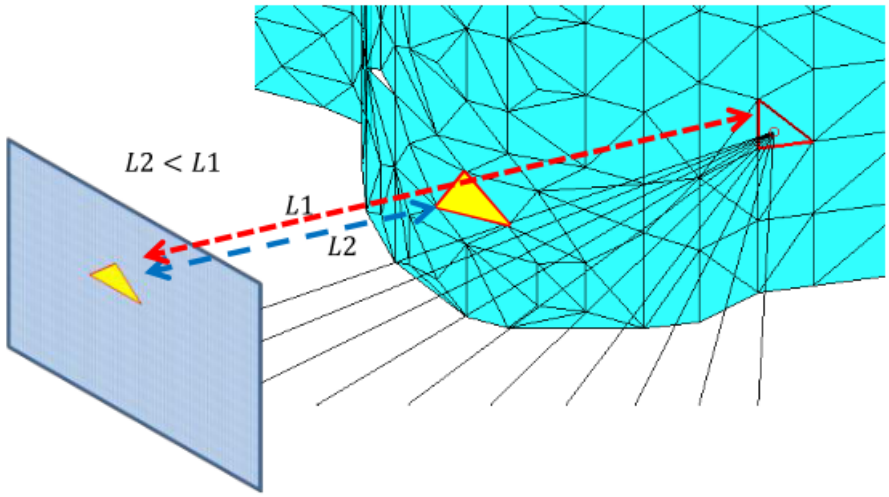


Figure 3.3: Meshes based method
[Alsadik et al., 2014]

The idea of z-buffering [Greene et al., 2005] method is to project the input objects into a 2D depth map along a certain angle. The unit pixel in the map can be represented as (x, y, z-value). The z-value stored the distance information from a target object to the viewpoint. The order of input object is corresponding to their

distance to the viewpoint. Therefore, in each pixel the object with the smallest z-value is considered as the visible one. (see Figure 3.3)

The main limitation of this method is that there is no avoidance of mesh surface generation. However, it is challenged to get perfect mesh surfaces from a point cloud, due to noises and surface missing. Mesh generation is often requires additional information such as correct normals and sufficiently density of the input point cloud, which are not guaranteed in point cloud production [Alsadik et al., 2014]. Another limitation is that every single mesh needs to be check during the visibility check, which is low efficiency in term of computation cost.

3.3 SPLATTING-BASED METHODS

Similar to mesh-based method, splatting-based method also can be described as two steps, surface splatting and visibility check. Instead of generating mesh surfaces, this method creates splatting surface. The procedure of surface splatting is described as following. First, select sample points from the input point cloud. Then calculate the normal of each sample point. Perpendicular to the normal, an elliptical or circular splat is drawn with a certain radius. Normally the radius is set to avoid holes between splats (see Figure 3.4). After the splat surfaces are generated, visible parts can be determined by geometrical intersections. The visibility check part of splatting-based methods have similarity to mesh-based method. Instead of ray-triangle intersection, ray-circle intersection is used in splatting-based method.

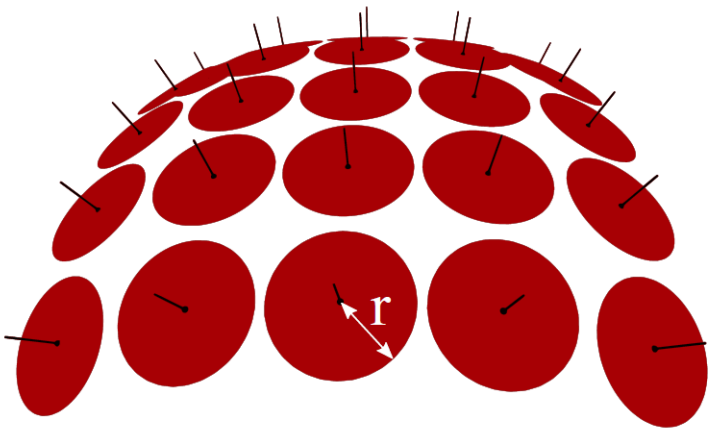


Figure 3.4: A splat is defined for each point as a normal-oriented disk with a radius r. Usually r is chosen such that there are no holes
[Peters, 2018b]

Wu and Kobbelt [2004] proposed an optimized sub-sampling method for surface splatting. The advantage of surface splating is that it is able to represent a point cloud without any connectivity or topological consistency conditions [Wu and Kobbelt, 2004]. And it has better flexibility and lower approximation error when using the same input point cloud, compared to mesh surface reconstruction [Wu and Kobbelt, 2004].

There are several limitations of the splatting-based method. First, it is a parameter-dependent method. The performance is influenced by the splatting radius. Second, the surface splatting is highly dependent on the normal of the sample points. In practise, it is difficult to get correct normal at every point in a point cloud. This method has difficulty to handle insufficient density and surfacing missing of the input.

3.4 HIDDEN POINT REMOVAL (HPR)

Hidden point removal is a direct visibility analysis method of point sets [Katz et al., 2007]. It was first proposed by Katz et al. [2007]. It consists of two steps, inversion and convex hull construction [Katz et al., 2007]. In the inversion, all the input points are projected into a spherical coordinate system. This process is called as spherical flipping [Katz et al., 2007]. Figure 3.5 gives an example of the spherical flipping process. This process projects the 2D input point cloud (in blue in Figure 3.5) into a spherical coordinate system and results in a spherical flipping point cloud (in red in Figure 3.5). The next step is convex hull construction. It creates a convex hull which consists of the transformed points and the centre of the sphere. The visible points can be determined by the following rule. A point is visible from the centre point, if its inverted point lies on the convex hull [Katz et al., 2007].

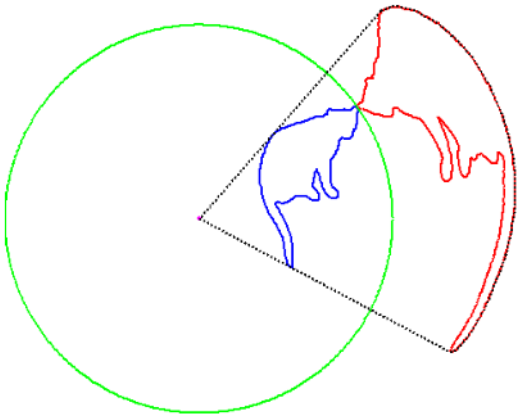


Figure 3.5: Spherical flipping (in red) of a 2D curve (in blue) using a sphere (in green) centred at the view point (in magenta).
[Katz et al., 2007]

There are advantages and disadvantages of the method. This is a direct solution for point cloud visibility analysis, which means it is able to determine the visibility without surface reconstruction or voxelization. Besides, this method is able to handle sparse point clouds [Alsadik et al., 2014]. The disadvantage is that it is necessary to set a suitable radius parameter when noises exist in the input [Mehra et al., 2010]. Another limitation is that it can only determine the points which are part of the input itself [Peters et al., 2015].

3.5 MEDIAL AXIS TRANSFORM BASED METHOD

Peters et al. [2015] proposed an approach to visibility analysis based on the medial axis transform. This method contains three steps, MAT approximating from point cloud, depthmap computation and point visibility querying (see Figure 3.6). For the MAT approximating, Peters et al. [2015] used an adapted version of shrinking ball algorithm [Ma et al., 2012]. This adapted shrinking ball algorithm is also involved in this thesis and is described in Chapter 4. To compute the depthmap, first project each the centre of medial ball into a 2D rasterized panel. In this plane, each pixel stores a current depth and the depth is updated when a new ball is projected. The visibility query process is similar to z-buffering [Greene et al., 2005] method. For each pixel, the ball corresponded to the minimal depth value is regarded as visible.

The advantage of this method is that it is able to deal with surface missing and noises of the input point cloud [Peters et al., 2015]. However, there are several

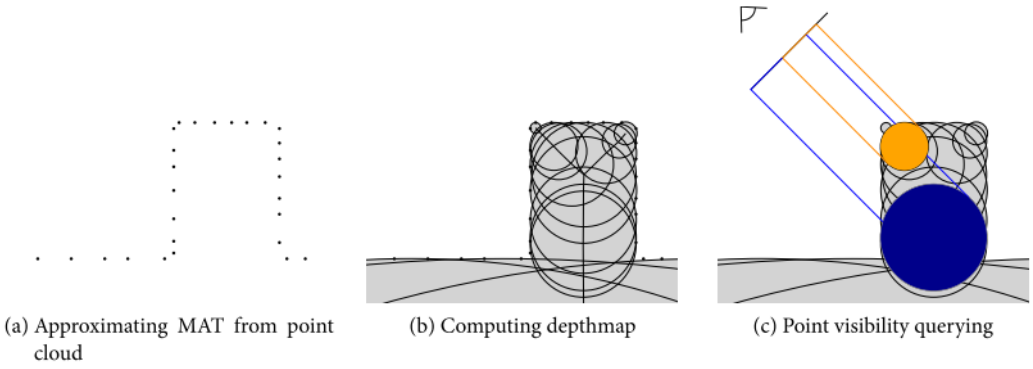


Figure 3.6: Three steps of the MAT based method [Peters et al., 2015]

limitations. First, the interior MAT and exterior MAT separation is based on perfect normals which is difficult to obtain in practice. Second, each viewpoint requires a corresponding depthmap, which means to each viewpoint an individual depthmap needs to be created.

4 METHODOLOGY

This chapter explains the methodology I developed of how to achieve visibility analysis for a point cloud based on a MAT approach. Figure 4.1 gives a summary of this MAT based approach which consists of MAT approximation, interior and exterior MAT separation, spatial index creation, ray generation and performing visibility queries (see Figure 4.1). The first step is MAT approximation which generates both interior and exterior MAT from an input point cloud (Section 4.1). As mentioned in Section 2.1.3, only interior MAT benefits visibility analysis. Therefore, the next step is to separate the interior and exterior MAT (Section 4.2). In order to achieve efficient visibility queries, a spatial index of the interior MAT is created based on the KD-Tree and R-Tree theory (Section 4.4.1). Ray generation is to model the views from a viewpoint. Two kinds of rays, parallel rays and radial rays, are taken into consideration (Section 4.6.1 and Section 4.6.2). The parallel rays from a viewpoint can be used to model solar potential applications such as solar lighting simulation and shadow analysis. The radial rays can be used to model the view sight from an origin. The final step is the visibility query. Brute force query is described in Section 4.3. The efficient KD-Tree and R-Tree based query are explained in Section 4.4.

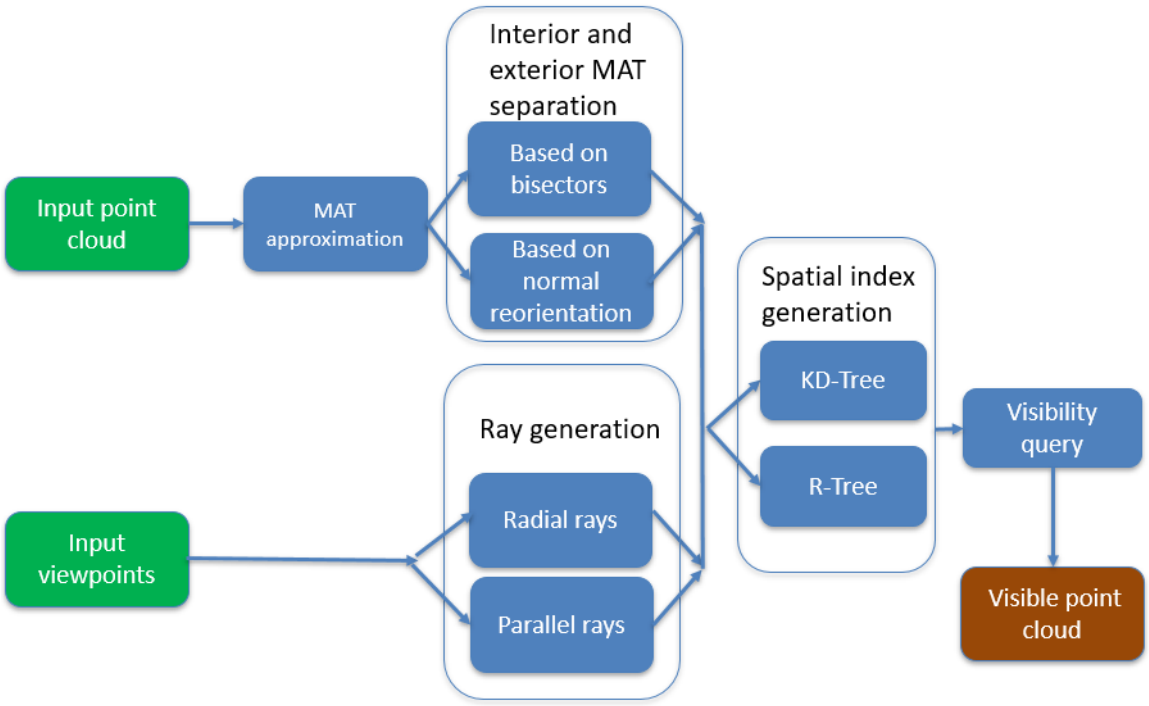


Figure 4.1: An overview of the MAT based approach

4.1 MAT APPROXIMATION

As described in Chapter 2, theoretically the medial axis transform consists of groups of surfaces which have a certain distance to the boundary. However, in this thesis the MAT is represented by groups of points generated by a shrinking ball algorithm. These points can be considered as the centres of medial balls which touch the boundary on one or more points. The original shrinking ball algorithm is proposed by Ma et al. [2012]. Peters et al. [2015] proposed an adapted version of the shrinking ball algorithm which is able to determine interior and exterior MAT by using the estimated normals. Figure 4.2 gives a 2D sketch about how the algorithm works. As illustrated in Figure 4.2, point p is a point of the input point cloud. At the beginning the MAT ball starts with a big radius and on one side it always touches the boundary at point p. Then the ball starts shrinking along the normal direction of point p. The other side of the ball may have an intersection with the point cloud at another point q. The shrinking stops when the ball's interior is empty and no closer point q is found[Peters et al., 2015].

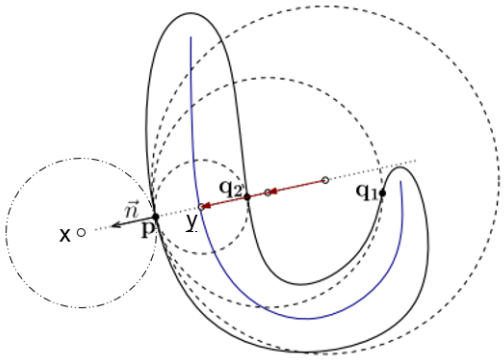


Figure 4.2: The adapted shrinking ball algorithm
[Peters et al., 2015]

Assuming that normals are correctly oriented, this version of the shrinking ball algorithm is able to separate the interior and exterior MAT by using the normals. For example, in Figure 4.2 point x is the exterior MAT result of point p. Point y is the interior MAT result of point p. The direction of vector $\vec{py}$ is opposite to the normal $\vec{n}$. Instead, the direction of vector $\vec{px}$ is the same as the normal $\vec{n}$. Therefore, if the normal of the input point cloud is always correct, the interior and exterior MAT can be determined by using the adapted shrinking ball algorithm. However, in practice it is challenged to acquire one hundred percent correct normal of a point cloud.

There are three parameters the shrinking ball algorithm: initial radius, denoise parameter and planar detection parameter. The denoise parameter and planar detection parameter aim to avoid generating noisy MAT. The idea of these parameters is to use the separation angle θ. In the Figure 4.3, the medial ball with a smaller separation angle θ_j will be skipped. And the medial ball with a larger separation angle θ_i will be generated.

The initial radius influence the results where there is only one touch point between the medial ball and the point cloud such as ground (see Figure 4.4 and Figure 4.5). Notice that the initial radius does not influence the medial ball result where the ball touches the point cloud at more than one points such as the building and the tree in Figure 4.5. It does impact the medial ball shapes where there is only one touching point between the ball and the point cloud such as the ground in Figure 4.5. More detail and discussions of the initial radius will be discussed in Chapter 7.

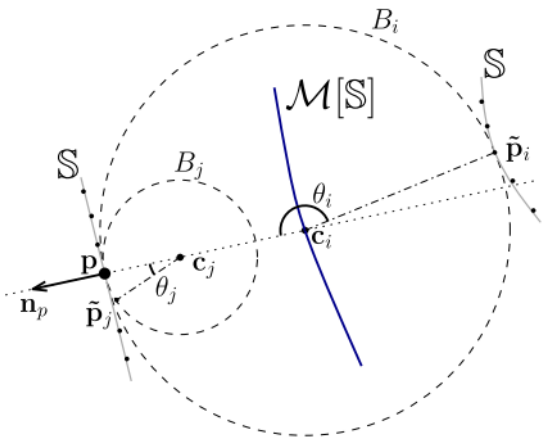


Figure 4.3: An example of the denoise parameter [Peters and Ledoux, 2016]

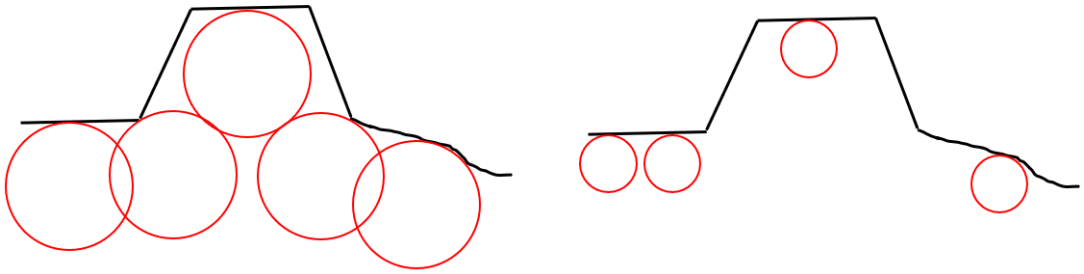


Figure 4.4: MAT balls with different initial radii

4.2 INTERIOR AND EXTERIOR MAT SEPARATION

There are two solutions to separate the interior and exterior MAT. The first one is to fix the normal orientation solution before running the shrinking ball algorithm (Section 4.2.1). If all the normals are correct, the interior and exterior MAT can be separated properly after running the adapted shrinking ball algorithm. Another solution based on the geometrical properties (bisectors) of the MAT, and it is proposed in Section 4.2.2.

4.2.1 MAT separation based on normal reorientation

Nan [2018] explains a normal reorientation approach to deal with the ambiguity in the normal direction. The concept of the method is described as following. First, build a graph that connects neighbouring point. From this graph the minimum spanning tree is computed. Then propagate the normal orientation through the graph. The following pseudo code illustrates the process of normal propagation (see Algorithm 4.1). The idea is to keep the consistency of normal directions between neighbours.

Algorithm 4.1: Normal propagation process

```
for neighbors P_i,P_j in the graph do
    if n_i^T * n_j < 0 then
        Flip n_j
```

The advantage of the normal reorientation method is that it is able to flip the normals by 180 degrees which originally point at the opposite direction. The normal

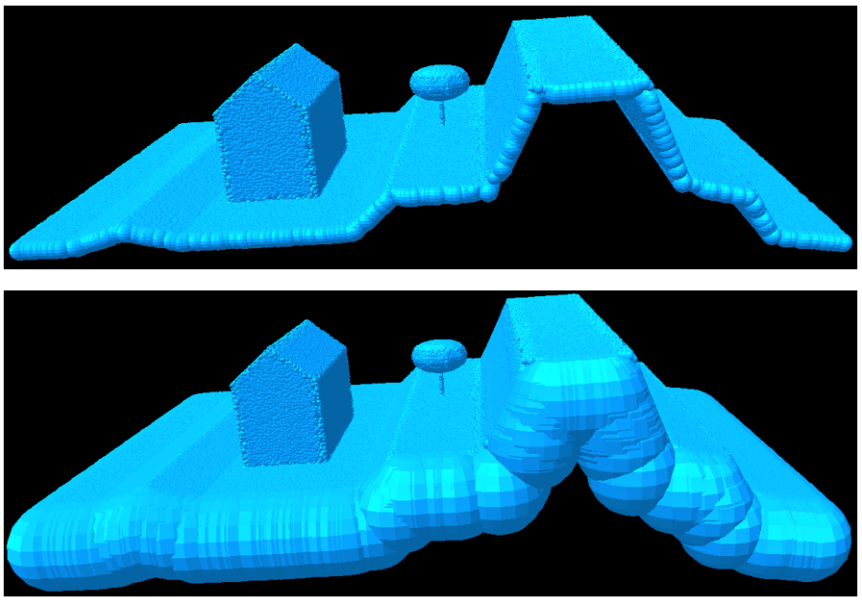


Figure 4.5: MAT balls with different initial radii

directions of a neighbour area are consistent. However, this method may fail at sharp edges or corners [Nan, 2018]. Another limitation is that it is difficult to determine which direction is the correct one. Either all the normals point at the correct direction or they point in the reverse direction. Sometimes, even though the normal has been flipped by 180 degrees, it may still point at an incorrect direction.

4.2.2 MAT separation based on bisectors

Due to the limitation of the normal reorientation solution, a new method based on the MAT geometrical properties is proposed in this thesis. The idea of this method is to use bisectors of MAT sheets to determine the interior MAT and exterior MAT. This method contains three steps, MAT approximation, MAT clustering into MAT sheets and interior and exterior MAT sheets determination (see the workflow in Figure 4.6).

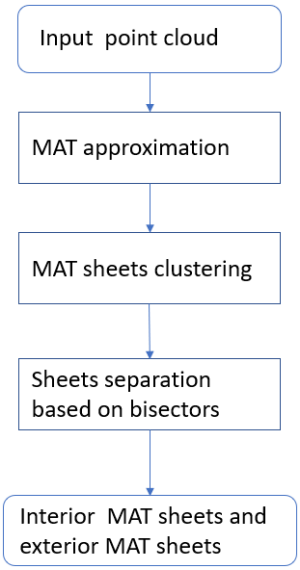


Figure 4.6: The workflow of MAT sheet determination based on bisectors

MAT clustering into MAT sheets

The MAT can be divided into different clusters based on the geometry. For example, Figure 4.7 illustrates that there is no intersection between interior and exterior MAT medial balls. The interior and exterior medial balls are located in different sides of the input point cloud. Therefore, it is feasible to divide the medial balls into interior and exterior MAT based on their relative positions. For example, in Figure 4.7 interior and exterior MAT locates in the different sides of the input point cloud.

Peters [2018b] proposed an approach to segment the MAT into MAT sheets based on the overlap between the medial balls (see Figure 4.8). The idea is to cluster the interior and exterior MAT into different MAT clusters. Peters [2018b] named the MAT clusters MAT sheets. Later in this thesis, the MAT clusters and MAT sheets are the same objects. Figure 4.8 shows the rule of how to determine if two medial balls belong to the same MAT sheets. d is the distance between the centres of the medial balls. r_1, r_2 are the radii of two medial balls respectively. s is the overlap ratio of two balls and it is a value from 0 to $+\infty$. 0 means that distance between the two balls is extremely large. $+\infty$ represents that the centres of the two balls are located in the same position. s can be set to different values to deal with different density point cloud. When the point cloud is sparse, s can be set to a small value. An example of MAT clustering is shown in Figure 4.9 where MAT is clustered into different sheets.

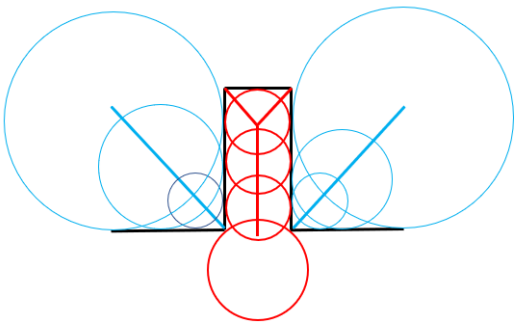


Figure 4.7: No intersection of interior and exterior MAT medial balls

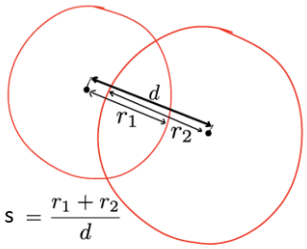


Figure 4.8: MAT medial balls overlap
[Peters, 2018b]

Interior and exterior MAT sheets determination based on the bisectors

Figure 4.10 illustrates what is the bisector of medial balls and its geometrical properties. o_1 is an interior medial ball of the input which touches the input at point p_1 and p_2. b_1 is unit length vector considered as the bisector of the medial ball o_1 and b_1 bisects the separation angle $\angle P_1 b_1 q_1$. The exterior medial ball o_2 has the similar structure where b_2 is the bisector vector. Normally the directions of the interior and exterior MAT are different. If only take the vertical direction into consideration, the

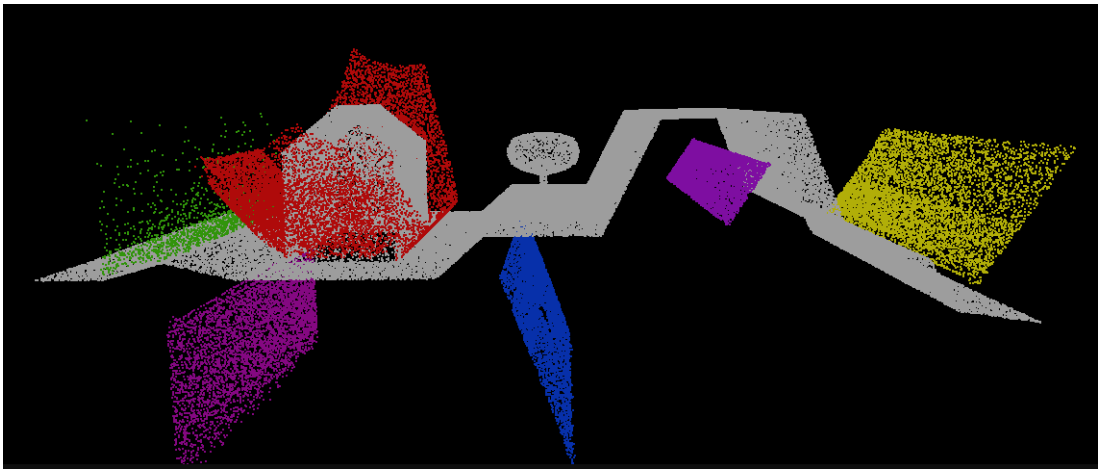


Figure 4.9: Point cloud with parts of MAT sheets, different sheets in different colours

interior bisectors point upward and the exterior bisectors point downward in most cases (see Figure 4.10).

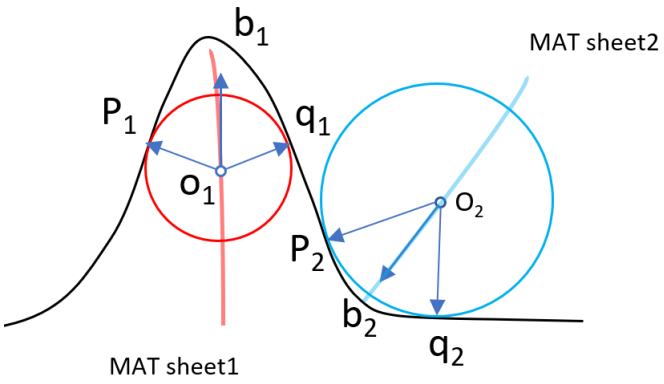


Figure 4.10: Bisectors of the interior and exterior MAT point to different directions

However, there are special situations when there is direction ambiguity of the bisectors (see Figure 4.11). In this case, the rectangle in black is the input point cloud of buildings (in black). The exterior medial balls and interior medial balls are in blue and red colours respectively. The separation angle of ball o_1 is 180 degrees. The bisector of medial ball o_1 has ambiguity due to the parallel surfaces, which means the bisector can either point upward or downward. The same bisector direction ambiguity exists in the exterior medial balls such as the ball o_2. Notice that this kind of direction ambiguity does not happen everywhere of the MAT sheet. It exists in the certain terrains of the point cloud such as buildings that consist of many parallel planes.

The direction ambiguity of bisectors can be solved by MAT clustering based method. This method contains three steps, ①zero value assignment, ②MAT sheet clustering and ③MAT sheet value assignment. The first step is to manually set the ambiguity bisector to zero vector which has no value and no impact on the vertical direction. The next step is the MAT sheet clustering which has been explained in section 4.2.2. The third step is to compute the z-direction (vertical direction) value of each MAT sheet. The idea is to calculate the average z-direction value of all the bisector vectors of one MAT sheet. And assign this value as the MAT sheet value. Since that the ambiguity bisectors have been set to zero vector, they do not influence the value of the corresponding MAT sheet. If the bisector points upward, its z-direction value is defined as positive, and vice versa. The interior MAT sheet has

positive MAT sheet value and the exterior sheet has negative MAT sheet value (see Figure 4.12 and Figure 4.13).

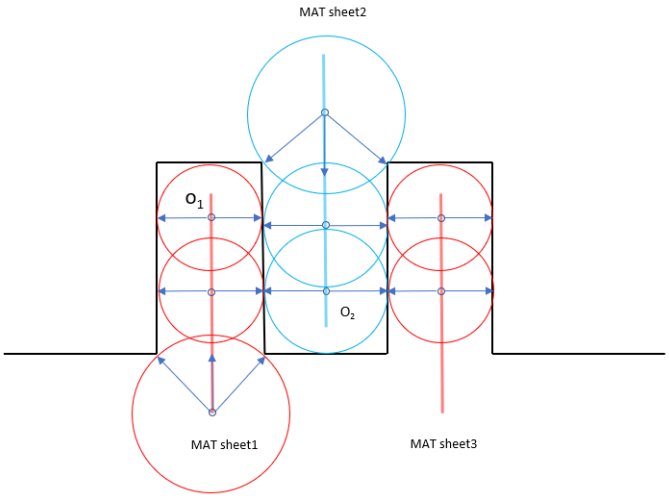


Figure 4.11: Special case of bisectors direction ambiguity

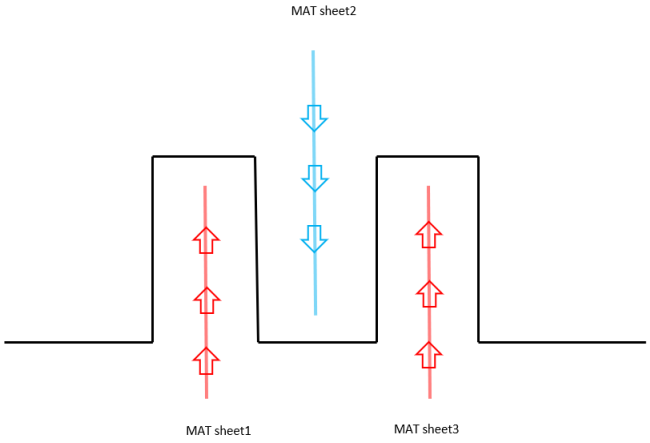


Figure 4.12: The z-direction MAT sheet value can be used to determine interior and exterior MAT sheet

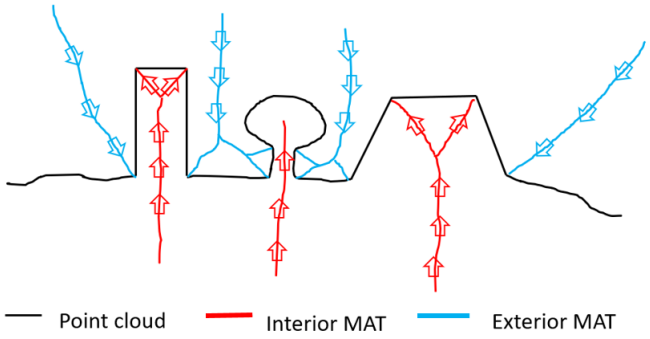


Figure 4.13: Different bisector directions of interior and exterior sheets

Figure 4.14 shows a case when the sheet value successfully determines the interior MAT and exterior MAT of a certain shape building. In the left figure of Figure 4.14, one sheet (in red) with a positive sheet value is classified as interior. However, this sheet is connect with an exterior sheet (in blue) and they are clustered into one bigger sheet at the end. The sheet value of the bigger sheet is still negative and it will be correctly classified as exterior MAT sheet.

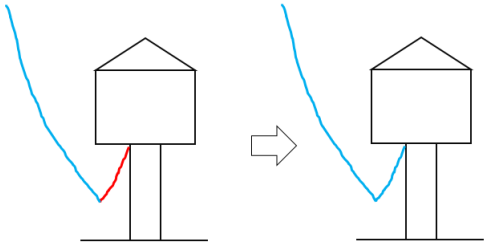


Figure 4.14: Successful determination of interior and exterior sheet by using sheet value

4.3 BRUTE FORCE SOLUTION OF VISIBILITY ANALYSIS

After the interior MAT determination, a brute force solution of visibility analysis can be proposed based on the 3D volume of the interior MAT. Because rays representing line of sight can be directly obstructed by the 3D volume. Figure 4.15 gives an example of how this approach works. In the Figure, A is the viewpoint, and B and C are points of the input point cloud. The 3D volume (in blue) consists of all the interior medial balls. Obviously, there is no occlusion between A and B, and the line of sight from A to C is obstructed by the 3D volume. Therefore, B is visible and C is not. By applying this method to all the points of the input, visibility analysis can be achieved.

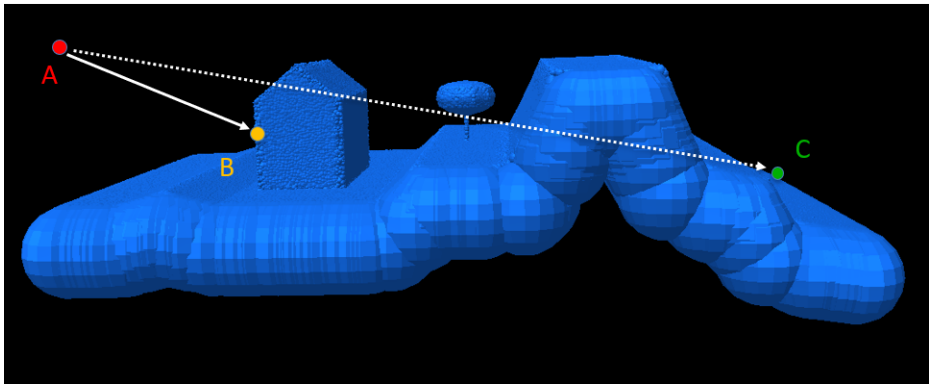


Figure 4.15: An example of the brute force solution of visibility analysis

The advantage of the brute force solution is that the accuracy of the result is guaranteed, since all the points of the input are checked. However, the limitation is the computation cost. All the points of the input have to be checked during the process. And for each point all the interior medial balls need to be checked. Apparently, the efficiency if this approach is not high. A new approach is proposed in the next section. It is worth mentioning is that the result of the brute force solution can be used in validation since it is always checking all the points and balls.

4.4 VISIBILITY QUERIES BASED ON KD-TREE

In this section, a KD-Tree based approach of visibility queries is explained. The idea is to create spatial index for all the interior medial balls by using a KD-Tree structure to speed up the visibility query. Instead of traversing all the points of the input, two patterns of rays, radial and parallel rays, are emitted from the viewpoint. Then, only the bounding boxes in the KD-Tree that intersect the rays need to be checked.

4.4.1 Spatial index generation of the interior medial balls by using a KD-Tree

The spatial index generation of the interior medial balls contains two steps, ① store the centre points in a KD-Tree and ② store the whole ball in the KD-Tree. The first step is straightforward. The centre of an interior medial ball is a 3D point. Therefore, a three-dimensional KD-Tree with bucket size 16 is created to save the centres, which means in each node maximum 16 centres are stored. Then second step takes the radius of the interior medial ball into consideration. Different nodes of the KD-Tree correspond to certain bounding boxes. The concept of this step is to save the interior medial ball to all the intersected bounding boxes of the KD-Tree.

Figure 4.16 gives an example of how the interior medial balls are saved in a KD-Tree. First the KD-Tree is created based on the centres of the interior medial balls. Then the balls intersected with multiple bounding boxes are copied to these bounding boxes. Fox example, ball C is only intersected with the bounding $box1$, so it is only saved in the bounding $box1$. Ball A is both intersected with the bounding $box2$ and $box3$, so ball A is duplicated once and saved both in the bounding $box2$ and $box3$.

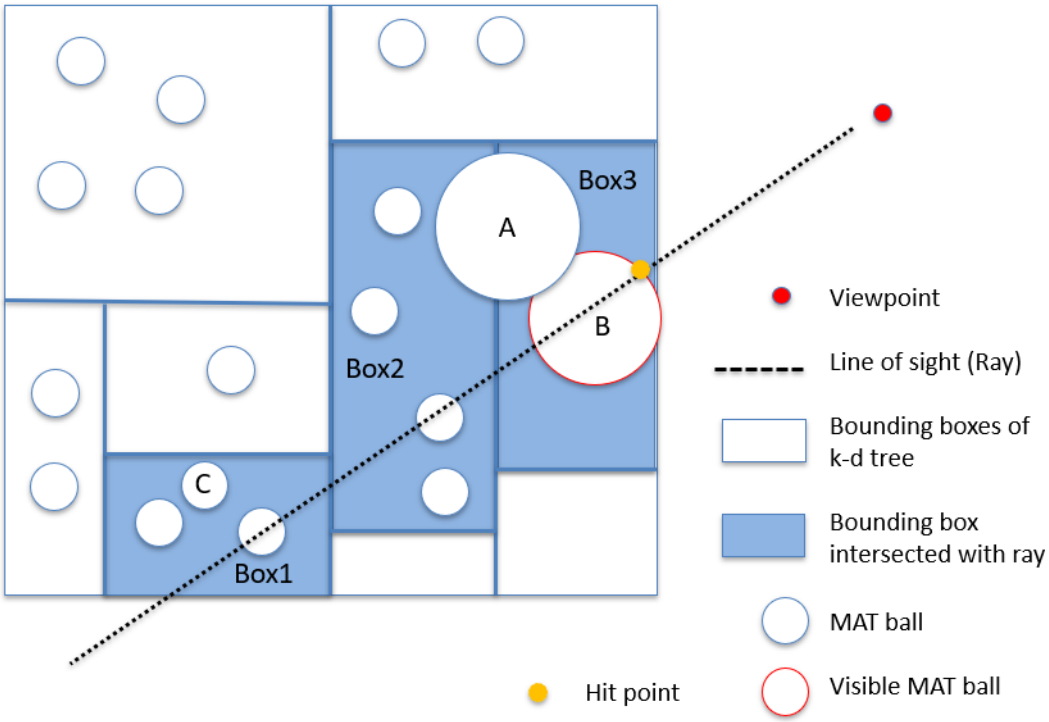


Figure 4.16: An example of how the interior medial balls are saved in a KD-Tree

A single ray query

Figure 4.16 also explains a single ray query process from a viewpoint. The goal of a ray query is to find the first hitting ball which is considered as the visible ball of that direction. A ray (dotted line) representing a line of sight is emitted from the viewpoint (in red). The ray has intersections with the bounding boxes ($Box1$, $Box2$, $Box3$) of the KD-Tree, which means only the medial balls in these bounding boxes have opportunity to intersect with the ray. Therefore, instead of checking all the medial balls of the KD-Tree, only the balls in the bounding boxes need to be checked. After checking all the balls in the bounding boxes ($Box1$, $Box2$, $Box3$), the nearest intersected ball ($BallB$) is marked as the only visible ball of the ray direction.

The Algorithm 4.2 describes the process of a single ray query. This algorithm first finds out the bounding boxes which have intersections with the ray. Then check all the medial balls in these bounding boxes and calculate the distance between the

ball and the viewpoint. Finally, find out the nearest ball to the viewpoint and set it as the visibility result of the ray.

Algorithm 4.2: Single ray process

```
Input: viewpoint A, the KD-Tree and a ray vector v_i from the viewpoint
for each bounding box Box_i in the KD-Tree do
    initial a intersection flag, bool ifinter
    if Box_i intersects with the ray vector v_i then
        ifinter = 1;
    else
        ifinter = 0;
initial a nearest medial ball container Ball_nearest
for All the balls stored in bounding boxes with ifineter = 1 do
    check the distance between ball and viewpoint A
    mark the ball with the shortest distance and updata the Ball_nearest
Output: the Ball_nearest considered as the visibility result of the single ray
```

Figure 4.17 illustrates why the medial balls need to be copied to all the intersected bounding boxes. In the figure, the ray representing the line of sight has intersections with the bounding boxes, *Box*1, *Box*2 and *Box*3. And it does not intersect *Box*4. *CircleD* is a medial ball with a large radius and it is firstly saved in the bounding box *Box*4, due to its centre's position. *CircleD* is intersected with *Box*2, *Box*3 and *Box*4. Since *CircleD* is not copied into *Box*2, *Box*3 and *Box*4, the ray have no intersection with *Box*4. *CircleD* will be skipped in this ray query process. Therefore, the visible ball of the ray is *ballB* (see Figure 4.16). Apparently, this result is wrong and *CircleD* is the visible ball of the ray query. This error can be solved if *CircleD* is saved in all the intersected bounding boxes. Then *CircleD* is saved in *Box*2, *Box*3 and *Box*4, and it will be checked during the ray query.

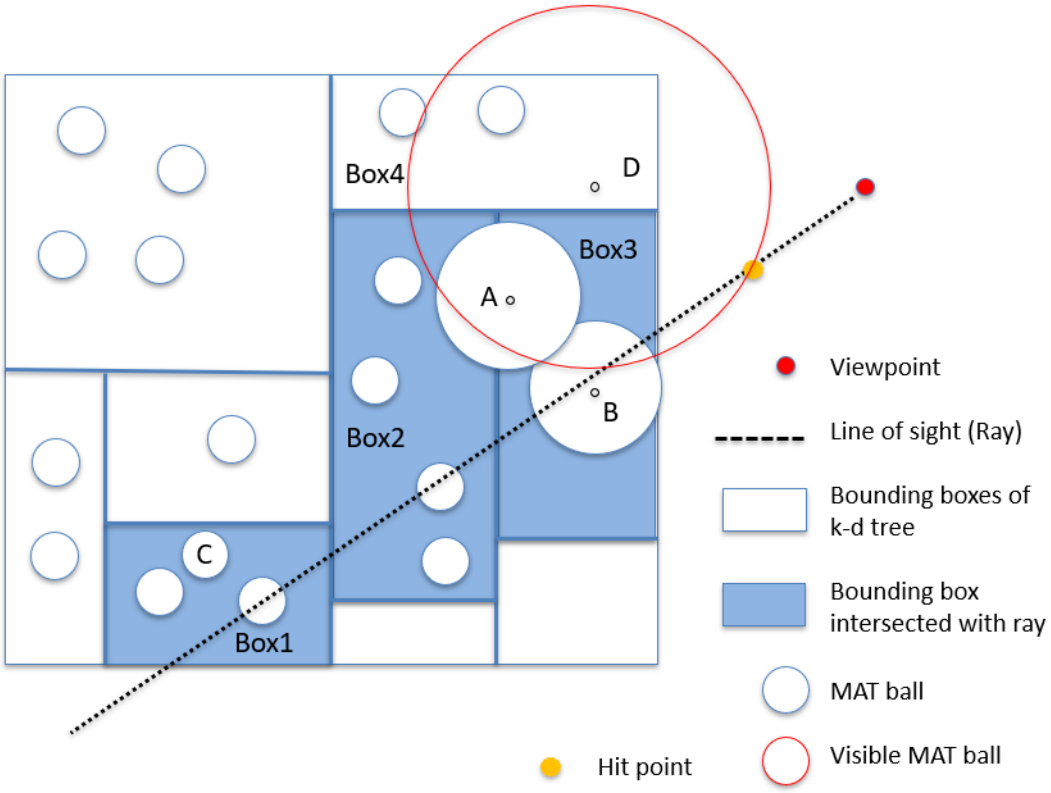


Figure 4.17: An example of how the interior medial balls are saved in a KD-Tree

4.5 VISIBILITY QUERIES BASED ON R-TREE

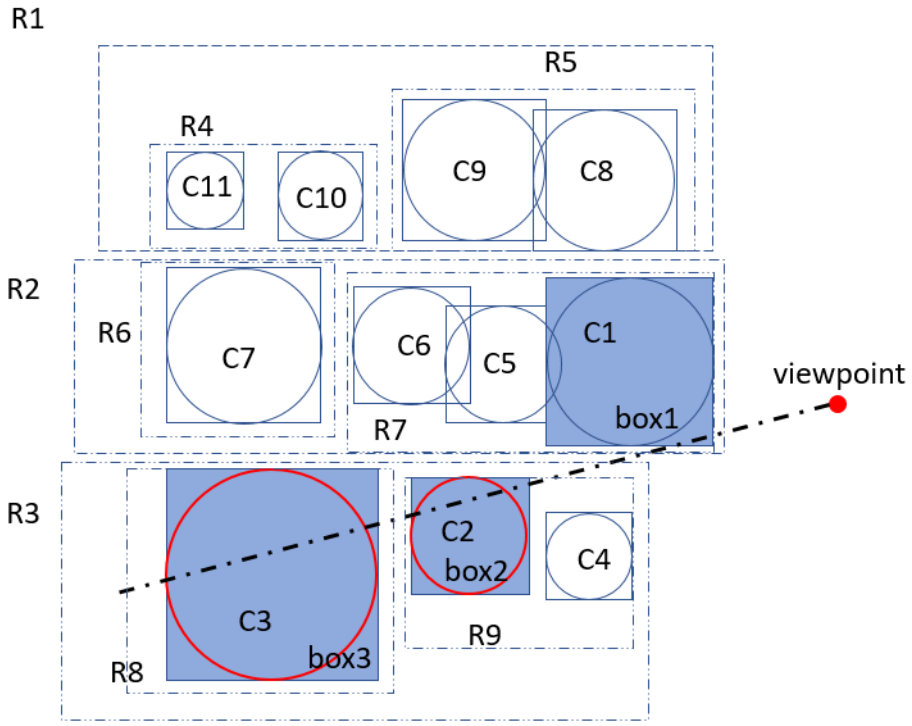


Figure 4.18: An example of R-tree structure of medial balls

Figure 4.18 shows how the medial balls are stored in an R-Tree. The input data are medial balls (C1 to C11) and a viewpoint. Each ball associates with the smallest bounding box. $R1$ to $R9$ are the non-leaf nodes in the R-Tree and they have a corresponding bounding box.The corresponding graph representation of the R-Tree is shown in Figure 4.19.

The single ray query based on R-Tree is similar to the one based on KD-Tree. The first step to get all the intersected bounding boxes ($box1$, $box2$, $box3$). The next step is to check the corresponding medial balls and find out all the medial balls intersected with the ray. They are $C2$ and $C3$. It is worth mentioning that although $box1$ is intersected with the ray, $C1$ has no intersection with the ray. The final step is to select the nearest ball to the viewpoint from all the balls intersected with the ray. Therefore, the result of this single ray query is $C2$.

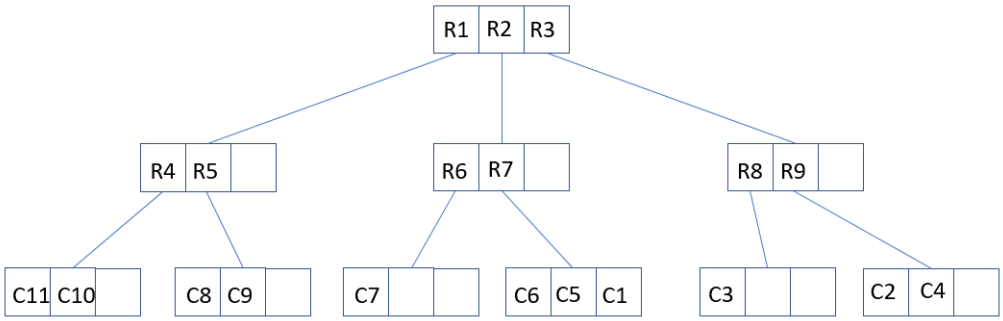


Figure 4.19: Graph representation of the RTree in Figure 4.18

4.6 VISIBILITY QUERY BASED ON TWO RAY PATTERNS

4.6.1 Visibility query based on radial rays

The visibility query based on radial rays consists of three steps: ray emission, ray filtering and ray intersection. Figure 4.20 gives an example of visibility query based on radial rays. First radial rays are emitted from the viewpoint (in red) to all directions. In 3D space, these rays will form a sphere. The next step is ray filtering. Notice the fact that some rays may not hit the point cloud and only parts of the rays intersect with the input point cloud. The idea of this step is to figure out which rays can have an intersection of the input point cloud. This step can be done by checking if the ray intersects with the bounding boxes of the input point cloud. This is a straightforward ray-box intersection process. As mentioned in section 3.1, the ray-box intersection can be done efficiently by using Williams et al.'s algorithm. The final is step is based on the single ray query mentioned in section 4.4.1. For each single ray query, a visible ball will be found in the end. Each medial ball corresponds to a point of the input point cloud. Therefore, after gathering all visible balls the visible points of the input point cloud can be acquired.

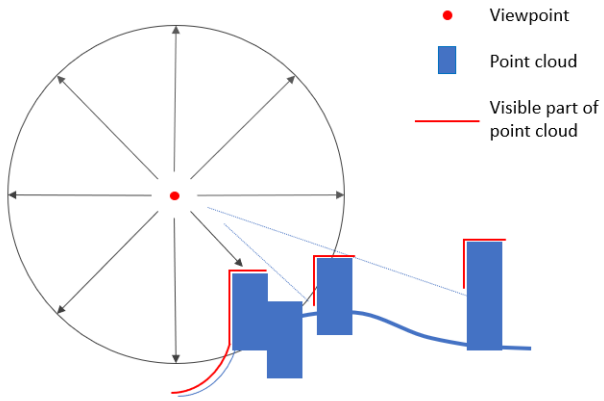


Figure 4.20: Visibility query based on radial rays

4.6.2 Visibility query based on parallel rays

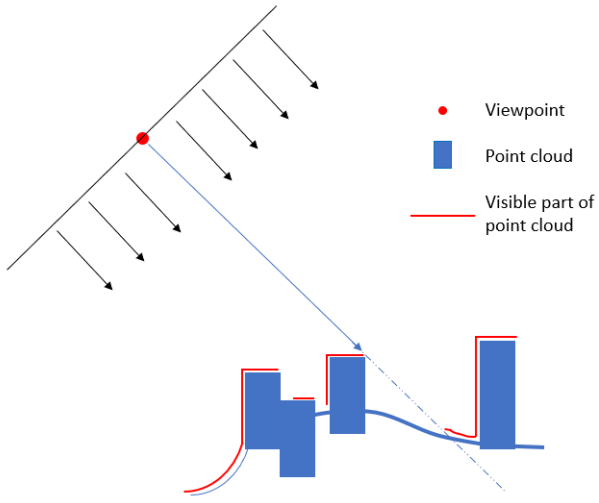


Figure 4.21: Visibility query based on parallel rays

The visibility query based on radial rays consists of three steps, ray emission, ray filtering and ray intersection. Figure 4.20 gives an example of a visibility query based on parallel rays. The first step is to create parallel rays. The idea is to generate a squared plane which is perpendicular to the line of sight. From that plane rays are generated parallel to the line of sight. Comparing to the radial rays visibility analysis, ray emission is the only different step. The ray filtering and ray intersection steps are the same as the ones in radial rays visibility analysis.

4.7 STRATEGIES FOR SPEEDING UP THE VISIBILITY QUERY

In order to speed up the visibility query process, several strategies are considered in this thesis. This section explains two strategies to improve the query efficiency. One is based on the medial ball simplification. The idea of it is to reduce the number of the medial balls without significantly sacrificing result quality. The second method is to use the GPU for parallel computing. The whole visibility query process consists of a number of single ray queries. These single ray queries are independent, which makes the whole process suitable for parallel computing.

4.7.1 Medial ball simplification

Figure 4.22 illustrates the process of medial ball simplification. On the left there are the medial balls and their corresponding points. The simplified result is shown on the right. The simplification is based on the overlap level between medial balls (see Figure 4.8). A threshold s can be set to determine if the balls should be simplified or not. The threshold can be calculated by the formula, $s = \frac{r_1+r_2}{d}$, where s describes how close these balls are. For example if one medial ball A is very close to medial ball B and the overlap level is larger than the threshold, these balls will be simplified into one ball. A medial ball is linked to a unique point of the input point cloud. Therefore, the simplified ball corresponds to all the points linked to the balls before simplification.

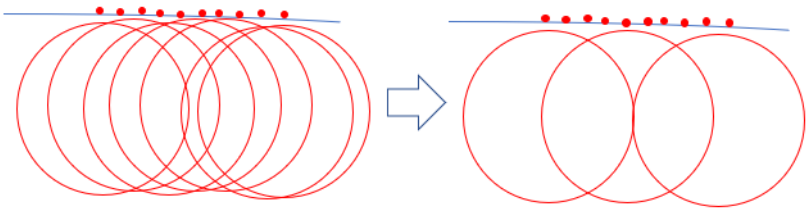


Figure 4.22: Visibility query based on parallel rays

4.7.2 Speed up by using GPU for parallel computing

Since every single ray query has the same type of query process, it is quite suitable to speed up the whole process by parallel computing. Recently GPU has been widely used in parallel computing. There are developed libraries of GPU computing, for example CUDA, OpenGL and AMP. This thesis uses AMP library to implement parallel calculation. (see Section 5.2.9) Ideally, each single ray query process is able to run on a computation unit of GPU. After that, all the visible results are merged in the CPU to get the final result. For multiple viewpoints, the process of each viewpoint can be considered one independent unit where parallel computing can be applied.

5 IMPLEMENTATION AND EXPERIMENTS

This chapter explains the implementation of the MAT based approach and its experiments. Section 5.1 describes the datasets tested and the developing tools. Section 5.2 explains how the implementation is done from a programming perspective.

5.1 DATASETS AND TOOLS

5.1.1 Datasets

In this thesis several point cloud datasets are used for experiments. They are AHN3 [AHN3, 2019], EMC point cloud of Rotterdam 2016, Dense point cloud of Bergamo and an artificial point cloud. The artificial point cloud is generated from meshes by software Meshlab. It has no normal errors. And it is used to develop all algorithms needed. The AHN3 and EMC Rotterdam point clouds are used to test the robustness and accuracy of the algorithms. The AHN3 point cloud is mainly used to test the anti-noises ability of the algorithms, due to that there are noises and surfacing missing. EMC Rotterdam 2016 is a dense point cloud and it is mainly used for MAT separation test. Visibility analysis experiments are tested on all the datasets mentioned to check the performance of the approaches.

Dataset name	Source	Density (pt/m^2)	Acquition technique	Usage
Artificial Point cloud	Meshlab	80-100	Generate from meshes	Algorithm Test
AHN3 Delft	Rijkswaterstaat	10-20	Laser scanning	Visibility query test
AHN3 Rotterdam	Rijkswaterstaat	10-20	Laser scanning	Visibility query test
EMC_Rotterdam_2016	Municipality Rotterdam	30-100	Laser scanning	MAT separation and result check
Dense point cloud of Bergamo	Private company AVT	80-150	Dense image matching	Result check

5.1.2 Developing tools and environment

Hardware

All the algorithm in this thesis are developed in a laptop with a 64-bit Window10 operating system. The detail of the laptop is stated as following:

Name and brand	Lenovo ThinkPad P1
Processor	Intel(R) Xeon(R) E-2176m CPU @ 2.70 GHz
RAM	16 GB
GPU	NVIDA Quadro P2000

Software

All the visibility analysis processes are done by a self-developed open source software. This software is based on a software framework, geoflow [Peters, 2018a]. It is developed by C++ programming language. It contains several C++ libraries and packages. The detail of the libraries and packages is stated as following:

- ImGui[1] is C++ developed library with bloat-free graphical user interface. It is used to created the user interface of the software.
- LAStools[2] is library which handle multiple point cloud processes, such as loading and saving point cloud, segmentation, point cloud format conversion and PCA normal approxmating.
- PCL[3] is another powerful library to deal with point cloud processes.
- Easy3D[Nan, 2018][4] is C++ develped library to handle 3D shapes. In this thesis, this library contributes to normal approximation and normal reorientations.
- KD-Tree library[5] is used to generate the KD-Tree data structure of a point cloud.
- Eigen[6] is used for normal and covariance matrix calculation.
- Boost[7] is used for geometry-related computing such as normal approximating and MAT approximating. The R-Tree library in Boost is used for R-Tree implementation.
- CGAL[8] provides multiple geometric algorithms for many fields, such as GIS and computer vision.
- AMP [9] is used for parallel computing to accelerate the query speed.

1 https://github.com/ocornut/imgui
2 https://github.com/LAStools/LAStools
3 https://github.com/PointCloudLibrary/pcl
4 https://github.com/LiangliangNan/Easy3D
5 https://cgl.ethz.ch/pointshop3d//sourcedoc/html2.0/classKdTree.html
6 http://eigen.tuxfamily.org/index.php?$title = Main_{p}age$
7 https://www.boost.org/
8 https://www.cgal.org/
9 https://docs.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-cpp-accelerated-massive-parallelism?view=vs-2019

5.2 IMPLEMENTATION

5.2.1 Framework and user interface

The software developed consists of four main parts, main toolbar (Figure 5.1 1), operation workspace (Figure 5.1 2), 3D viewer (Figure 5.1 2) and visualization settings (Figure 5.1 4). The main toolbar is used to load, save and close flowcharts which are JSON files and include the required information of each operation. Operation workspace is where operation nodes are drawn. In the operation workspace, operation nodes representing certain processes can be added, removed, set and linked. 3D viewer is used to visualize the point cloud and processed results. Visualization setting is used to change the visualization style.

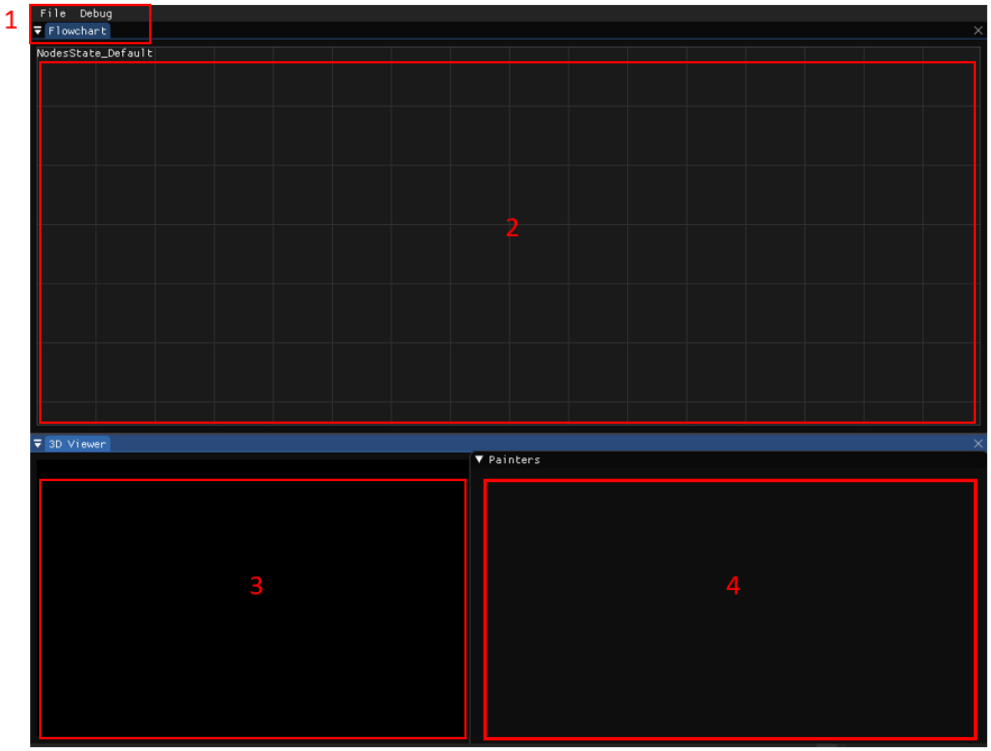


Figure 5.1: An overview of the software

5.2.2 Nodes, operations and connections

The point cloud operations are represented by nodes (see Figure 5.2). One node corresponds to one point cloud process. For example, the node *LASLoader(1)* represents the point cloud loading operation based on LAStools. Normally a node contains name, input interfaces and output interfaces, such as the node *ComputeNormalNode(3)* in Figure 5.2. The input interface is on the left side and outputs locates on the right side. The start and the end node only have a side of interfaces, such as the node *LASLoader(1)* and the node *Painter(4)*. The nodes can be connected by lines through which data is transferred. When all the required input has been transferred to a node, this node will run automatically. When the process is done, the node will turn to green colour. Otherwise, the node will stay in yellow colour waiting for more inputs. When right click a node, a dialogue where parameters can be set will pop up (see Figure 5.2).

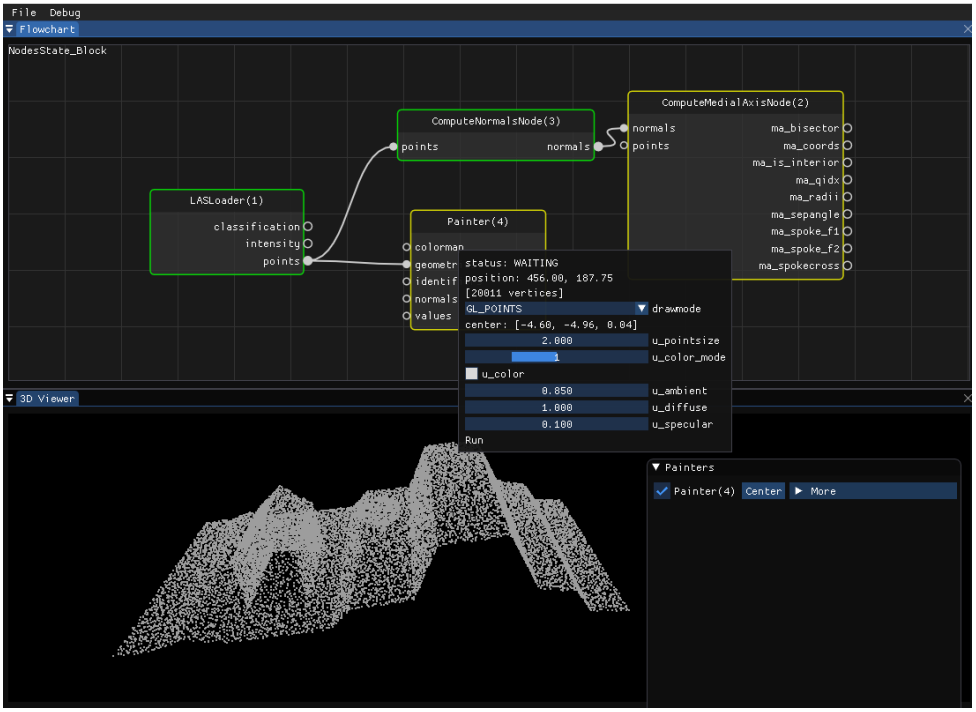


Figure 5.2: Point cloud operations represented by nodes

5.2.3 MAT approximation

As mentioned in section 4.1, the MAT approximation is done based on the adapted version of the shrinking ball algorithm. The detail and source code of algorithm is stated in Appendix I. This operation is done by a node *ComputeMedialAxisNode* (Figure 5.3). The input is the points of point cloud and the normals of points. MAT points can be calculated by this operation. Additional information such as bisectors, medial ball radii and medial ball indices can be computed as well. There are three parameters for this node which corresponds to the concept of denoise, planar detection and initial radius explained in Section 4.1.

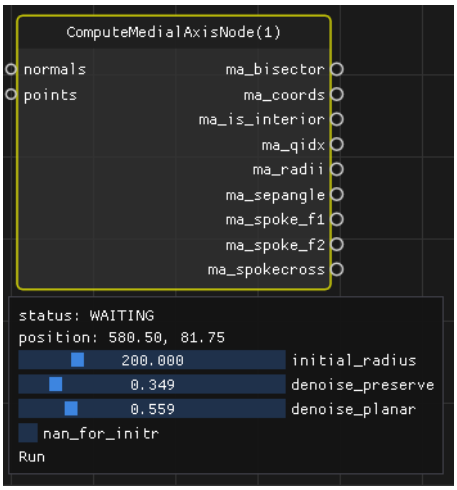


Figure 5.3: MAT approximation node

5.2.4 MAT separation based on normal reorientation

Figure 5.4 shows the steps of how to implement the normal reorientation approach. Since the artificial point cloud has one hundred percentage correct normals, the interior and exterior MAT can be separated perfectly based on the adapted shrinking ball algorithm (see section 4.1). In order to simulate the situation in practise, normals of the point cloud are manually set to wrong directions. After adding the noise to the point cloud (randomly change the normal directions of 10,000 points), the adapted shrinking ball algorithm is not able to generate correct interior MAT. Then the normal reorientation step will fix the wrong normals. This step is based on the Easy3D library. The source code is stated in Appendix I.

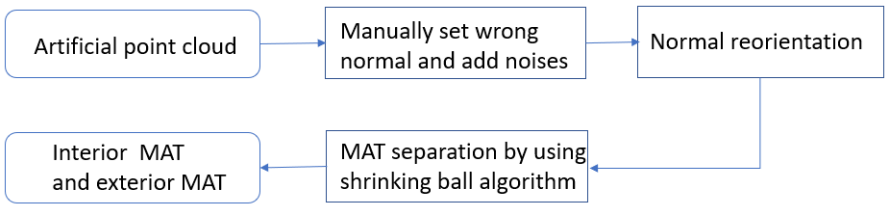


Figure 5.4: Steps of normal reorientation process

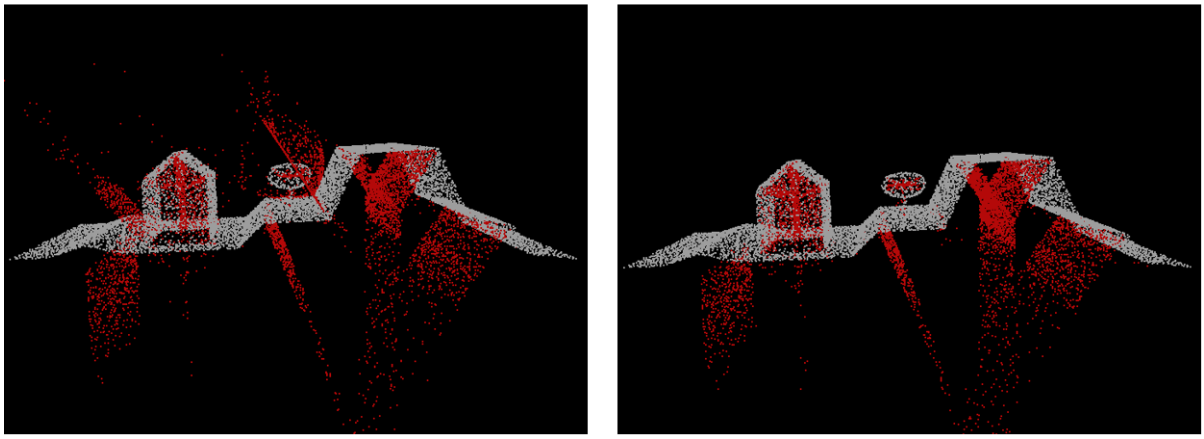


Figure 5.5: MAT balls with different initial radii (left before normal reorientation, right after the normal reorientation)

Figure 5.5 shows the difference of interior MAT between normal reorientation results and without normal reorientation. In the left figure, due to the noises and wrong normals added manually, interior sheets are wrongly classified. Several exterior sheets are regarded as interior ones near the building and tree. After fixing the direction of normals, the shrinking ball algorithm successfully separate interior MAT sheets and exterior ones (see Figure 5.5 right).

However, the input point cloud of this experiment is an artificial one, which cause restrictions. Although noises and wrong normals have been added into the point cloud, there are difference between this point cloud and point cloud in practice such as AHN3. More details and discussions of the performance and limitations of this normal reorientation process will be involved in Chapter 7.

5.2.5 MAT separation based on bisectors

Figure 5.6 shows the nodes of MAT separation based on bisector. *Node*1 is the MAT approximating operation (see section 5.2.3) and it calculates all the MAT points both interior and exterior. *Node*2 is for sheet clustering. As mentioned in section

4.2.2, this process uses a ball overlap value to determine whether balls belong to the same class. And this process generate a classification id to mark all the MAT points($segment_ids$ in Figure 5.6). In *node*3, classification id is used to merge MAT point into different MAT sheets. Finally, the bisector value of each sheet is computed in *node*4. The sheets with negative bisector value are classified into exterior MAT sheets and the sheets with positive values are classified into interior MAT sheets. The source code of this process is states in Appendix I.

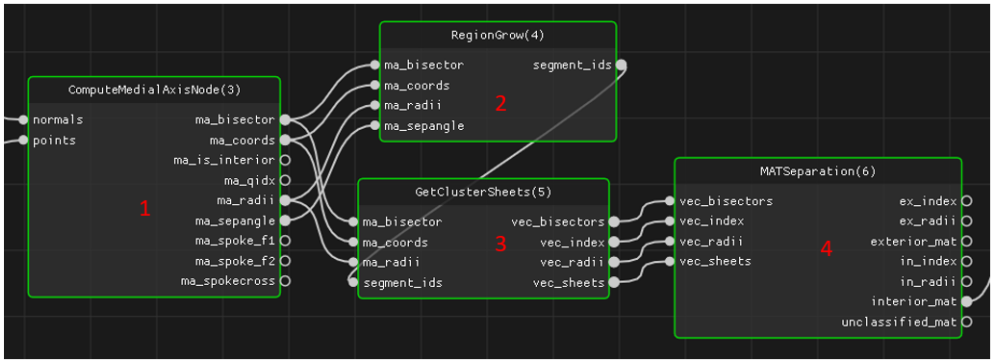


Figure 5.6: The nodes of MAT separation based on bisectors

5.2.6 Implementation of KD-Tree and R-Tree

The KD-Tree is implemented by a C++ library developed by Keiser [1993]. It has interfaces for basic KD-Tree queries such as finding the nearest node, finding n the nearest neighbours, return the nearest node to a certain segment, etc. To construct a KD-Tree, a constructed function *KdTree()* can be called.

The R-tree is implemented by the BOOST library. Algorithm 5.1 shows the process of constructing an R-Tree and a query of a segment intersecting with the R-Tree. This BOOST library also contains different queries based on R-tree, such as finding the nearest node of an input, rectangle intersects with R-Tree, finding n the nearest neighbours, etc.

Algorithm 5.1: R-Tree implementation

```
1 Input: all the medial balls
2 Define a R-Tree. boost::geometry::index::rtree
3 for each medial ball do
4     calculated it smallest bounding box and insect the box into the R-Tree.
5 Define a query object, such as a segment, bg::model::segment<point> seg
    (point A, point B)
6 Call the query function already implemented in BOOST.
    rtree.query(bgi::intersects(seg), std::backinserter(result));
7 Output: result which contains all the bounding boxes intersected with the
    segment.
```

5.2.7 Radial rays

Figure 5.7 shows the implementation of radial rays. The input is a viewpoint (in red). Radial vectors is created in the *nodeRadialRaysGenerator* and the start of the vectors is the viewpoint. The idea is to generate a group of spherical points which are the ends of the radial vectors. There are two parameters of the *nodeRadialRaysGenerator*, radius and density which control the size of the sphere and the interval between radial vectors. The source code of the *nodeRadialRaysGenerator* is in Appendix I.

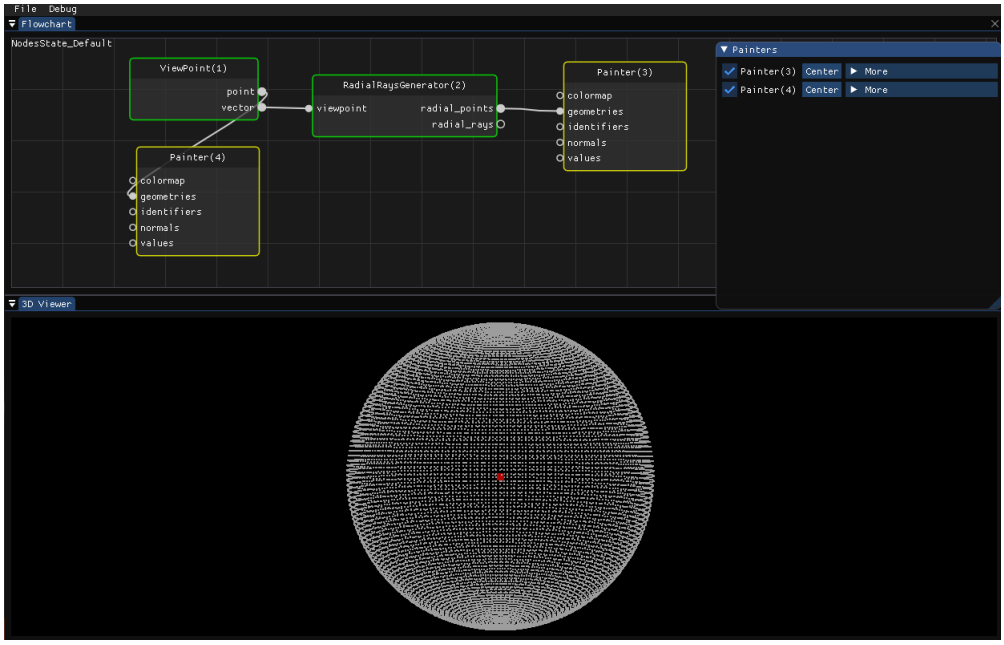


Figure 5.7: The implementation of radial rays

5.2.8 Parallel rays

Following the principles of parallel rays generation (section 4.6.2), the implementation is straightforward. Figure 5.8 illustrates the steps of parallel rays implementation. The first step is to input a vector that consists of a start point and an end point. The second step is to create a perpendicular plane at the start point. This can be calculated by applying the following mathematical principles:

- Input viewpoint P_{start} (a, b, c),
  Input vector v_1 (A, B, C) which is also the normal of the plane,
- Point Q(x, y, z) in the plane meets the condition of the following formula,
  $\langle A, B, C\rangle \cdot \langle x-a, y-b, z-c\rangle = 0$
- In other word,
  $A(x-a)+B(y-b)+C(z-c)=0$

This step is done in the *nodeParallelVector* (see Figure 5.9). The output are two vectors, *Headvectors* and *Endvectors*. The start points of the vectors are saved in *Headvectors*, and the corresponding end points are stored in *Endvectors*.

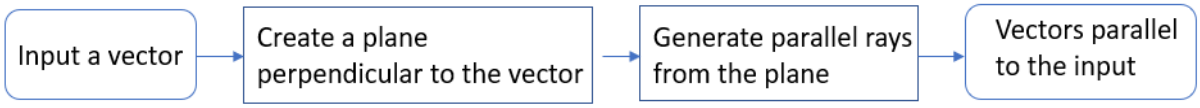


Figure 5.8: The steps of parallel ray implementation

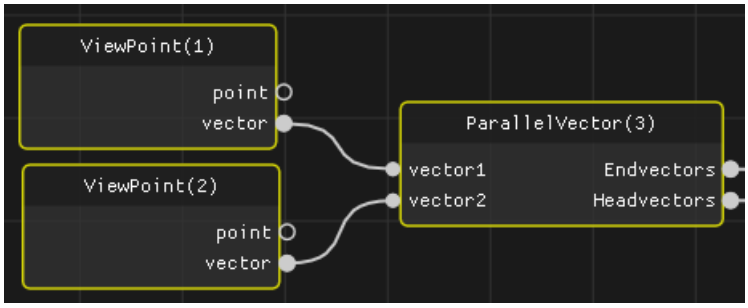


Figure 5.9: The node implementation of parallel rays

5.2.9 Parallel computing by GPU and multiple threads

In this thesis the parallel computing is implemented based on C++ Accelerated Massive Parallelism (AMP) [Microsoft_AMP_Developing_Group, 2014]. Since I use a 4-core CPU, all the rays need to be check is equally divided into 4 groups. Then I created 4 threads in the process. In each thread, rays are parallelly check in the GPU. At the end, the result of each ray is merged into an entire c++ vector. The source code of this process is described in Appendix A.1.

6 RESULTS AND DISCUSSIONS

This chapter describes the visibility results and the validation analysis of this research. The visibility results of the normal reorientation approach and bisector based approach are discussed. Four different datasets are used to test these approaches. The validation part is done by making a comparison with the results of an open-source software, CloudCompare. The efficiency results of approaches are also discussed in this chapter.

6.1 VISIBILITY RESULTS OF DATASETS

6.1.1 Normal reorientation approach

Artificial point cloud

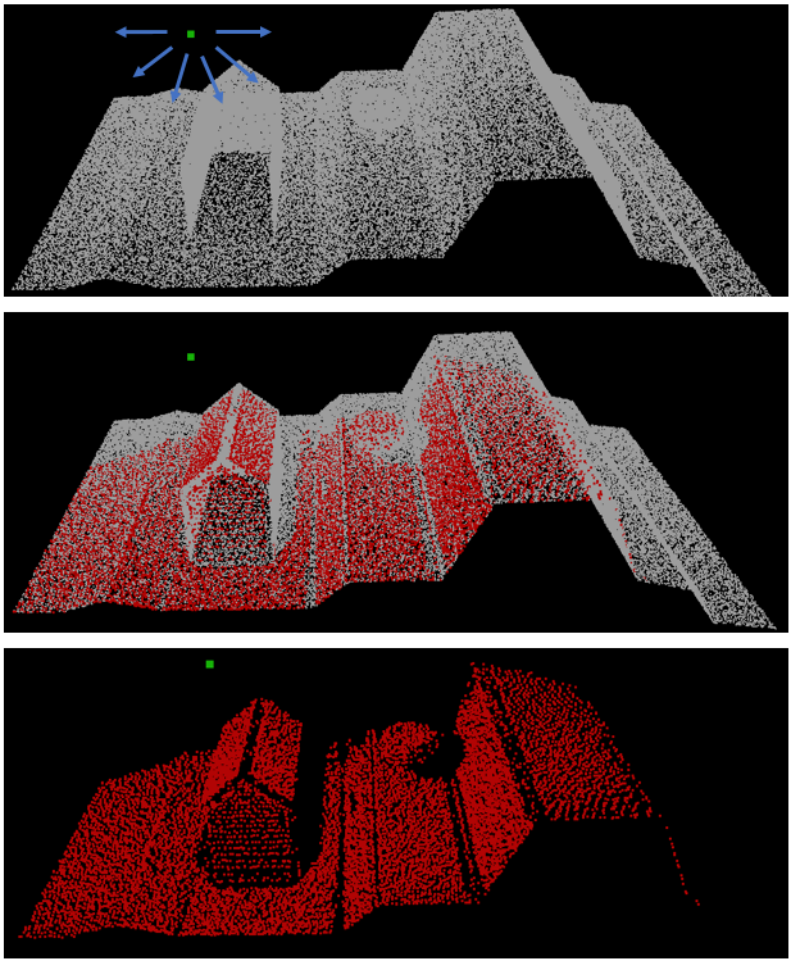


Figure 6.1: The visibility result of the artificial point cloud (radial rays)

As mentioned in Section 5.2.4, wrong normals are manually added to the artificial point cloud. For this point cloud, the normal reorientation approach is able to

acquire decent visibility results. Figure 6.1 shows the result based on radial rays. The green dot is the viewpoint. And the white point cloud is the input point cloud with 100,052 points. There are 6395 visible points (in red). The top picture only shows the viewpoint and input point cloud. The medium one shows the situation when visible point cloud is added. The bottom one only illustrates the viewpoint and visible points.

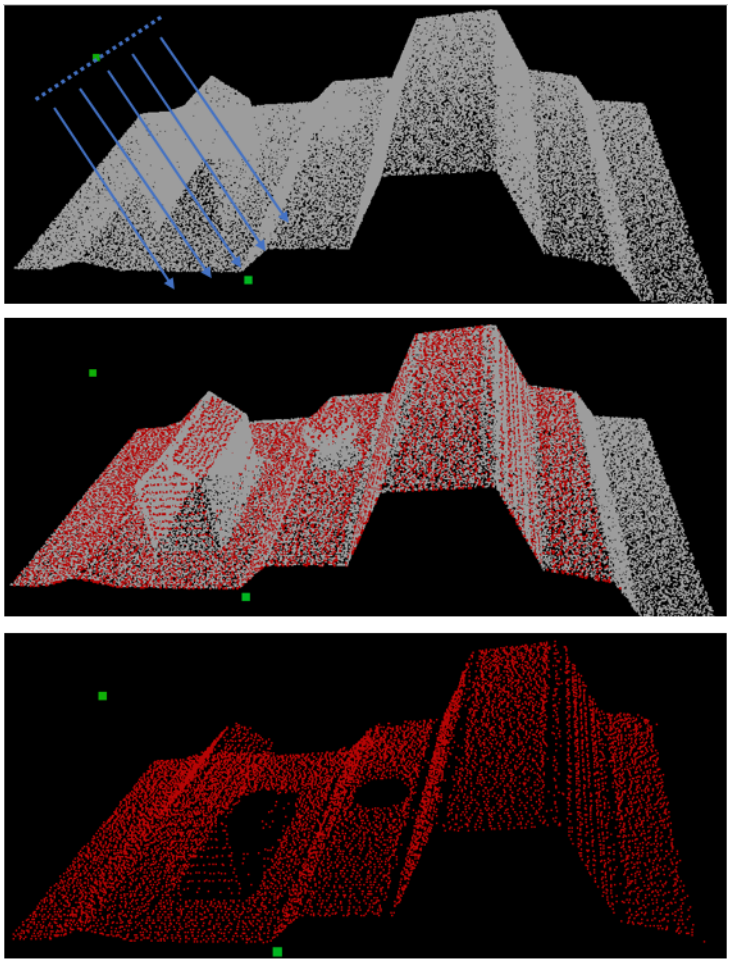


Figure 6.2: The visibility result of the artificial point cloud (parallel rays)

Figure 6.2 shows the visibility result of parallel rays. The ray direction is defined by a vector which is made by a start point and an end point (in green). The top picture only shows the viewpoint and input point cloud. The middle one shows the situation when visible point cloud is added. The bottom one only illustrates the viewpoint and visible points. 12581 points of the input is visible (in red).

AHN3 point cloud, a building in Rotterdam (Building A)

For the AHN3 point cloud, the normal reorientation approach is not able to deliver the correct result, because it cannot determine which normal direction is correct and it only can flip the normal by 180 degrees. Taking a building as an example, this approach is able to make sure the normals within each single facade are consistent. However, the normals of two facades can point to the opposite direction.

Figure 6.4 gives an example of the situation when the normal reorientation failed to determine the interior MAT. The input is an AHN3 point cloud of a building in Rotterdam (in Figure 6.3, name it *BuildingA*). The interior MAT is in blue colour. The origin interior MAT results are on the left side (Figure 6.4 *a* and *b*) and the normal reoriented results (Figure 6.4 *c* and *d*) are on the right side. The origin

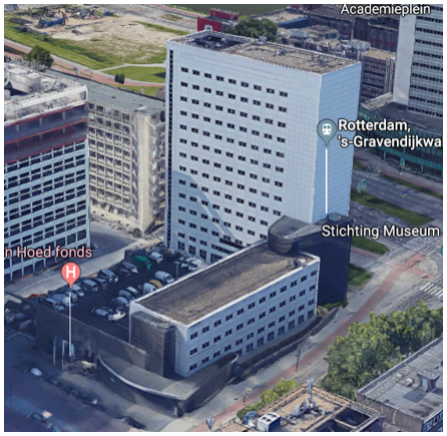


Figure 6.3: Building A shown in Google Maps
[Google, 2019]

interior MAT is calculated by the adapted shrinking ball algorithm with normals computed by the PCA method. Due to the noises of the AHN3 point cloud and PCA limitations at sharp edges, part of the exterior MAT is assigned as interior. For example in *a* and *b*, on the left of the building a group of "interior" MAT is located outside the building.

Figure 6.4 *c* and *d* show the interior MAT after normal reorientation. However, this interior MAT shown in Figure 6.4 *c* and *d* is originally classified as exterior MAT. The normal reorientation approach is able to separate the MAT into two classes, but it cannot determine which one is interior. Because this method makes normals point at the same direction, but it does not know if the pointing direction is correct or opposite. After one more step of manual determination, interior MAT result in Figure 6.4 *c* and *d* is obtained. Even though after normal reorientation, the interior MAT is still not all correct, especially at sharp edges shown in the red circle.

Figure 6.5 shows the interior medial balls of the original PCA normal and after normal reorientation. Obviously, the original version contains a large number of wrong classified medial ball. After normal reorientation, the number of wrong medial ball reduces but they still exist. And there are more errors on the left side of the building.

Figure 6.6 shows the visibility result of the normal reorientation approach. In order to ensure the quality of the output, the brute force approach is used. *a* and *b* are the result of the same viewpoint on the left of the building. *c* and *d* are the result of the same viewpoint on the right of the building. As mentioned above, there are more errors of the interior medial ball on the left part of the building. Therefore, the performance of the left viewpoint is poorer than the right viewpoint's. In general, the visibility result of the normal reorientation approach is far from satisfied.

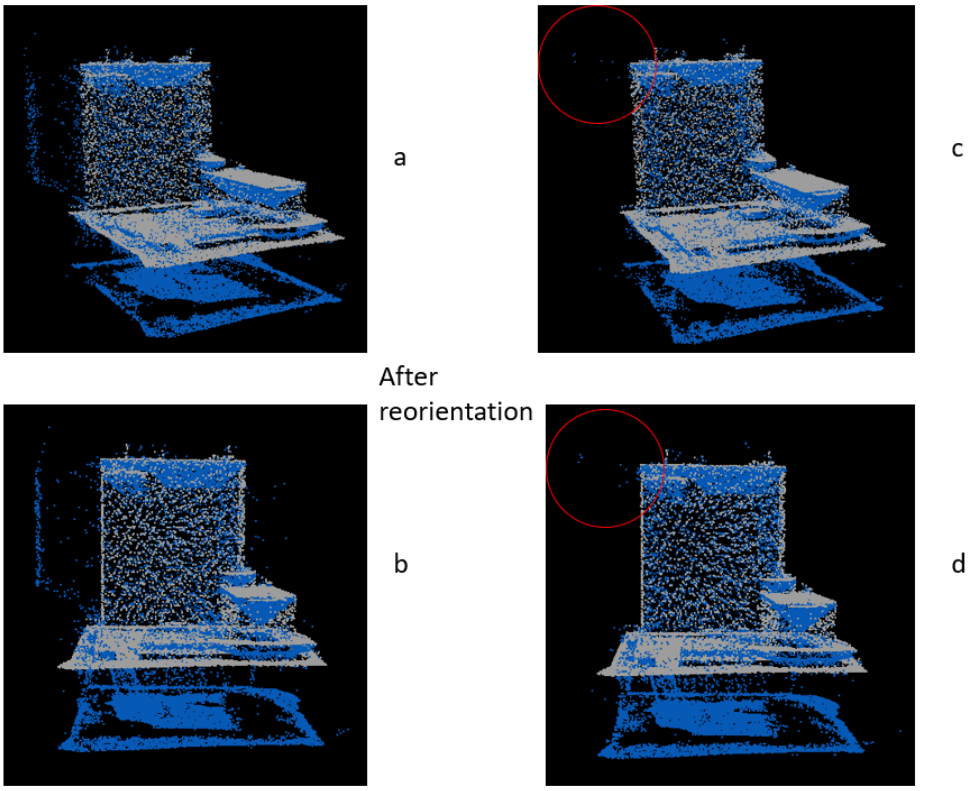


Figure 6.4: Normal orientation failed when dealing with an AHN3 building, interior MAT in blue, input point cloud in white, initial radius of shrinking ball is 10 meters

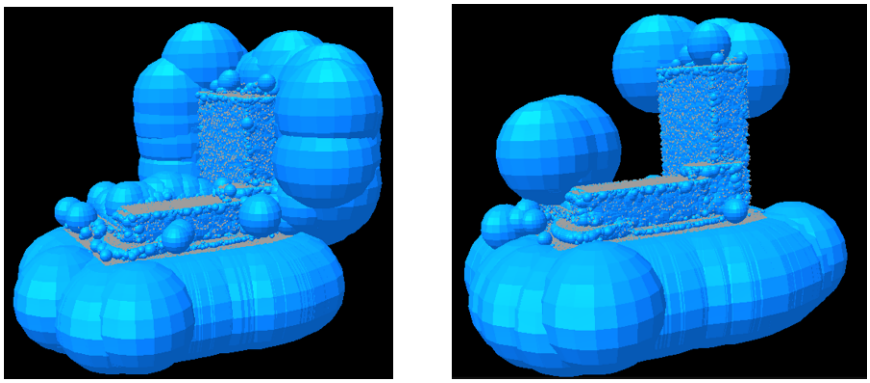


Figure 6.5: Interior medial MAT balls, left original based on PCA normals, right after normal reorientation

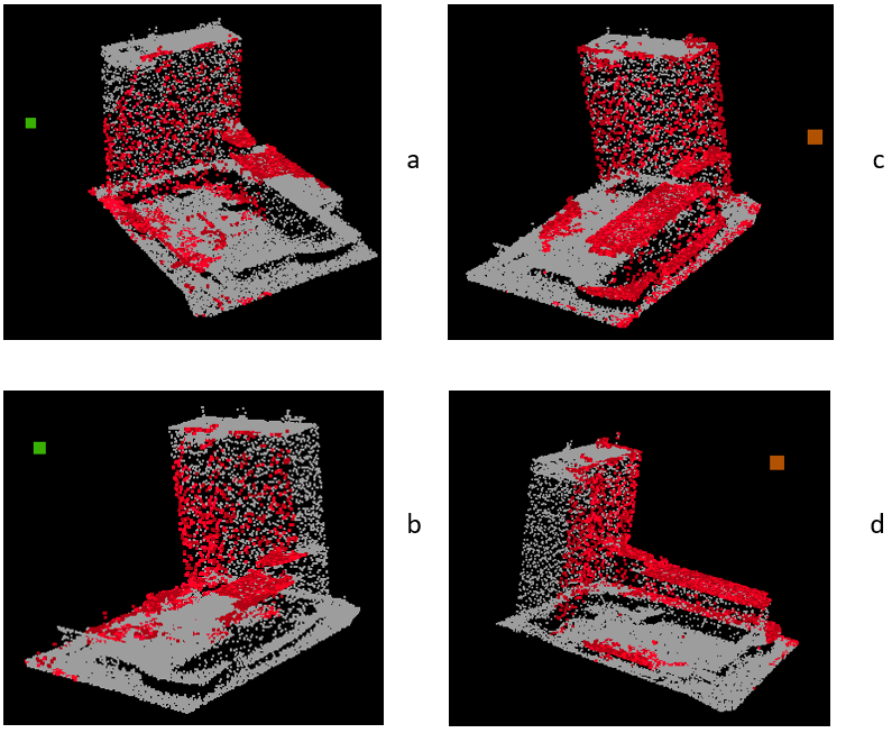


Figure 6.6: Visibility result of normal reorientation approach (Brute force ray checking, input PC in white, visible points in red, left viewpoint in green, right viewpoint in brown)

6.1.2 Bisector based approach

Artificial point cloud

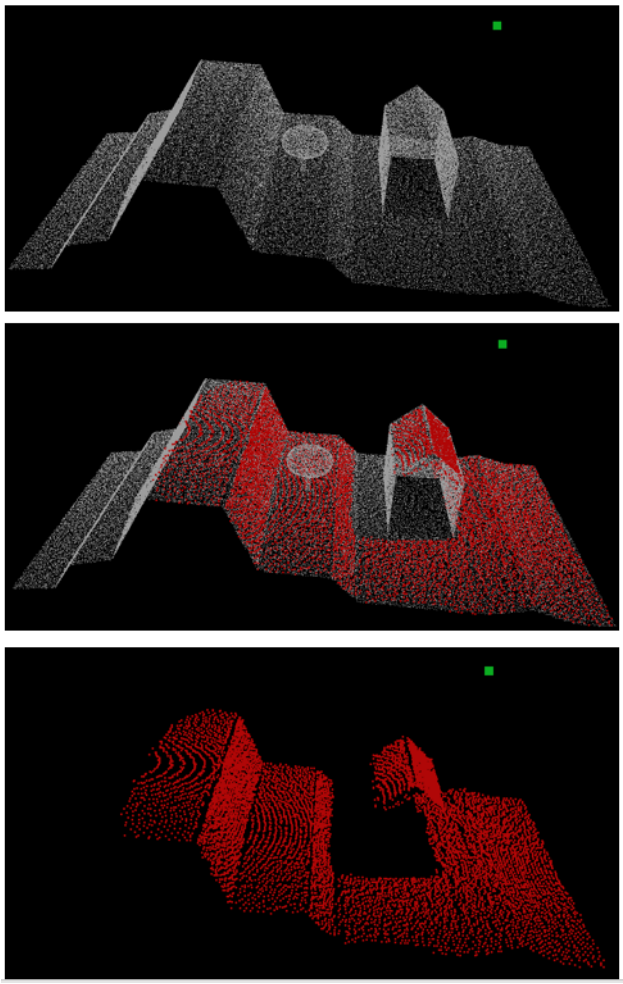


Figure 6.7: Visibility result of radial rays based on bisector approach

Figure 6.7 shows the visibility result of the radial ray query based on the bisector approach. The viewpoint is shown in green in the figure. The input point cloud is shown in white. Visible points from the viewpoint is stated in red. Closer to the viewpoint, denser result is obtained, due to the properties of the spherical rays. Notice that there is an obvious spherical texture pattern in the visible points.

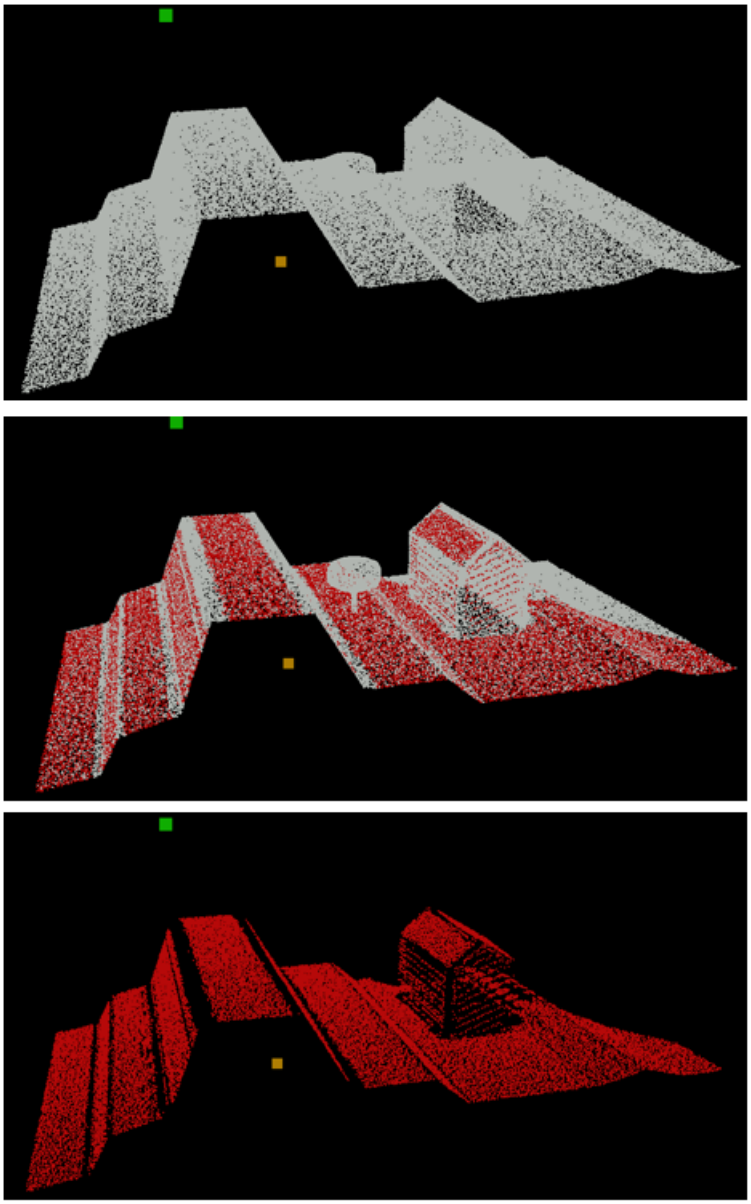


Figure 6.8: Visibility result of parallel rays based on bisector approach

Figure 6.8 shows the visibility result of the parallel ray query based on bisector approach. The ray direction is defined by a start point (in green) and an end point (in brown). The input point cloud is shown in white. Visible points from the viewpoint is stated in red. There are blank stripes in the figure, because of the interval of parallel rays.

Worth mentioning that both in Figure 6.7 and Figure 6.8 the point cloud of the tree is ignored. It does not mean the bisector approach does work for the trees. The reason is the spacial MAT geometry of the trees. The interior MAT of the tree has been classified as unknown MAT class. Figure 6.9 gives the answer why the MAT of the tree has been classified into unknown class. The principle of MAT separation is based on the MAT sheet value. Positive value stands for interior MAT, and negative value means exterior MAT. Zero value is classified as unknown MAT. In the Figure 6.9, the point cloud of the house is hollow, in other words there are no points at the bottom of the house. The point cloud of the tree is totally closed. The sheet value of MAT sheet a and MAT sheet c (both in green) is zero. The sheet value of sheet b is positive. Sheet a and b will be merged into one sheet in the end. Therefore, sheet a and b will be assigned a positive sheet value. And sheet a and b will be determined

as interior MAT. However, the sheet value of sheet c is always zero. Sheet c will be classified as unknown MAT.

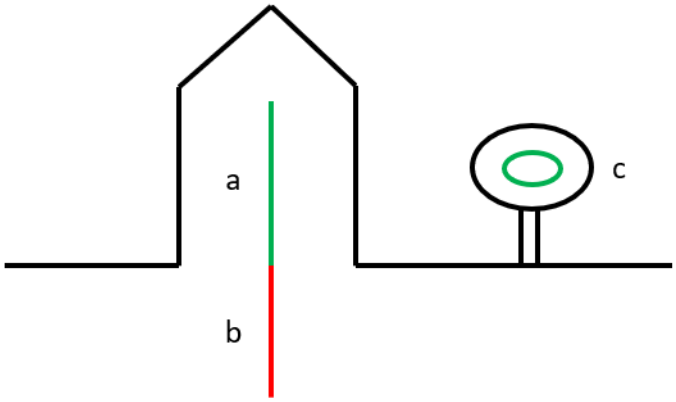


Figure 6.9: Part of the interior MAT of the artificial point cloud

AHN3 point cloud, a building in Rotterdam (Building A)

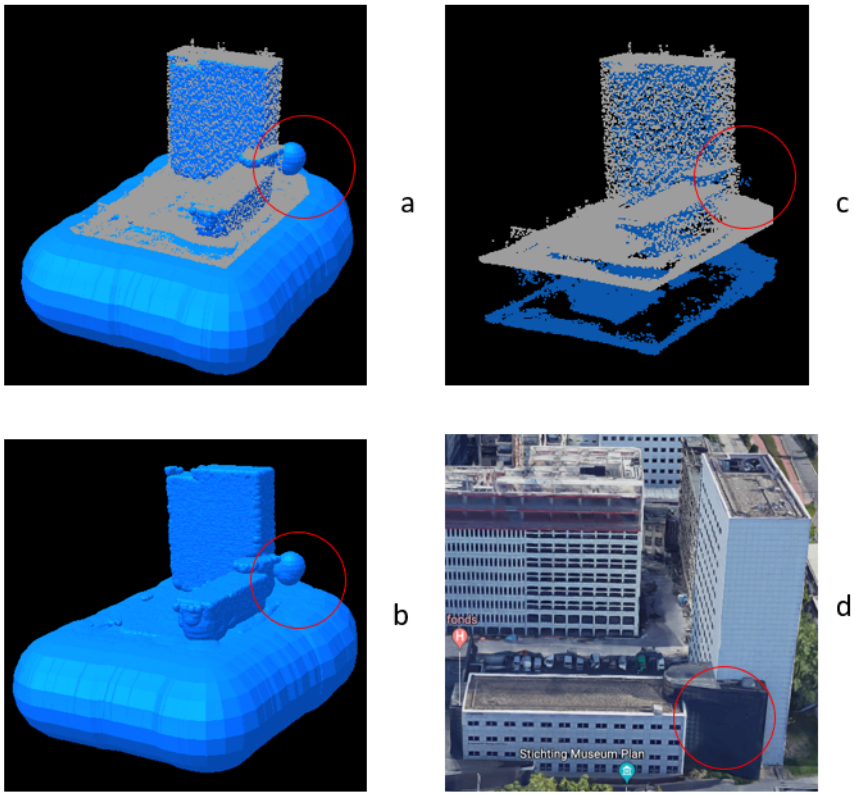


Figure 6.10: Interior MAT and medial balls results based on bisector approach

Figure 6.10 illustrates the interior MAT and medial ball result of *BuildingA*. The input AHN3 point cloud of building A is shown in Figure 6.10 *c* in white. The interior MAT points is shown in Figure 6.10 *a* and *c* in blue. Interior medial balls are displayed in Figure 6.10 *a* and *b* in blue colour. Comparing to the medial ball generated in Figure 6.5, the medial balls in Figure 6.10 are more accurate. There are only a few errors of the interior MAT and medial balls which are displayed within a red circle in Figure 6.10. The reason why the errors exist is that the input point

cloud is too sparse at that area. As displayed in Figure 6.10 *d*, the object in the red circle area is a facade made by glass. During laser scanning, the majority of the laser beams go through the glass and only a few beams are reflected. There are almost no points, which cause errors. Figures 6.12 6.13 6.14 and 6.15 show more visibility results of different datasets based on the bisector approach and the brute force rays query.

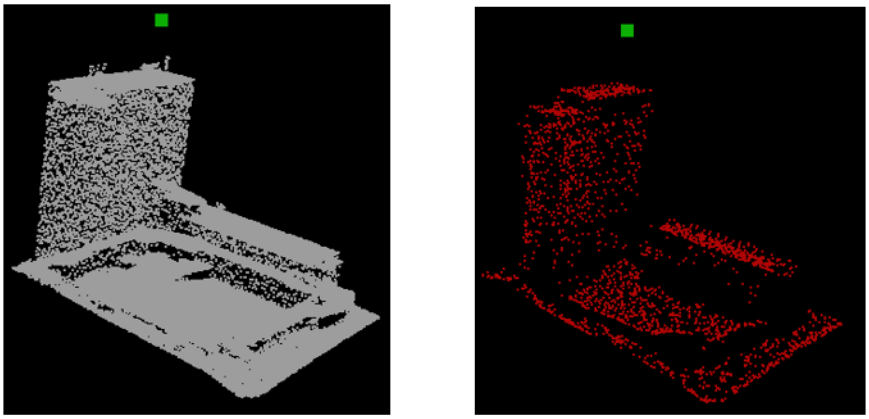


Figure 6.11: Visibility result of the Building A based on bisector, radial rays (left: input PC, right: result)

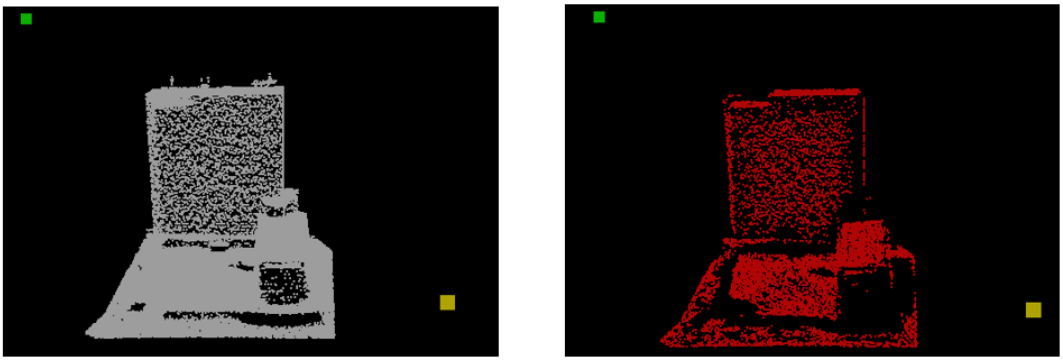


Figure 6.12: Visibility result of the Building A based on bisector, parallel rays (left: input PC, right: result)

AHN3 Point cloud Delft, Aula

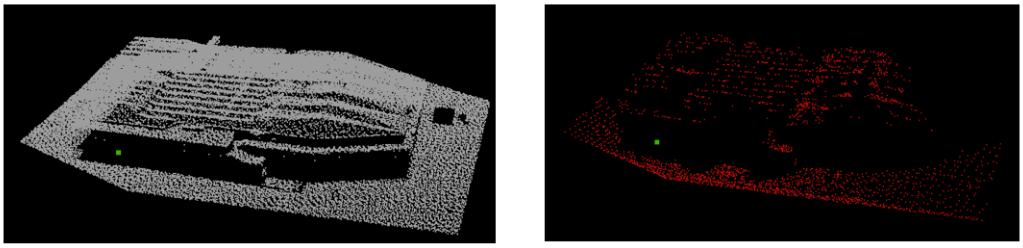


Figure 6.13: Visibility result of the Aula based on bisector, radial rays (left: input PC, right: result)

EMC Rotterdam 2016 Point cloud, Museum Boijmans Van Beuningen

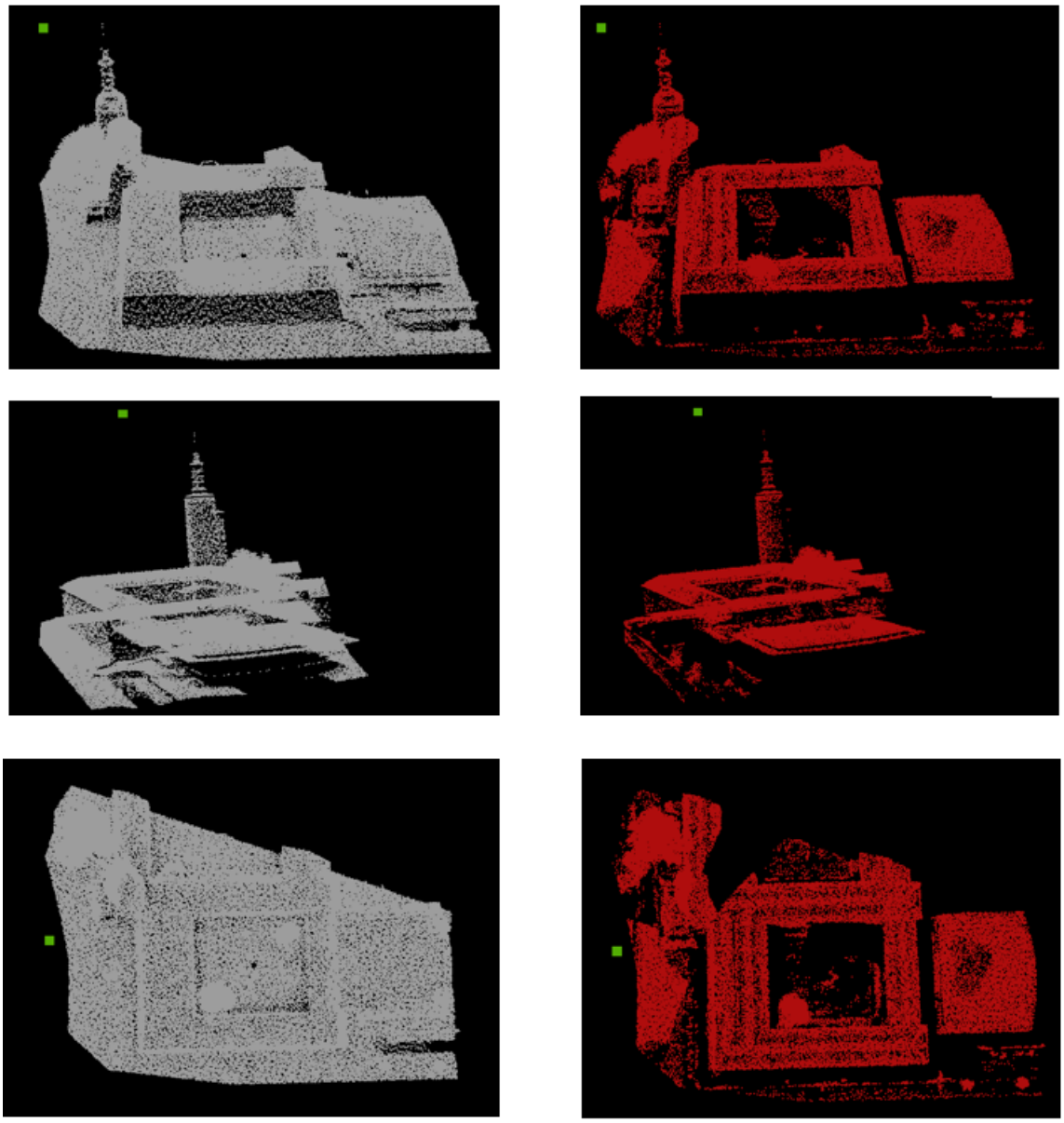


Figure 6.14: Visibility result of Museum Boijmans Van Beuningen based on bisector, brute force approach (left: input PC, right: result)

Dense point cloud Bergamo

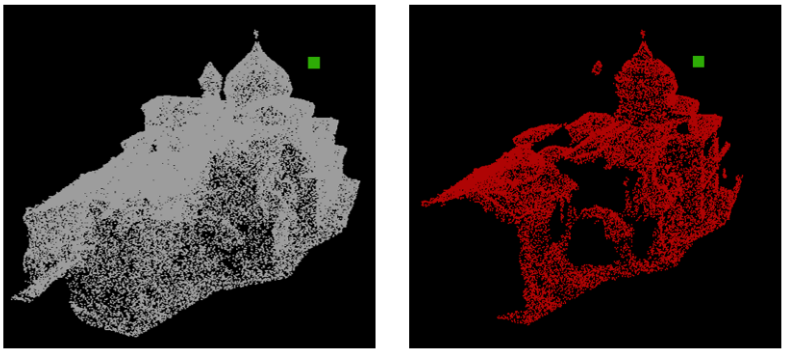


Figure 6.15: Visibility result of buildings based on bisector, brute force approach (left: input PC, right: result)

6.2 VALIDATION

As described in section 6.1, the normal reorientation is not able to deliver satisfying visibility results when processing the AHN3 dataset. Therefore, this section focuses on the validation of the bisector based approach.

6.2.1 Ray tracing vs brute force approach

If the interior MAT generated by the bisector approach is correct, the brute force approach will always give correct result, because it checks every medial ball. The result of the brute force approach can be used to evaluate the results of radial rays and parallel rays.

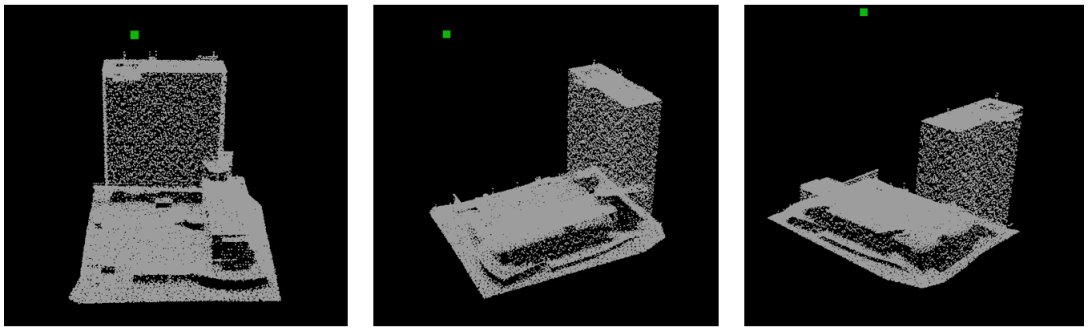


Figure 6.16: Original point cloud of Building A, viewing from three angles, left angle α, medium angle β, right angle γ, viewpoint in green, original point cloud in white

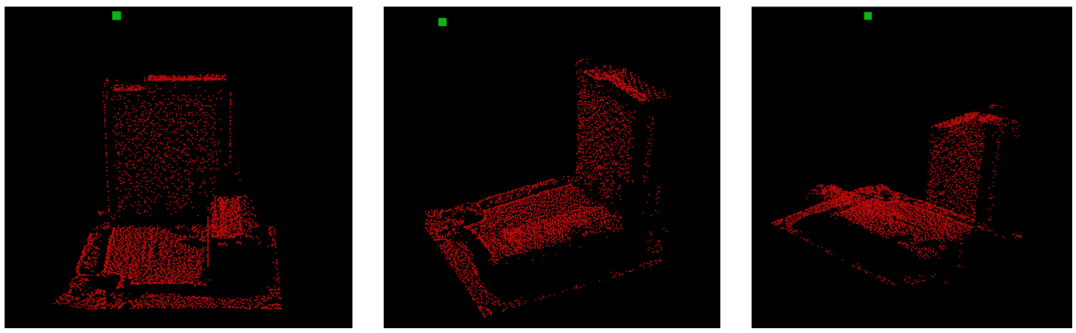


Figure 6.17: Radial ray visibility result of Building A, viewpoint in green, visible points in red

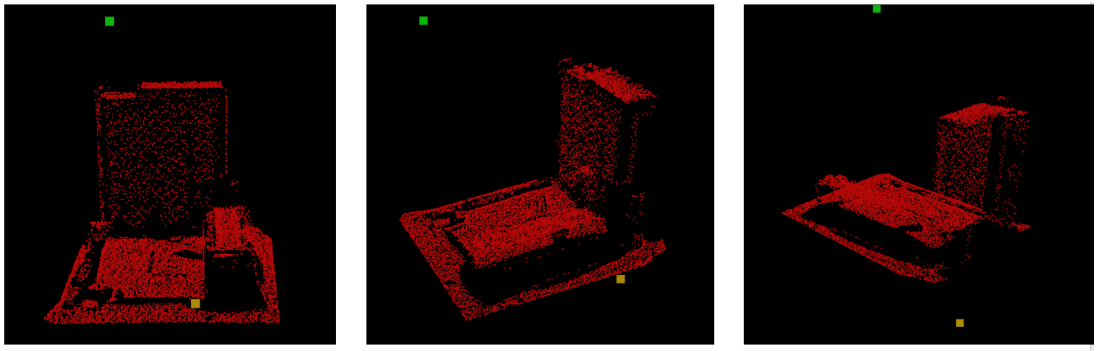


Figure 6.18: Parallel ray visibility result of Building A, start point in green, end point in yellow, visible points in red

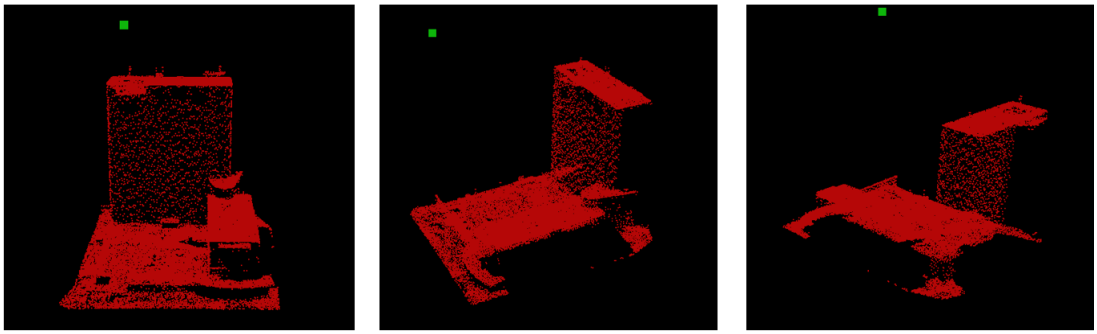


Figure 6.19: Brute force visibility result of building A, viewpoint in green, visible points in red

Figure 6.16 shows a viewpoint and an input point cloud viewing from three angles α, β, γ. Figure 6.17 illustrates the corresponding radial ray result. Parallel ray result is stated in Figure 6.18. The brute force approach result is displayed in Figure 6.19. In the brute force approach all medial balls are traversed, so the densest result is acquired. There is an obvious characteristic of ray tracing. Not all the rays have an intersection with the input, which results in relatively sparser result. For radial rays, the closer to the viewpoint, the denser the result. The result of parallel ray approach have equal interval.

6.2.2 Brute force approach vs open-source software

In practice, error exists in the interior MAT separated by the bisector approach. And it results in inaccuracy of the brute force approach. To evaluate the performance of the brute force approach, comparisons are made between an open-source software and the brute force approach. Figure 6.20 shows the results of the brute force approach and the result of hidden point remover of CloudCompare. In general the point the density of two results looks the same. There are points missing in the result of the brute force approach.

However, we do not know if the result of the hidden point remover is correct. This is an obvious limitation of the validation. To evaluate both of the brute force result and hidden point remover result, ground truth is required. And the viewpoint need to be put at exactly the same position.

The quality of the brute force result can be reflected on the quality of the interior medial balls. Figure 6.21 shows the inertial medial balls used in brute force approach. The errors mainly locate nearby the trees. The reason can be that it is challenge to extract perfect interior MAT at the points of the trees, due to the special geometry of the tree points. At the top of the tower, the original point cloud is too sparse to generate a sufficient interior MAT.

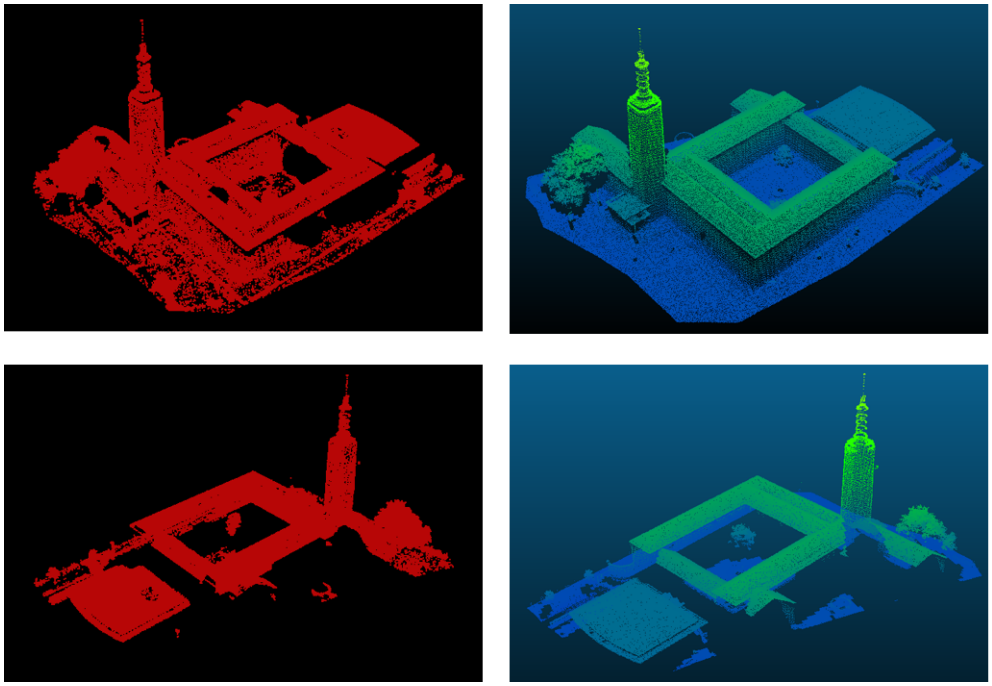


Figure 6.20: brute force approach vs hidden point cloud remover in CloudCompare

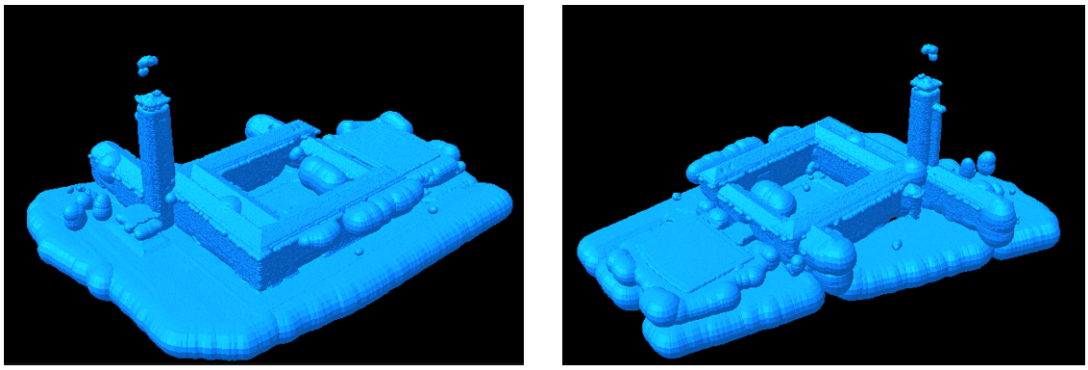


Figure 6.21: Interior balls of the point cloud of the Museum Boijmans Van Beuningen

6.3 EFFICIENCY PERFORMANCE

6.3.1 Brute force approach vs ray tracing approach

Table 6.1 shows the time cost of each approach. The input is the point cloud of *BuilindA*. The three different approaches share the same viewpoint. There is an extra end point in parallel ray approach. The visibility results have been stated in Figure 6.17, 6.18 and 6.19. According to the table, the brute force approach is faster than the other two approaches. All these three approaches are based on single ray query (see Section 4.4.1). If the input point cloud is the same, the interior medial balls generated will be the same. Running time difference is decided by the number of single ray query. Therefore, the reason of why the brute force approach is the fastest can be that: the number of single ray query in the other two approaches is large than the brute force approach.

For brute force approach, it is only influenced by the size of the input point cloud. And every time the number of the single ray query equals the number (51567 in the table) of the input point cloud. The maximum running time of the brute force approach can be stated as following.

$$T_{sum} = N_{ball} * N_{ball} * T_{unit}$$

where N_{ball} is the total number of all interior medial balls, T_{sum} is the total running time of the process, and T_{unit} is time cost of checking if the input ray intersects with the input ball.

In practice the running time is smaller than the running time stated above. In the brute force query process, if a ray is already blocked by one of the medial balls, the query of this ray will stop immediately and mark the input point as invisible. Then the next ray query will start to check the next input point. Therefore, not always all medial balls are checked in a single ray query. Only when the input point is a visible point, all the medial ball is checked.

For the ray tracing approach, the sum of the checking times can be calculated by the following formula.

$$T_{sum} = \sum_1^i (B_j) * T_{unit}$$

where i is the number of intersected rays, B_j means the total number of medial balls in all intersected bounding boxes, T_{sum} means the total time used in the process and T_{unit} is the same as the T_{unit} in brute force process.

In term of time cost, brute force approach has an advantage when the input point cloud is small. When handling a massive point cloud, the radial and parallel approaches can be shorter.

Table 6.1: Running time of three approaches

Approach name	input point size	number of the ray generated	number of intecsect rays	running time (ms)
Radial ray	51567	80000	16109	22333
Parallel ray	51567	160801	35696	80591
Brute force	51567	-	-	7842

The efficiency of the brute force approach can improve by using spatial index. Original brute force approach needs to check every medial ball for each single ray query. By using spatial index, it is unnecessary to all the medial balls for a single query. Instead, only the balls located in interesting bounding boxes have the opportunity to be the visible ball and need to be checked. This characteristic is able to reduce the number of balls checked sharply, which results in high improvement on running speed. More discussion of efficiency improvement by using spatial index is explained in the following sections.

6.3.2 With and without a KD-Tree

The efficiency contribution of a KD-Tree data structure is described in this section. Table 6.2 states the records of the running time of the brute force approach process with a KD-Tree and without.

The difference between using a KD-Tree and not using a KD-Tree is the way of checking the medial ball. Without a KD-Tree, all the medial balls need to be check for a single ray query. With a KD-Tree, only the medial balls located in bounding boxes that intersect the ray will be checked.

Only taking the query process into consideration, visibility query with a KD-Tree is faster than the one without a KD-Tree (see Table 6.2 column 3 and 5). The larger the input point cloud is, the more processing time is saved. However, the KD-Tree generation process also takes time. If including the KD-Tree generation cost, the total processing time with a KD-Tree is longer than the one without a KD-Tree (see Table 6.2 column 4 and 5). But worth mentioning that the KD-Tree can be reused multiple times. For example, if there are 100 viewpoints, the KD-Tree only needs to be generated once and can be reused 100 times. Each time using a KD-Tree, query

Table 6.2: Running time of different inputs by using a KD-Tree and not using a KD-Tree

input point cloud size	KD-Tree generation cost (ms)	Query process cost with KD-Tree (ms)	Total Runing time With a KD-Tree (ms)	Total Runing time Without a KD-Tree (ms)
51567	9619	7842	17461	9461
128918	68680	40382	109062	51430
257835	269307	136270	405577	201024
463352	934075	320670	1254745	582918

time is saved. In total, the whole processing time can be lower than the one without using KD-Tree as long as the time saved is longer that the KD-Tree generation cost.

Notice that in the Table 6.3.1 the running time of query process with a KD-Tree is not fast. For 463352 points, the process takes 320 seconds. This is caused by the way of KD-Tree construction. As explained in Section 4.4, the KD-Tree construction contains two steps. First, create a KD-Tree based on the centres of the medial balls. Second, duplicate the medial balls to all the intersected bounding ball. The second step could steeply increase the number of the balls in the KD-Tree. Figure 6.22 gives an example of a medial ball with a large radius that intersects will all the bounding boxes. Since the medial intersects with all the bounding boxes, it will be duplicated into all the bounding boxes. If there are a number of this kind of large medial balls, the elements of the KD-Tree will increase severely.

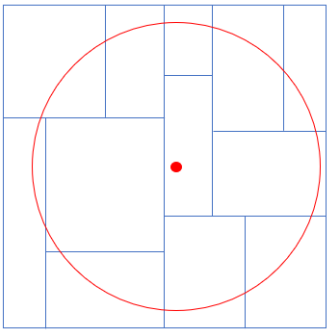


Figure 6.22: A medial ball with a large radius in the KD-Tree intersects with bounding boxes

Table 6.3: Number of the medial balls before and after duplication

input point cloud name	Total number of medial balls in the KD-Tree before duplication process	Total number of medial balls in the KD-Tree after duplication process	increase percentage
A area of dense point cloud Bergamo	41847	1861191	4448%
Arificial point cloud	100052	1118059	1117%
Building A (AHN3)	33439	3995752	11949%
Museum Bojimans Van Beuningen (EMC)	63814	3684302	5773%

The Table 6.3 records the numbers of medial balls before and after duplication of different datasets. These records show that the number of the medial balls in the KD-Tree grows by 44 to 119 times, which results in low efficiency.

6.3.3 With R-Tree and without R-Tree

The efficiency contribution of an R-Tree data structure is described in this section. Table 6.4 states the records of the running time of the brute force approach process with an R-Tree and without an R-Tree. According to the table, the R-Tree achieve a significant improvement of running time. For one dataset, the spatial index only need to be generated once and it can be reused for multiple viewpoints.

Table 6.4: Running time of different inputs by using an R-Tree and not using an R-Tree

input point cloud size	R-Tree generation cost (ms)	Query process cost with R-Tree (ms)	Total Running time With a R-Tree (ms)	Total Running time Without a R-Tree (ms)
51567	116	1764	1880	9461
128918	136	10241	10377	51430
257835	265	29530	30092	201024
463352	986	33580	34566	582918

6.3.4 KD-Tree VS R-Tree

Since the queries based on KD-Tree and R-Tree use the same interior MAT, KD-Tree and R-Tree spatial index have no influence on the geometrical quality of the result. They only have impact on the performance of running time of the process.

Figure 6.23 shows the difference of the running time of the query based on KD-Tree, R-Tree and without spatial index. The data is from Table 6.2 and Table 6.4. As indicated in Figure 6.23, R-Tree spatial index is able to sharply improve the query speed, comparing to the other two queries. As the size of the input point cloud increases, the improvement on query speed of R-Tree index is more obvious. The running cost based on the KD-Tree increases steeply. As mentioned in Section 6.22, the reason is that the number of elements in the KD-Tree has been increased sharply. However, in the R-Tree data structure, it takes the whole medial ball into consideration. Every time when a medial ball is inserted into the R-Tree, the bounding box of the medial ball is only inserted once. There is no medial ball duplication. Therefore, the number of the elements in the R-Tree always equals the size of the input point cloud.

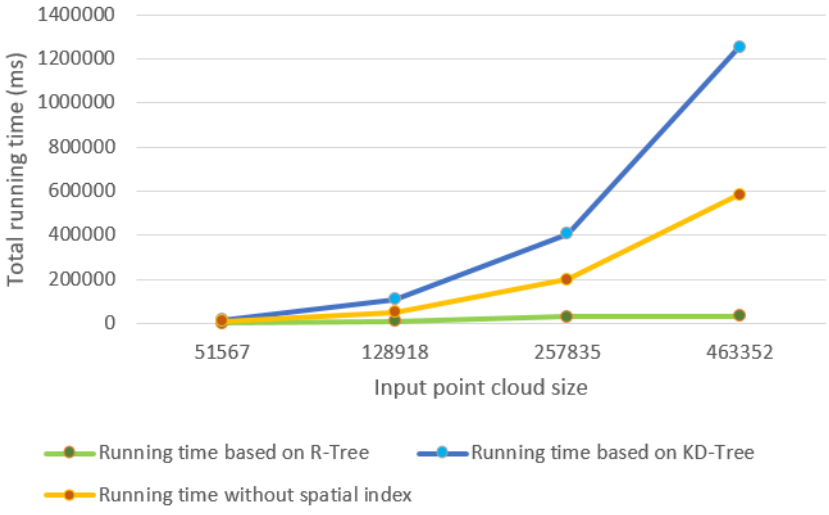


Figure 6.23: Running time comparison among KD-Tree, R-Tree and without spatial index (including the tree generation time cost)

In general, the running speed based on R-Tree is way faster than the one based on KD-Tree and the one without spatial index. It does not mean KD-Tree is unsuitable data structure for medial balls. I believe the problem is the medial ball duplication in the KD-Tree implementation. There is a limitation of the my KD-Tree implementation. I believe that if the ball duplication can be eliminated and store the entire ball as a element in the KD-Tree, the KD-Tree could also improve the running speed.

6.3.5 MAT simplification

Figure 6.24 shows the visibility results with different simplification thresholds. And their running time is stated in table 6.5. The simplification threshold is the overlap percentage of the medial balls. When the simplification threshold s is set to zero, there is no simplification. When the simplification threshold s is set to 0.05, the running speed is improved sharply without sacrificing the quality of the result.

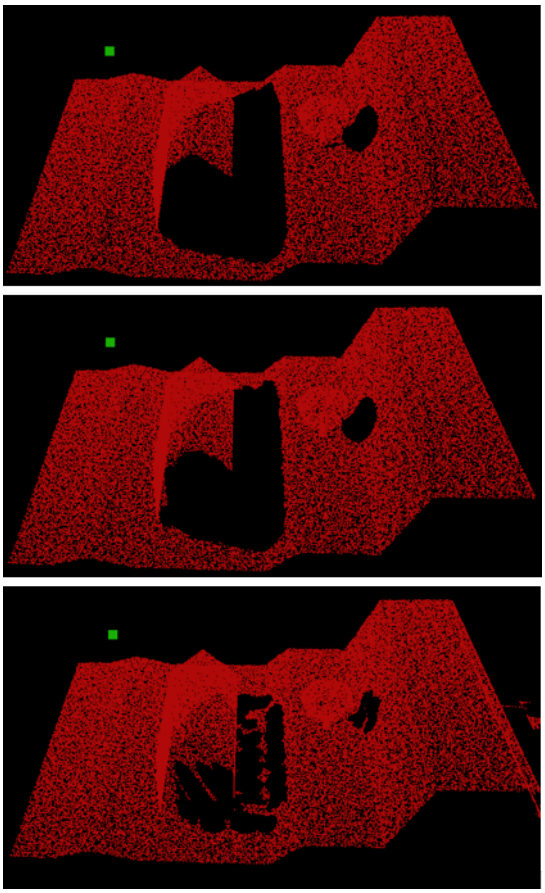


Figure 6.24: The visibility results with different simplification parameters (up simplification threshold s= 0, medium simplification threshold s= 0.05, down simplification threshold s= 0.1)

Table 6.5: Simplification results with different overlap ratio

Simplification threshold	Running time (ms)
0	6794
0.05	2296
0.1	875

6.3.6 Initial Radius

Theoretically, the initial radius of the shrinking ball algorithm does not matter. However, in practice, the initial radius does have influence on the performance. Figure 6.25 gives an example of how the initial radius influence the performance. In Figure 6.25, the upper picture has a very small initial radius. Therefore, even though there are errors at the top of the tower and tree, these wrong medial balls have less influence to block the view of sight. The lower has a larger initial radius. The wrong medial balls are larger, which means that those error balls can block more area.

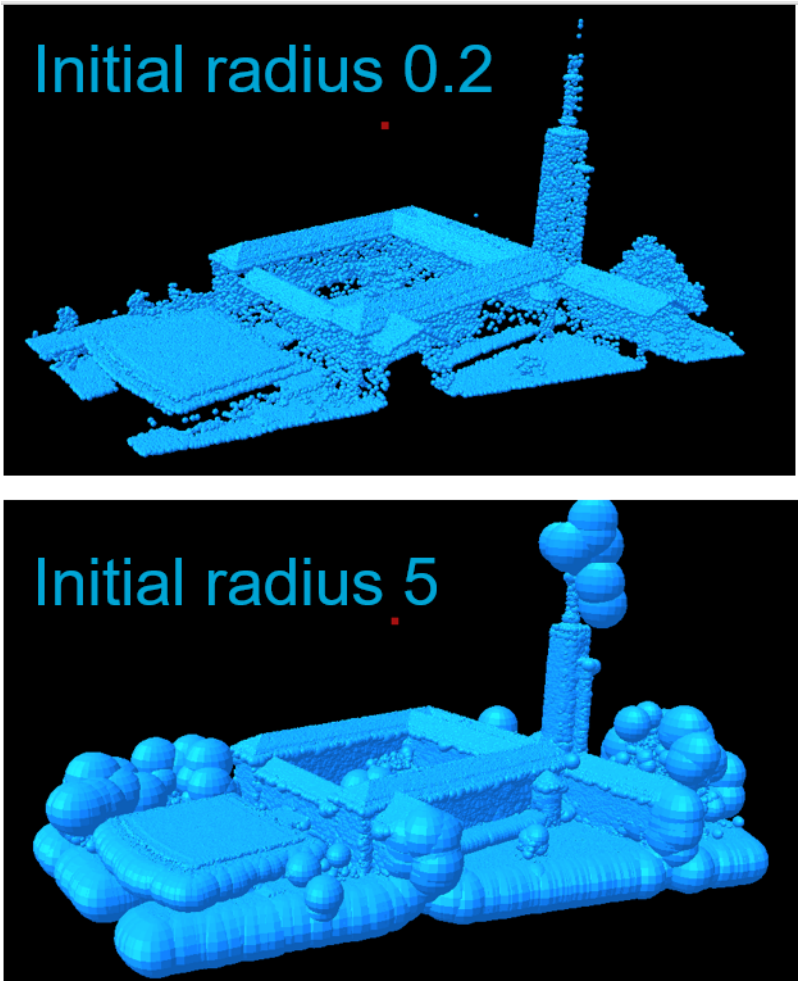


Figure 6.25: Influence of initial radius

7 CONCLUSION AND FUTURE WORK

In this chapter, research questions are answered. And future work of this topic is described.

7.1 CONCLUSION

7.1.1 Answers for the research questions

1. How can the Medial Axis Transform be used for visibility analysis in a point cloud?

To answer this question, the first need to answer which part of the Medial Axis Transform can be used. In this research, the Medial Axis Transform result of a point cloud is classified into two different types, interior and exterior MAT. Only the interior MAT can be used for visibility analysis. The exterior MAT always locates above the input point cloud (see Figure 7.1). And it obstructs the input point cloud, which results in negative contribution for visibility analysis.

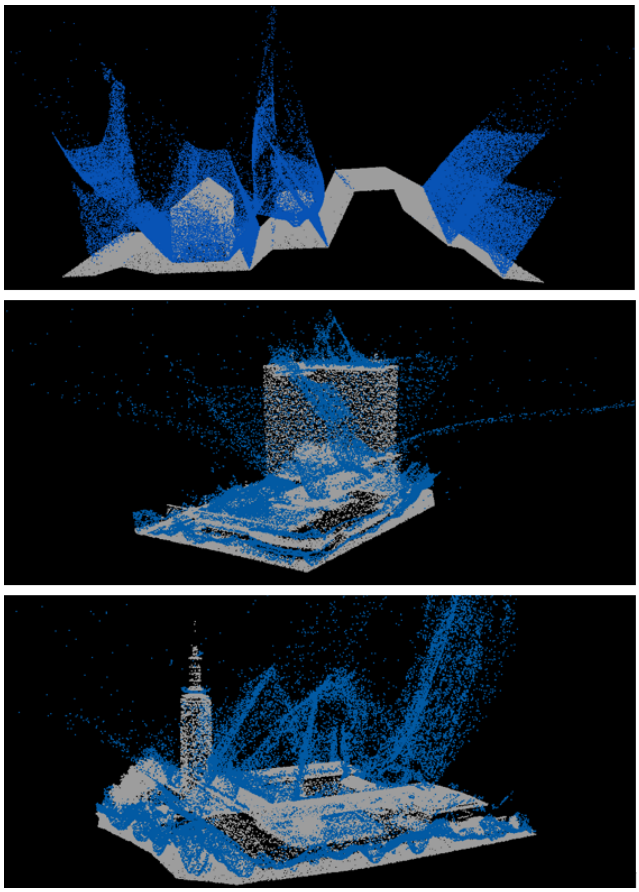


Figure 7.1: Exterior MAT with initial radius 200 meters

On the contrary, the interior MAT locates beneath the input point cloud. 3D volume information can be generated from the interior MAT, such as interior medial

balls (see Figure 4.15 and 6.10). The 3D volume of interior MAT is beneath the input point cloud in most cases. And the input point cloud is always lying on the 3D volume. Therefore, the 3D volume information can be used to create occlusion. It can block the points beneath it and has no influence on the points above it. This kind of occlusion information contributes to the visibility analysis.

2. What is an efficient spatial index for storing MAT for visibility analysis?

In this research, the interior MAT medial balls are used for visibility analysis. Therefore, this question can be asked more specificity as what is an efficient spatial index for storing interior MAT balls. The visibility query in this research is based on the single ray query (see Section 4.4.1). The target of the single ray query is the nearest intersected medial ball to the viewpoint. KD-Tree and R-Tree are a suitable data structure to create spatial index for medial ball theoretically, because KD-Tree and R-Tree can reduce the number of the medial balls checked by finding out the intersected bounding boxes. KD-Tree and R-Tree can divide the 3D space into partitions. Each partition corresponds a bounding box in the KD-Tree or R-Tree. During the ray query, only the medial balls in the intersected bounding boxes have possibility to be the visible ball, which sharply improve the efficiency of the query. Because the number of medial balls need to be checked reduces significantly.

However, in this thesis, there are limitations of the KD-Tree implementation. The ball duplication step steeply increases the number of balls stored in the KD-Tree, which results in relatively slow query. But I think, theoretically, KD-Tree is also able to improve the running speed, if the implementation of KD-Tree is better.

In general, the R-Tree sharply improve the running efficiency of queries. As shown in the Table 6.4, R-Tree severely decreases the running time of queries, comparing to the queries without R-Tree. Therefore, R-Tree is an efficient spatial index for MAT.

3. How to handle multiple viewpoints and different directions?

One can add multiple viewpoints to the software which is described in the Chapter 5. In the software, individual workflow of each viewpoint can be created and runs independently (see Figure 7.2). And results corresponding to each viewpoint will be generated independently as well.

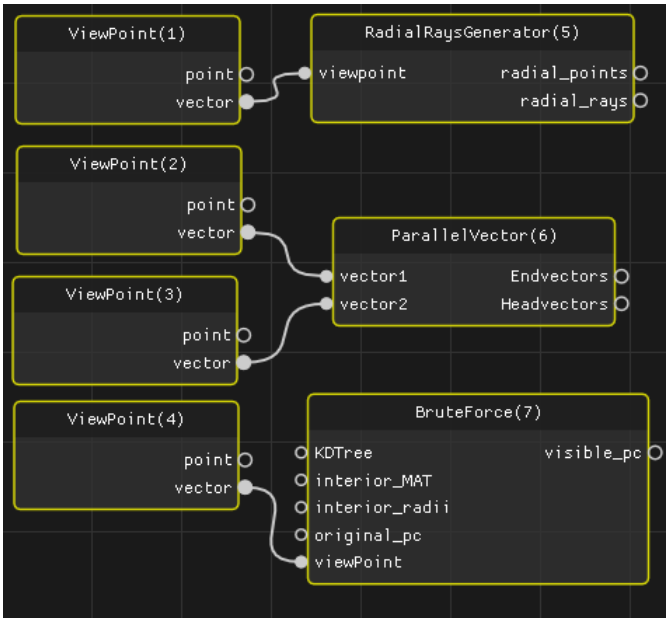


Figure 7.2: Workflow of multiple viewpoints in the geoflow software

For different view directions, it can be solved by using multiple parallel ray query nodes in the software. One group of parallel ray represents one direction. If visibility results of more directions are required, several groups of parallel ray can be used to obtain the results.

4. How to separate interior and exterior MAT?

In this research, two methods for MAT separation are proposed, the normal reorientation approach and the bisector based approach. The idea of normal reorientation is to flip the normals of point cloud and make them consistent. This normal reorientation approach works for the artificial point cloud generated from meshes but fails when experimenting AHN3 point cloud. The limitation of this approach is that it cannot determine the correct normal direction. And it may fail at corners and sharp edges.

The bisector based approach makes use of the MAT geometry property to separate the interior and exterior MAT. This method groups the MAT points into different sheets based on their distances. The sheet made up by the interior MAT points has a positive sheet value. The sheet value of the sheet made up by exterior MAT points is negative. Sheet value is the key information to determine interior MAT and exterior MAT.

5. What are the advantages and disadvantages of MAT based methods?

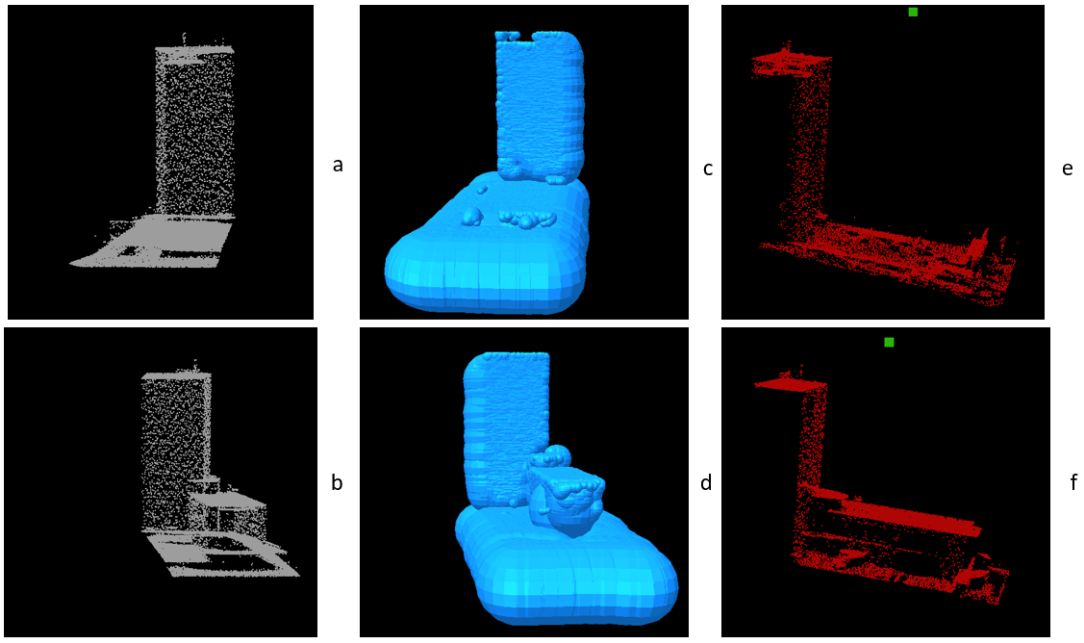


Figure 7.3: Visibility analysis by bisector based approach when dealing with surface missing

There are several advantages of MAT based methods. First, it avoids surface reconstruction. Second, it is able to handle surface missing of the input. In Figure 7.3, this point cloud of *BuildingA* is manually cut into two halves, left and right (see Figure 7.3 *a* and *b*). The whole surface is missing at where the building has been cut. However, the bisector based method can still generate proper medial balls (see Figure 7.3 *c* and *d*) and deliver decent visibility results (see Figure 7.3 *e* and *f*). Third, the MAT based method is able to deliver visibility result of multiple inputs. Figure 7.4 illustrates the situation when there are two input point cloud. One input is the artificial point cloud, and another input is a point cloud of a line. The idea is to use the artificial point cloud as a barrier and to check which part of the line point cloud is visible. The viewpoint is shown in green, the visible parts of the line point

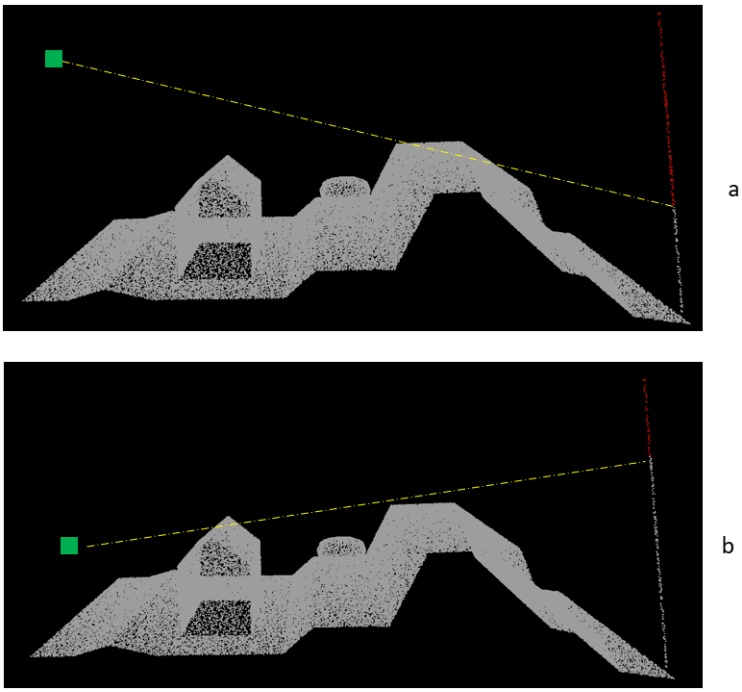


Figure 7.4: Visibility analysis of two inputs by bisector based approach

cloud is in red and invisible parts is in white. Fourth, this method achieve highly efficient performance on running speed by using the R-Tree spatial index (see Table 6.4 and Figure 6.23).

For the disadvantage, the performance of this kind of method is highly related the MAT quality generated from the input. It has a high requirement of interior and exterior MAT separation.

6. How good are the results of this method comparing to existing methods?

The MAT based method generates auxiliary geometry (medial balls) to achieve visibility analysis on a point cloud. I didn't test all the method introduced in the related work Chapter, but the hidden point remover. Accord to Figure 6.20, the result of the MAT based method looks similar comparing to the one of hidden point remover. However, there are points missing due to the error of interior medial balls (see Figure 6.20 and 6.21).

7. Does the methods work for all kinds of objects types of point cloud, such as trees, buildings, roads and grounds?

This method delivers good results when dealing with regular shape point cloud, such as building, roads and grounds. Even there is surfacing missing, it is still able to give decent visible results (see 7.3). However, when comes to the point cloud of tree, it is difficult to deliver good result, due to that there are errors at the interior medial balls of tree points.

8. Does the methods work for different datasets, e.g. LIDAR data, dense image matching

In this thesis, three different kinds of point cloud datasets are experimented. They are LIDAR point cloud, point cloud from dense image matching and artificial point cloud. The normal reorientation approach has difficulty to deal with AHN3 point cloud. The bisector based approach works for all the three datasets.

7.2 FUTURE WORK

7.2.1 Point cloud classification

The MAT of the point cloud can be group into different sheets. In this research the bisector geometry information of the sheet is used to determine interior and exterior MAT. There are more geometrical information of sheets, such as the average elevation, horizontal angles and vertical angles. These MAT geometrical information of the point cloud can be used to classify different point cloud objects.

7.2.2 Normal repair of point cloud

Since the adapted shrinking ball algorithm is able to determine interior and exterior MAT by using correct normal, there is a regular link between the normals and interior and exterior MAT. As explained in Section 4.1, the vector made up of input point and interior MAT, $\vec{pq}$ has the opposite direction to the correct normal $\vec{n}$ of the input point. The interior and exterior MAT correctly separated may contribute to the normal repair of the point cloud. Correct normals of point cloud benefits the visibility analysis.

7.2.3 Radius determination of ball-pivoting algorithm

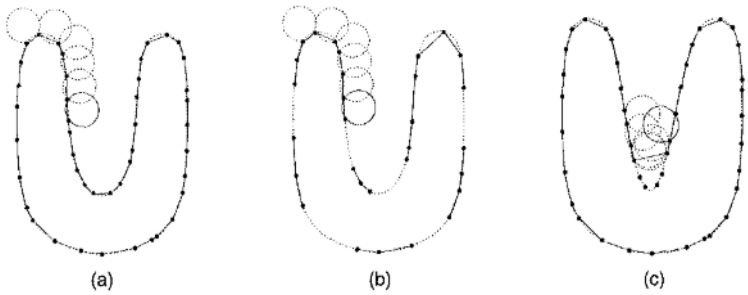


Figure 7.5: The Ball Pivoting Algorithm in 2D
[Bernardini et al., 1999]

Bernardini et al. [1999] proposed a ball-pivoting algorithm for surface construction of points (see Figure 7.5). The idea of the method is to roll a ball on the surface. Normally the rolling ball has a fixed radius. And it may fail at narrow place (see Figure 7.5 (c)). The MAT based method can benefit the ball-pivoting algorithm by setting a dynamic radius of the rolling ball. The radius of the rolling ball at a certain point can be calculated by using the radius of the interior medial ball. The surface reconstructed can be used for visibility analysis, as described in Section 3.2 (mesh based method).

BIBLIOGRAPHY

AHN3 (2019). AHN3 POINT CLOUD.

Alsadik, B., Gerke, M., and Vosselman, G. (2014). Visibility analysis of point cloud in close range photogrammetry. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2(5):9–16.

Amenta, N. and Kolluri, R. K. (2001). The medial axis of a union of balls. 20:25–37.

Arya, S. and Mount, D. M. (1993). Algorithms for fast vector quantization. *Data Compression Conference Proceedings*, pages 381–390.

Bartie, P., Reitsma, F., Kingham, S., and Mills, S. (2010). Advancing visibility modelling algorithms for urban environments. *Computers, Environment and Urban Systems*, 34(6):518–531.

Bentley, J. L. (1975). bentley_KDtree.pdf. *Communications of the ACM*.

Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G. (1999). The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359.

Biswas, J. (2012). Depth Camera Based Indoor Mobile Robot Localization and Navigation.

Blum, H. (1967). A transformation for extracting new descriptors. *Models for the perception of speech and visual form*.

Brookhuis, K. A., Driel, C. J. G. V., Hof, T., Arem, B. V., and Hoedemaeker, M. (2008). Driving with a congestion assistant ; mental workload and acceptance. *Applied Ergonomics*, 40(6):1019–1025.

Chire (2010). Visualization of a 3D RTree.

Google (2019). Google Maps.

Greene, N., Kass, M., and Miller, G. (2005). Hierarchical Z-buffer visibility.

Guttman, A. (1984). R-Trees A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING.

Jolliffe, I. (2002). *Principal component analysis. Springer Series in Statistics*.

Kaplan, M. R. (1987). The Use of Spatial Coherence in Ray Tracing. *Techniques for Computer Graphics*.

Katz, S., Tal, A., and Basri, R. (2007). Direct visibility of point sets. *ACM Transactions on Graphics*, 26(3):24.

Kazhdan, M., Bolitho, M., and Hoppe, H. (2006). Poisson Surface Reconstruction. In *Proceedings of Eurographics Symposium on Geometry Processing*.

Keiser, R. (1993). KDTree Source code.

Levoy, M. (2000). QSplat : A Multiresolution Point Rendering System for Large Meshes. pages 343–352.

Ma, J., Bae, S. W., and Choi, S. (2012). 3D medial axis point approximation using nearest neighbors and the normal field. *Visual Computer*, 28(1):7–19.

Mehra, R., Tripathi, P., Sheffer, A., and Mitra, N. J. (2010). Visibility of noisy point cloud data. *Computers and Graphics (Pergamon)*, 34(3):219–230.

Mena-Chalco (2010). Ray/box intersection.

Microsoft_AMP_Developing_Group (2014). C++ AMP (Accelerated Massive Parallelism).

Mittal, A. and Davis, L. S. (2004). Visibility Analysis and Sensor Planning in Dynamic Environments. pages 175–189.

Möller, T. and Trumbore, B. (2005). Fast, Minimum Storage Ray/Triangle Intersection. *Acm Siggraph*, (1):1–7.

Nan, L. (2018). Normal reorientation method.

PCL_Developing_Group (2019). Point Cloud Library.

Peters, R. (2017). Point cloud normal estimation based on the 3D medial axis transform.

Peters, R. (2018a). Geoflow nodes.

Peters, R. (2018b). *Geographical point cloud modelling with the 3D medial axis transform*. PhD thesis.

Peters, R. and Ledoux, H. (2016). Robust approximation of the Medial Axis Transform of LiDAR point clouds as a tool for visualisation. *Computers and Geosciences*, 90:123–133.

Peters, R., Ledoux, H., and Biljecki, F. (2015). Visibility analysis in a point cloud based on the medial axis transform. *Eurograph Workshop on Urban Data Modelling and Visualisation*, pages 7–12.

Ramon, G., Colunga, F., Zhuo, S., Lozano, R., Castillo, P., Ramon, G., Colunga, F., Zhuo, S., Lozano, R., and Vision, P. C. A. (2014). A Vision and GPS-Based Real-Time Trajectory Planning for a MAV in Unknown and Low-Sunlight Environments To cite this version : HAL Id : hal-00923131.

Skinkie (2010). Simple example of an R-tree for 2D rectangles.

Watson, I. D. and Johnson, G. T. (1987). Graphical estimation of sky view-factors in urban environments. *Journal of Climatology*, 7(2):193–197.

Williams, A., Barrus, S., Morley, R. K., and Shirley, P. (2005). An efficient and robust ray-box intersection algorithm. *ACM SIGGRAPH 2005 Courses on - SIGGRAPH '05*, page 9.

Wu, J. and Kobbelt, L. (2004). Optimized sub-sampling of point sets for surface splatting. *Computer Graphics Forum*, 23(3 SPEC. ISS.):643–652.

A APPENDIX C++ SOURCE CODE

A.1 APPENDIX I

Here is the main C++ code, masbnodes.cpp file, which contains all the methods stated in the Chapter 4. Other source code, such as the software UI and relevant third party library, can be found in the GitHub website (all code of this thesis are included).

Link: https://github.com/twut/geoflow-nodes.

Listing A.1: masbnodes.cpp

```
#include "masb_nodes.hpp"
#include "iostream"
#include <fstream>
#include <sstream>
#include <algorithm>
#include <thread>
#include <time.h>
#include<amp.h>
#include "region_grower.hpp"
#include "region_grower_testers.hpp"



namespace geoflow::nodes::mat {


void ComputeMedialAxisNode::process() {
std::cout << "computing MAT " << std::endl;
auto point_collection = input("points").get<PointCollection>();
auto normals_vec3f = input("normals").get<vec3f>();

masb::ma_parameters params;
params.initial_radius = param<float>("initial_radius");
params.denoise_preserve = param<float>("denoise_preserve");
params.denoise_planar = param<float>("denoise_planar");
params.nan_for_initr = param<bool>("nan_for_initr");

//std::cout << "where is the error " << std::endl;
// prepare data structures and transfer data
masb::ma_data madata;
madata.m = point_collection.size();

masb::PointList coords;
coords.reserve(madata.m);
for (auto& p : point_collection) {
coords.push_back(masb::Point(p.data()));
}
```

```
std::cout << "where is the error " << std::endl;

masb::VectorList normals;
normals.reserve(madata.m);
for (auto& n : normals_vec3f) {
normals.push_back(masb::Vector(n.data()));
}
masb::PointList ma_coords_(madata.m * 2);
std::vector<int> ma_qidx_(madata.m * 2);

madata.coords = &coords;
madata.normals = &normals;
madata.ma_coords = &ma_coords_;
madata.ma_qidx = ma_qidx_.data();

// compute mat points
masb::compute_masb_points(params, madata);

// retrieve mat points
vec1i ma_qidx;
ma_qidx.reserve(madata.m * 2);
for (size_t i = 0; i < madata.m * 2; ++i) {
ma_qidx.push_back(madata.ma_qidx[i]);
}

PointCollection ma_coords;
ma_coords.reserve(madata.m * 2);
for (auto& c : ma_coords_) {
ma_coords.push_back({ c[0], c[1], c[2] });
}

// Compute medial geometry
vec1f ma_radii(madata.m * 2);
vec1f ma_sepangle(madata.m * 2);
vec3f ma_spoke_f1(madata.m * 2);
vec3f ma_spoke_f2(madata.m * 2);
vec3f ma_bisector(madata.m * 2);
vec3f ma_spokecross(madata.m * 2);
for (size_t i = 0; i < madata.m * 2; ++i) {
auto i_ = i % madata.m;
auto& c = ma_coords_[i];
// feature points
auto& f1 = coords[i_];
auto& f2 = coords[ma_qidx[i]];
// radius
ma_radii[i] = Vrui::Geometry::dist(f1, c);
// spoke vectors
auto s1 = f1 - c;
auto s2 = f2 - c;
ma_spoke_f1[i] = { s1[0], s1[1], s1[2] };
ma_spoke_f2[i] = { s2[0], s2[1], s2[2] };
// bisector
s1.normalize();
s2.normalize();
auto b = (s1 + s2).normalize();
```

```
ma_bisector[i] = { b[0], b[1], b[2] };
// separation angle
ma_sepangle[i] = std::acos(s1*s2);
// cross product of spoke vectors
auto scross = Vrui::Geometry::cross(s1, s2).normalize();
ma_spokecross[i] = { scross[0], scross[1], scross[2] };
}
vec1i ma_is_interior(madata.m * 2, 0);
std::fill_n(ma_is_interior.begin(), madata.m, 1);

output("ma_coords").set(ma_coords);
output("ma_qidx").set(ma_qidx);
output("ma_radii").set(ma_radii);
output("ma_is_interior").set(ma_is_interior);
output("ma_sepangle").set(ma_sepangle);
output("ma_bisector").set(ma_bisector);
output("ma_spoke_f1").set(ma_spoke_f1);
output("ma_spoke_f2").set(ma_spoke_f2);
output("ma_spokecross").set(ma_spokecross);

std::cout << "computing MAT done" << std::endl;
}


void NegNormalDetector::process()
{
//----------input ----------//
auto pc = input("originalPC").get<PointCollection>();
auto normals_vec3f = input("normals").get<vec3f>();
auto offset = input("offset").get<float>();

//----------output----------//
PointCollection neg_pc;
vec3f normal_fixed;

// ----------process ----------//
for (int i = 0; i < normals_vec3f.size(); i++)
{
if (normals_vec3f[i][2] < offset )
{
neg_pc.push_back({ pc[i][0],pc[i][1],pc[i][2] });
normal_fixed.push_back({ -normals_vec3f[i][0] ,-normals_vec3f[i][1] ,-normals_vec3f[i
}
else
{
normal_fixed.push_back({ normals_vec3f[i][0] ,normals_vec3f[i][1] ,normals_vec3f[i][2
}
}
output("pc").set(neg_pc);
output("normals_fixed").set(normal_fixed);
}


void MATfilter::process() {
std::cout << "Filter running" << std::endl;
//auto pc = input("originalPC").get<PointCollection>();
```

```
auto matpoints = input("ma_coords").get<PointCollection>();
auto interior_index = input("ma_is_interior").get<vec1i>();
auto ma_radii = input("ma_radii").get<vec1f>();
//auto normals_vec3f = input("normals").get<vec3f>();

//float offset = input("offset").get<float>();
//float min_z = input("min_z").get<float>();


std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\ma_is_int
std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);
for (auto a : interior_index)
{
outfile << a << std::endl;
}
outfile.close();


//------output------------------//
PointCollection interior_mat;
PointCollection exterior_mat;
vec1f interior_radii;
vec1i interior_idx;
vec1f exterior_radii;
vec1i exterior_idx;

//----------process-----------//

/* std::string filepath2 = "c:\\users\\tengw\\documents\\git\\Results\\ex_in
std::ofstream outfile2(filepath2, std::fstream::out | std::fstream::trunc);

std::string filepath1 = "c:\\users\\tengw\\documents\\git\\Results\\in_filter
std::ofstream outfile1(filepath1, std::fstream::out | std::fstream::trunc);*


for (int i = 0; i < matpoints.size(); i++)
{
if (interior_index[i] == 1 )
{
interior_mat.push_back(matpoints[i]);
interior_radii.push_back(ma_radii[i]);
interior_idx.push_back(i);
//outfile1 << matpoints[i][0] << "," << matpoints[i][1] << "," << matpoints[

}
else
{
exterior_mat.push_back(matpoints[i]);
exterior_radii.push_back(ma_radii[i]);
exterior_idx.push_back(i-0.5*matpoints.size());
//outfile2 << i - 0.5*matpoints.size() << std::endl;
}
}

for (int i = 0; i < interior_mat.size(); i++)
{
```

```
//Vector3D in_pt(interior_mat[i][0], interior_mat[i][1], interior_mat[i][2]);
//Vector3D pc_point(pc[i][0], pc[i][1], pc[i][2]);

// normal points upwards
/*float flag = if_interiorMAT(in_pt, pc_point);

if (normals_vec3f[i][2] < 0)
{
if (flag < 0 || flag == 0) continue;
if (flag > 0)
{
auto temp_mat = interior_mat[i];
auto temp_radii = interior_radii[i];
auto temp_index = interior_idx[i];

interior_mat[i] = exterior_mat[i];
interior_radii[i] = exterior_radii[i];
interior_idx[i] = exterior_idx[i];

exterior_mat[i] = temp_mat;
exterior_radii[i] = temp_radii;
exterior_idx[i] = temp_index;
}
}
if (normals_vec3f[i][2] > 0)
{
if (flag > 0 || flag == 0) continue;
if(flag <0)
{
auto temp_mat = interior_mat[i];
auto temp_radii = interior_radii[i];
auto temp_index = interior_idx[i];

interior_mat[i] = exterior_mat[i];
interior_radii[i] = exterior_radii[i];
interior_idx[i] = exterior_idx[i];

exterior_mat[i] = temp_mat;
exterior_radii[i] = temp_radii;
exterior_idx[i] = temp_index;
}

}*/

}

//outfile1.close();
//outfile2.close();

std::cout << "Number of input points:" << matpoints.size() << std::endl;
std::cout << "Number of interior MAT points :" << interior_mat.size() << std::endl;
std::cout << "Number of exterior MAT points :" << exterior_mat.size() << std::endl;
output("interior_mat").set(interior_mat);
output("interior_radii").set(interior_radii);
output("interior_idx").set(interior_idx);
```

```
output("exterior_mat").set(exterior_mat);
output("exterior_radii").set(exterior_radii);
output("exterior_idx").set(exterior_idx);
}
void RegionGrowMedialAxisNode::process()
{
auto ma_coords = input("ma_coords").get<PointCollection>();
auto ma_bisector = input("ma_bisector").get<vec3f>();
auto ma_sepangle = input("ma_sepangle").get<vec1f>();
auto ma_radii = input("ma_radii").get<vec1f>();

regiongrower::RegionGrower<MaData, Region> R;
R.min_segment_count = param<int>("min_count");

MaData D(ma_coords, ma_bisector, ma_sepangle, ma_radii, param<int>("k"));

switch (param<int>("method"))
{
case 0: {
AngleOfVectorsTester T_bisector_angle(param<float>("bisector_angle"));
R.grow_regions(D, T_bisector_angle); break;
} case 1: {
DiffOfAnglesTester T_separation_angle(param<float>("separation_angle"));
R.grow_regions(D, T_separation_angle); break;
} case 2: {
BallOverlapTester T_ball_overlap(param<float>("ball_overlap"));
R.grow_regions(D, T_ball_overlap); break;
} case 3: {
CountTester T_shape_count(param<int>("shape_count"));
R.grow_regions(D, T_shape_count); break;
} default: break;
};

std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\segment_i
std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);

vec1i segment_ids;
for (auto& region_id : R.region_ids) {
segment_ids.push_back(int(region_id));
}

std::set<int> id_values;
for (int i = 0; i < ma_coords.size(); i++)
{
id_values.insert(segment_ids[i]);
outfile << ma_coords[i][0] << "," << ma_coords[i][1] << "," << ma_coords[i][
}
outfile.close();
output("segment_ids").set(segment_ids);
std::cout << "total classes:" << id_values.size() << std::endl;
for (set<int>::iterator it = id_values.begin(); it != id_values.end(); it++)
{
std::cout << *it << " occurs " << std::endl;
}
}
```

```
void ShowClusterMAT::process()
{
//----------input----------//
int classifiaction = param<int>("classification");
auto segment_ids = input("segment_ids").get<vec1i>();
auto MAT_points = input("ma_coords").get<PointCollection>();
// ----------output----------//
PointCollection selected_mat;
// ----------process ----------//
std::set<int> s;
for (int i = 0; i < segment_ids.size(); i++)
{
s.insert(segment_ids[i]);
}

for (int i = 0; i < segment_ids.size(); i++)
{
if (segment_ids[i] == classifiaction)
{
selected_mat.push_back({ MAT_points[i][0],MAT_points[i][1],MAT_points[i][2] });
}
}

output("mat").set(selected_mat);
}
void GetClusterSheets::process()
{
std::cout << "Trying to get all the sheets" << std::endl;
//----------input----------//
auto segment_ids = input("segment_ids").get<vec1i>();
auto MAT_points = input("ma_coords").get<PointCollection>();
auto bisectors = input("ma_bisector").get<vec3f>();
auto radii = input("ma_radii").get<vec1f>();
//----------output----------//
std::vector<PointCollection> vec_sheets;
std::vector<float> vec_bisectors;
std::vector<vec1f> vec_radii;
std::vector<vec1i> vec_index;
//------process------//
std::set<int> id_values;

//std::cout << "size of ALL MAT points:" << MAT_points.size() << std::endl;
//std::cout << "size of segment_id:" << segment_ids.size() << std::endl;
// segment_id is the cluster id of all MAT, size = size of ALL MAT

for (int i = 0; i < segment_ids.size(); i++)
{
id_values.insert(segment_ids[i]);
}

int num = id_values.size(); // how many different clusters generated

for (set<int>::iterator it = id_values.begin(); it != id_values.end(); it++)
{
PointCollection one_mat_sheet;
float one_sheet_bisector =0;
```

```
vec1f one_radii;
vec1i one_index;

for (int i = 0; i < segment_ids.size(); i++)
{

if (*it == segment_ids[i])
{

one_mat_sheet.push_back({ MAT_points[i][0],MAT_points[i][1],MAT_points[i][2]
one_sheet_bisector += bisectors[i][2];
one_index.push_back(i);
one_radii.push_back(radii[i]);

}
//one_sheet_bisector = one_sheet_bisector / count;
}
vec_sheets.push_back(one_mat_sheet);
vec_index.push_back(one_index);
vec_radii.push_back(one_radii);
std::cout << "one sheet bisector value:" << one_sheet_bisector << std::endl;
vec_bisectors.push_back(one_sheet_bisector);
}

output("vec_sheets").set(vec_sheets);
output("vec_bisectors").set(vec_bisectors);
output("vec_radii").set(vec_radii);
output("vec_index").set(vec_index);

std::cout << "sheets done" << std::endl;
std::cout << "bisector group size:" << vec_bisectors.size() << std::endl;
std::cout << "in total:" << vec_sheets.size() << " obtained" << std::endl;
}
void MATSeparation::process()
{
std::cout << "MAT separation starts" << std::endl;
// ----------  input ------------//
auto vec_sheets = input("vec_sheets").get<std::vector<PointCollection>>();
auto vec_bisectors = input("vec_bisectors").get<std::vector<float>>();
auto vec_radii = input("vec_radii").get<std::vector<vec1f>>();
auto vec_index = input("vec_index").get<std::vector<vec1i>>();

//auto points = input("points").get<PointCollection>();
int offset = param<float>("offset");

// ------------output-----------//
PointCollection interior_MAT;
PointCollection exterior_MAT;
PointCollection unclassified_MAT;


vec1f ex_radii;
vec1f in_radii;
vec1i in_index;
vec1i ex_index;
```

```
//————————process————————————//
int num = 0;
for (auto sheet : vec_sheets)
{
for (auto pt : sheet)
num++;
}
std::cout << "input_mat_point_size:" << num << std::endl;

for (int i=1;i<vec_bisectors.size();i++)
{
if (vec_bisectors[i] < 0)
{
//std::cout << "ex sheet size:" << vec_sheets[i].size() << std::endl;
for (auto pt : vec_sheets[i])
{

exterior_MAT.push_back(pt);

}
for (auto r : vec_radii[i])
{
ex_radii.push_back(r);
}
for (auto id : vec_index[i])
{
ex_index.push_back(id);
}

}
//if (vec_bisectors[i] >= 0)
else
{
//std::cout << "in sheet size:" << vec_sheets[i].size() << std::endl;
for (auto pt : vec_sheets[i])
{
interior_MAT.push_back(pt);
}
for (auto r : vec_radii[i])
{
in_radii.push_back(r);
}
for (auto id : vec_index[i])
{
in_index.push_back(id);
}
}
}



output("interior_mat").set(interior_MAT);
output("exterior_mat").set(exterior_MAT);
output("unclassified_mat").set(unclassified_MAT);
```

```
output("in_radii").set(in_radii);
output("ex_radii").set(ex_radii);
output("in_index").set(in_index);
output("ex_index").set(ex_index);

}

void MATsimplification::process() {
std::cout << "simplification is running" << std::endl;
//--------------input-----------------//
auto point_collection = input("interior_mat").get<PointCollection>();
auto radii = input("interior_radii").get<vec1f>();
auto indice = input("interior_idx").get<vec1i>();
float threshold = input("threshold").get<float>();
//----------output-----------------//
PointCollection sim_mat;
vec1f  sim_radii;
std::vector<std::vector<int>> sim_idx;




if (threshold == 0)
{
sim_mat = point_collection;
sim_radii = radii;
for (auto idx : indice)
{
std::vector<int> idx_vec;
idx_vec.push_back(idx);
sim_idx.push_back(idx_vec);
}
}
else
{

std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\Sim_MAT_c
std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);

std::vector<KdTree::sphere> MAT_sph;
KdTree::sphere sp1;
Vector3D p1 = { point_collection[0][0],point_collection[0][1],point_collecti
sp1.pos = p1;
sp1.radius = radii[0];
sp1.index.push_back(indice[0]);

MAT_sph.push_back(sp1);


for (int i = 0; i < point_collection.size(); i++)
{
Vector3D v1 = { point_collection[i][0],point_collection[i][1],point_collecti
bool ifsim = MATsimplification::ifSimplify(v1, radii[i], indice[i], threshol
}
```

```
std::cout << "MAT_to_kd_done" << std::endl;

for (int i = 0; i < MAT_sph.size(); i++)
{
arr3f point = { MAT_sph[i].pos.x,MAT_sph[i].pos.y,MAT_sph[i].pos.z };
sim_mat.push_back(point);
sim_radii.push_back(MAT_sph[i].radius);
sim_idx.push_back(MAT_sph[i].index);
outfile << MAT_sph[i].pos.x << "," << MAT_sph[i].pos.y << "," << MAT_sph[i].pos.z <<

}
outfile.close();
}



output("interior_mat").set(sim_mat);
output("interior_radii").set(sim_radii);
output("interior_idx").set(sim_idx);
std::cout << "Simplified_MAT_size:" << sim_mat.size() << std::endl;

}

void ComputeNormalsNode::process() {
//----------------input----------------//
auto point_collection = input("points").get<PointCollection>();




masb::ma_data madata;
madata.m = point_collection.size();
masb::PointList coords;
coords.reserve(madata.m);
for (auto& p : point_collection) {
coords.push_back(masb::Point(p.data()));
}
masb::VectorList normals(madata.m);
madata.coords = &coords;
madata.normals = &normals;

masb::compute_normals(params, madata);

//std::string filepath = "c:\\users\\tengw\\documents\\git\\Reorien\\las_normal.txt";
//std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);

vec3f normals_vec3f;
normals_vec3f.reserve(madata.m);
for (auto& n : *madata.normals) {
normals_vec3f.push_back({ n[0], n[1], n[2] });
//outfile << n[0] << "," << n[1] << "," << n[2] << std::endl;
}
//outfile.close();
output("normals").set(normals_vec3f);
}
```

```
void MultiKDtree::process() {
/* auto point_collection = input("points").get<PointCollection>();
masb::ma_data madata;
madata.m = point_collection.size();
auto number = point_collection.size();
vec3f points1 ;
vec3f points2 ;
Vector3D* mp_Points = new Vector3D[number];

KdTree* kd = NewBuildKdTree(mp_Points, number);
Vector3D center = (*kd).centerofBoundingBox();
for (int i = 0; i < number; i++) {
if (mp_Points[i][0] < center.x) {
points1.push_back({mp_Points[i][0],mp_Points[i][1],mp_Points[i][2] });
}
else {
points2.push_back({ mp_Points[i][0],mp_Points[i][1],mp_Points[i][2] });
}
}
int count1 = points1.size();
int count2 = points2.size();
Vector3D* m_Points1 = new Vector3D[count1];
Vector3D* m_Points2 = new Vector3D[count2];
for (int i = 0; i < count1;i++) {
m_Points1[i].x = points1[i][0];
m_Points1[i].y = points1[i][1];
m_Points1[i].z = points1[i][2];
}
KdTree* kd1 = NewBuildKdTree(m_Points1, count1);

for (int j = 0; j < count2; j++) {
m_Points2[j].x = points2[j][0];
m_Points2[j].y = points2[j][1];
m_Points2[j].z = points2[j][2];
}
KdTree* kd2 = NewBuildKdTree(m_Points2, count2);

output("KDTree1").set(kd1);
output("KDTree2").set(kd2);*/
}

void BuildKDtree::process()
{
clock_t starttime, endtime;
starttime = clock();

auto point_collection = input("points").get<PointCollection>();
auto radii = input("radii").get<vec1f>();
//////////////////////////
//auto indice = input("indice").get<vec1i>();
auto indice = input("indice").get<std::vector<vec1i>>();
masb::ma_data madata;
madata.m = point_collection.size();
m_nPoints = point_collection.size();
std::cout << "MAT_point_size:" << madata.m << std::endl;
```

```
KdTree::sphere* mp_Points = new KdTree::sphere[m_nPoints];
Vector3D* Points = new Vector3D[m_nPoints];


Vector3D center;

vec3f points;
points.reserve(madata.m);
for (auto& p : point_collection) {
points.push_back({ p[0], p[1], p[2] });
}
std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\MAT_points_out.txt"
std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);

std::vector<KdTree::sphere> allPointsVec;


for (int i = 0; i < m_nPoints; i++) {




mp_Points[i].pos.x = points[i][0];
mp_Points[i].pos.y = points[i][1];
mp_Points[i].pos.z = points[i][2];
mp_Points[i].radius = radii[i];
mp_Points[i].index = indice[i];
//mp_Points[i].index.push_back(indice[i]);


Points[i].x = points[i][0];
Points[i].y = points[i][1];
Points[i].z = points[i][2];




outfile<< mp_Points[i].pos.x << "," << mp_Points[i].pos.y << "," << mp_Points[i].pos.z
allPointsVec.push_back(mp_Points[i]);
}
outfile.close();

KdTree* kd = BuildKDtree::BuildKdTree(Points, m_nPoints,20);
center = (*kd).centerofBoundingBox();
(*kd).m_allballs = allPointsVec;

//std::cout << "center point of bounding box:" << center.x << "," << center.y << "," <
std::cout << "Total number of points in kd-tree"<< allPointsVec.size() << std::endl;
std::cout<< "number of bounding box:" << (*kd).m_maxpoint.size() << std::endl;
//std::cout << "Vector size of kdtreepoints:" << (*kd).m_boxKdTreePoint.size() << std
std::cout << "size of level:" << (*kd).m_currentlevel.size() << std::endl;

std::cout << "Level points are saving" << std::endl;


int new_count = 0;
```

```
for (int i =0; i<(*kd).m_maxpoint.size() ; i++)
{

//std::vector<KdTree::sphere> levelpoints = BuildKDtree::NewGetLevelPoints((
std::vector<KdTree::sphere> levelpoints = BuildKDtree::GetLevelPoints((*kd).


new_count +=levelpoints.size() ;
//std::cout <<"Number of points in each level:"<< levelpoints.size() << std:
(*kd).m_levelpoints.push_back(levelpoints);
}
std::cout << "total level points:" << new_count << std::endl;
std::cout << "KDTree output done" << std::endl;



/*std::cout << "Total level points:" << new_count << std::endl;
std::cout << "At the end the size of allPointsVec are:" << allPointsVec.size
std::cout << "The size of m_levelpoints:" << (*kd).m_levelpoints.size() << s
std::cout << "The size of boxs:" << (*kd).m_maxpoint.size() << std::endl;

std::cout << "checking points in each box" << std::endl;
int flag = 0;
for (int i = 0; i < (*kd).m_maxpoint.size(); i++) {
for (auto point : (*kd).m_levelpoints[(*kd).m_maxpoint.size() - i-1]) {
if(point.pos.x<= (*kd).m_maxpoint[i].x && point.pos.x >= (*kd).m_minpoint[i]
if(point.pos.y <= (*kd).m_maxpoint[i].y && point.pos.y >= (*kd).m_minpoint[i
if (point.pos.z <= (*kd).m_maxpoint[i].z && point.pos.z >= (*kd).m_minpoint[
{
flag++;
}

}
}
std::cout << "Correct points:" << flag<<std::endl;
std::cout << "Wrong points:" << new_count -flag << std::endl;*/


output("KDTree").set(kd);
endtime = clock();
std::cout << "KD-Tree running time:" << endtime - starttime << std::endl;
}

void AMPGPUQueryTest::process() {

clock_t starttime, endtime;
starttime = clock();

//-----------------input-------------------------------//
std::cout << "GPU query start" << std::endl;
auto kd = input("KDTree").get<KdTree*>();
auto point_collection = input("MATpoints").get<PointCollection>();
auto radii = input("radii").get<vec1f>();
//auto indice = input("indice").get<vec1i>();
//auto indice = input("indice").get<std::vector<vec1i>>();
```

```
auto interval = input("interval").get<float>();

// ----------output----------//
PointCollection visible_mat;
vec1f visible_radii;
vec1i visible_indice;

// ----------//
masb::ma_data madata;
madata.m = point_collection.size();
int number = point_collection.size();
std::cout << "MAT_point_size:" << madata.m << std::endl;


Vector3D* mp_Points = new Vector3D[number];




vec3f points;
points.reserve(madata.m);
for (auto& p : point_collection) {
points.push_back({ p[0], p[1], p[2] });
}
for (int i = 0; i < number; i++) {
mp_Points[i].x = points[i][0];
mp_Points[i].y = points[i][1];
mp_Points[i].z = points[i][2];
}

std::vector<KdTree::sphere> visble_sph;

Vector3D v1 = input("Vector1").get<Vector3D>();
Vector3DNew v1_new(v1);

std::vector<Vector3D> v2_list = AMPGPUQueryTest::SpherePoints(v1, 500, interval);
int linesize = v2_list.size();

std::vector<Vector3DNew> v2_new;
for (auto a : v2_list)
{
Vector3DNew new_a(a);
v2_new.push_back(new_a);
}
std::vector<int> result;

//run the query function here check if ray intersects with bounding boxes.
std::cout << "Ray-boxes_check_starts" << std::endl;

AMPGPUQueryTest::GPUQuery(v2_new, v1_new, linesize, kd, result);

std::cout << "Ray-boxes_check_done" << std::endl;

int GPU_box_check_count = 0;

std::vector<Vector3D> v2_intersected;
```

```
// processing bar                        //

//float progress = 0.0;
//while (progress < 1.0) {
//    int barWidth = 70;

//    std::cout << "[";
//    int pos = barWidth * progress;
//    for (int i = 0; i < barWidth; ++i) {
//        if (i < pos) std::cout << "=";
//        else if (i == pos) std::cout << ">";
//        else std::cout << " ";
//    }
//    std::cout << "] " << int(progress * 100.0) << " %\r";
//    std::cout.flush();

//    progress += 0.16; // for demonstration only
//}
//std::cout << std::endl;

for (int i = 0; i < result.size();i++) {
if (result[i] == 1)
{

//v2_intersected.push_back(v2_list[i]);

GPU_box_check_count++;

auto sph1 = AMPGPUQueryTest::GetOneLineResult(v1,v2_list[i],kd);
visble_sph.push_back(sph1);
}

}
std::cout << "Number of intersected directions:" << GPU_box_check_count << s
//————————————Mutiple Threads————————————————//
/* auto CutVecList = MutiThreadsOneQuery::CutVecList(v2_intersected, 4);

std::thread t[4];


int Threads_Count = 0;
for (std::vector<Vector3D> vec_cut : CutVecList) {
t[Threads_Count] = std::thread(AMPGPUQueryTest::GetQueryResult, vec_cut, v1,
t[Threads_Count].join();
Threads_Count++;
}*/




//std::cout << "Visble MAT size:" << visble_sph.size() << std::endl;



/*for (auto item : visble_sph) {
```

```
visible_mat.push_back({ item.pos.x,item.pos.y,item.pos.z });
visible_radii.push_back(item.radius);
visible_indice.push_back(item.index);
}*/
//////////////////
for (auto item : visble_sph) {
visible_mat.push_back({ item.pos.x,item.pos.y,item.pos.z });
visible_radii.push_back(item.radius);
for(auto idx: item.index)
visible_indice.push_back(idx);
}


endtime = clock();

output("MAT_points").set(visible_mat);
output("radii").set(visible_radii);
output("indice").set(visible_indice);
std::cout << "GPUNode running time:" << endtime - starttime << std::endl;

}

void MutiThreadsOneQuery::process() {
//----------------input----------------//
std::cout << "Mutiple Threads Query start" << std::endl;
auto kd = input("KDTree").get<KdTree*>();
auto point_collection = input("MATpoints").get<PointCollection>();
auto radii = input("radii").get<vec1f>();
// ----------------output----------------//
PointCollection visible_mat;
vec1f visible_radii;

// ----------------------------------------//
masb::ma_data madata;
madata.m = point_collection.size();
int number = point_collection.size();
std::cout << "MAT point size:" << madata.m << std::endl;


Vector3D* mp_Points = new Vector3D[number];




vec3f points;
points.reserve(madata.m);
for (auto& p : point_collection) {
points.push_back({ p[0], p[1], p[2] });
}
for (int i = 0; i < number; i++) {
mp_Points[i].x = points[i][0];
mp_Points[i].y = points[i][1];
mp_Points[i].z = points[i][2];
}

std::vector<sphere> visble_sph;
```

```
Vector3D v1 = input("Vector1").get<Vector3D>();

auto v2_list = OneQuery::SpherePoints(v1, 500);

clock_t starttime, endtime;
starttime = clock();
auto CutVecList = MutiThreadsOneQuery::CutVecList(v2_list, 4);


std::thread t[4];


int Threads_Count = 0;
for (std::vector<Vector3D> vec_cut : CutVecList) {
t[Threads_Count]=std::thread(MutiThreadsOneQuery::GetQueryResult, vec_cut, v
t[Threads_Count].join();
Threads_Count++;
}




////////////////////////////
/*int vetsize = v2_list.size();
std::vector<Vector3D> threadvec1;
std::vector<Vector3D> threadvec2;
std::vector<Vector3D> threadvec3;
std::vector<Vector3D> threadvec4;


std::for_each(begin(v2_list), begin(v2_list) + 0.25*vetsize, [&threadvec1](V
threadvec1.push_back(x);
});
std::for_each(begin(v2_list) + 0.25*vetsize, begin(v2_list) + 0.5*vetsize, [
threadvec2.push_back(y);
});
std::for_each(begin(v2_list) + 0.5*vetsize, begin(v2_list) + 0.75*vetsize, [
threadvec3.push_back(z);
});
std::for_each(begin(v2_list) + 0.75*vetsize, end(v2_list), [&threadvec4](Ve
threadvec4.push_back(v);

});


std::cout << "Vector divide done" << std::endl;



std::thread t1(MutiThreadsOneQuery::GetQueryResult, threadvec1, v1,kd,std::r
t1.join();
std::thread t2(MutiThreadsOneQuery::GetQueryResult, threadvec2, v1, kd, std:
t2.join();
std::thread t3(MutiThreadsOneQuery::GetQueryResult, threadvec3, v1, kd, std:
```

```
t3.join();
std::thread t4(MutiThreadsOneQuery::GetQueryResult, threadvec4, v1, kd, std::ref(visbl
t4.join();
*/

//////////////////////////
//2739

std::cout << "Multi Threads query done" << std::endl;
std::cout << "Visble MAT size:" << visble_sph.size() << std::endl;


for (auto item : visble_sph) {
visible_mat.push_back({ item.pos.x,item.pos.y,item.pos.z });
visible_radii.push_back(item.r);
}
endtime = clock();

output("MAT_points").set(visible_mat);
output("radii").set(visible_radii);
std::cout << "Multiple threads running time:" << endtime - starttime << std::endl;

}
void GetRadialRayResults::process()
{
std::cout << "get radial rays result starts" << std::endl;
clock_t starttime, endtime;
starttime = clock();
// ------------------input ---------------------//
auto kd = input("KDTree").get<KdTree*>();
auto point_collection = input("MATpoints").get<PointCollection>();
auto radii = input("radii").get<vec1f>();
auto viewpoint = input("viewpoint").get<Vector3D>();
auto radial_vectors = input("radial_rays").get<std::vector<Vector3D>>();
std::cout << "number of rays:" << radial_vectors.size()<< std::endl;
// ------------------- output -------------------//
std::vector<KdTree::sphere> visble_sph;

PointCollection visible_mat;
vec1f visible_radii;
vec1i visible_indice;

//----------process-------------//
std::vector<bool> ifinter;

for (int j = 0; j < radial_vectors.size(); j++)
{
Vector3D hit;
bool inter = GetRaysResult::CheckLineBox((*kd).m_min, (*kd).m_max, viewpoint, radial_v
ifinter.push_back(inter);
}
int count_intersect = 0;
for (int j = 0; j < radial_vectors.size(); j++)
{
if (ifinter[j] == 1)
{
```

```
count_intersect++;
auto sph1 = AMPGPUQueryTest::GetOneLineResult(viewpoint, radial_vectors[j],
visble_sph.push_back(sph1);
}
}
std::cout << "!!!!! number of intersection:" << count_intersect << std::endl

for (auto item : visble_sph) {
visible_mat.push_back({ item.pos.x,item.pos.y,item.pos.z });
visible_radii.push_back(item.radius);
for (auto idx : item.index)
visible_indice.push_back(idx);
}

std::cout <<"vis_id size"<< visible_indice.size() << std::endl;

endtime = clock();

output("MAT_points").set(visible_mat);
output("radii").set(visible_radii);
output("indice").set(visible_indice);
std::cout << "Get rays result running time:" << endtime - starttime << std::

}
void GetRaysResult::process()
{
std::cout << "get rays result starts" << std::endl;
clock_t starttime, endtime;
starttime = clock();
// ----------------input----------------//
auto kd = input("KDTree").get<KdTree*>();
auto point_collection = input("MATpoints").get<PointCollection>();
auto radii = input("radii").get<vec1f>();
auto headvectors = input("Headvectors").get<std::vector<Vector3D>>();
auto endvectors = input("Endvectors").get<std::vector<Vector3D>>();
std::cout << "the number of rays: " << endvectors.size() << std::endl;
//--------output----------------------//
std::vector<KdTree::sphere> visble_sph;

PointCollection visible_mat;
vec1f visible_radii;
vec1i visible_indice;

//----------process-------------//
// check ray intersect with box first here AMP query //
std::vector<bool> ifinter;

for (int j = 0; j < headvectors.size(); j++)
{
Vector3D hit;
bool inter = CheckLineBox((*kd).m_min,(*kd).m_max,headvectors[j], endvectors
ifinter.push_back(inter);
}

int count_intersect = 0;
for (int j = 0; j < headvectors.size(); j++)
```

```
{
if (ifinter[j] == 1)
{
count_intersect++;
auto sph1 = AMPGPUQueryTest::GetOneLineResult(headvectors[j], endvectors[j], kd);
visble_sph.push_back(sph1);
}
}
std::cout << "!!!!! number of intersection:" << count_intersect << std::endl;
for (auto item : visble_sph) {
visible_mat.push_back({ item.pos.x,item.pos.y,item.pos.z });
visible_radii.push_back(item.radius);
for (auto idx : item.index)
visible_indice.push_back(idx);
}



endtime = clock();

output("MAT_points").set(visible_mat);
output("radii").set(visible_radii);
output("indice").set(visible_indice);
std::cout << "Get rays result running time:" << endtime - starttime << std::endl;
}

void OneQuery::process() {
std::cout << "OneQuery start" << std::endl;

//----------------input----------------//

auto kd = input("KDTree").get<KdTree*>();
auto point_collection = input("MATpoints").get<PointCollection>();
auto radii = input("radii").get<vec1f>();


// ----------------output----------------//
PointCollection visible_mat;
vec1f visible_radii;

// --------------------------------------//

masb::ma_data madata;
madata.m = point_collection.size();
int number = point_collection.size();
std::cout << "MAT point size:" << madata.m << std::endl;


Vector3D* mp_Points = new Vector3D[number];




vec3f points;
points.reserve(madata.m);
for (auto& p : point_collection) {
```

```
points.push_back({ p[0], p[1], p[2] });
}
for (int i = 0; i < number; i++) {
mp_Points[i].x = points[i][0];
mp_Points[i].y = points[i][1];
mp_Points[i].z = points[i][2];
}

Vector3D v1 = input("Vector1").get<Vector3D>();

auto v2_list = OneQuery::SpherePoints(v1, 500);
//Vector3D v2 = input("Vector2").get<Vector3D>();



for (Vector3D v2 : v2_list) {
std::vector<Vector3D> pointlist;
std::vector<float> radiilist;
Vector3D hit;
int count = 0;
for (int i = 0; i < (*kd).m_maxpoint.size(); i++) {
bool a = OneQuery::CheckLineBox((*kd).m_minpoint[i], (*kd).m_maxpoint[i], v1

if (a == 1) {
//std::cout << "intersect bounding box level:" << i << std::endl;
//std::cout << "points inside" << std::endl;
for (auto pt : (*kd).m_levelpoints[(*kd).m_maxpoint.size() - i - 1]) {
count++;
//std::cout << pt_index << std::endl;

//std::cout << mp_Points[pt_index].x << "," << mp_Points[pt_index].y << ","
float dis = VisibiltyQurey::DistanceOfPointToLine(v1, v2, pt.pos);
//std::cout << "distance to line:" << dis << std::endl;

if (dis <= pt.radius) {
//std::cout << "Radius:" << pt.radius<<std::endl;
pointlist.push_back(pt.pos);
radiilist.push_back(pt.radius);
//visible_mat.push_back({ mp_Points[pt_index].x,mp_Points[pt_index].y,mp_Po
//visible_radii.push_back(radii[pt_index]);
}
}
//std::cout << "total points inside:" << count << std::endl;
}
}


if (pointlist.size() > 0) {
std::cout << "----------------this direction has :" << pointlist.size()<<"
float minDis = OneQuery::PointToPointDis(v1, pointlist[0]);
//float minDis = OneQuery::PointToPointDis(v1, pointlist[0]) - radiilist[0];

int flag = 0;
for (int i = 0; i < pointlist.size(); i++)
{
//float temp = OneQuery::PointToPointDis(v1, pointlist[i]) - radiilist[i];
```

```
float temp = OneQuery::PointToPointDis(v1, pointlist[i]);
if (temp < minDis) {
minDis = temp;
flag = i;
}
}
visible_mat.push_back({ pointlist[flag].x,pointlist[flag].y,pointlist[flag].z });
visible_radii.push_back(radiilist[flag]);
}
}

output("MAT_points").set(visible_mat);
output("radii").set(visible_radii);

}

void KDTreeNearestQurey::process() {
//auto viewpoint = input("viewpoint").get<PointCollection>();
std::cout << "Nearest points qurey starting" << std::endl;
int number = int(input("Number").get<float>());

PointCollection resultPoints;
resultPoints.reserve(number);

std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\Query_points_output
std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);

if (input("KDTree").has_data())
{
auto kdtree = input("KDTree").get<KdTree*>();
std::cout << "kdtree output successfully" << std::endl;
auto MATpoints = input("MATpoints").get<PointCollection>();


//Vector3D v1 = { -99.0594,-90.828,6.59866 };

//Vector3D v2 = { 0,0,0 };
//(*kdtree).queryLineIntersection(v1, v2, 100, 1, 1);
//for (int i = 0; i < number; i++)
//{
//    int index = (*kdtree).getNeighbourPositionIndex(i);
//    std::cout << "index   " << index << std::endl;
//    std::cout << "x   " << MATpoints[index][0] << std::endl;
//    std::cout << "y   " << MATpoints[index][1] << std::endl;
//    std::cout << "z   " << MATpoints[index][2] << std::endl;
//    resultPoints.push_back({ MATpoints[index][0], MATpoints[index][1] ,MATpoints[ind
//}
//output("Points").set(resultPoints);

//Vector3D v1 = { -99.0594,-90.828,6.59866 };
Vector3D v1 = input("ViewPoint").get<Vector3D>();
(*kdtree).setNOfNeighbours(number);
(*kdtree).queryPosition(v1);
for (int i = 0; i < number; i++) {
int index = (*kdtree).getNeighbourPositionIndex(i);
outfile << MATpoints[index][0] << "," << MATpoints[index][1] << "," << MATpoints[index
```

```
resultPoints.push_back({ MATpoints[index][0], MATpoints[index][1] ,MATpoints
}
output("Points").set(resultPoints);

}
else std::cout << "KDTree is empty " << std::endl;
outfile.close();
std::cout << "KDtree nearest query results done." << std::endl;
}
void KDTreeLineQurey::process() {

std::cout << "Line qurey starting" << std::endl;
//----------------Input Data----------------
int number = int(input("Number").get<float>());
float distance = input("Distance").get<float>();

Vector3D v1 = input("Vector1").get<Vector3D>();
Vector3D v2 = input("Vector2").get<Vector3D>();
auto kdtree = input("KDTree").get<KdTree*>();
auto MATpoints = input("MATpoints").get<PointCollection>();
//----------------------Output Data-----------
PointCollection resultPoints;
resultPoints.reserve(number);
std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\LineQuery
std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);
(*kdtree).queryLineIntersection(v1, v2, distance, 1, 1);
for (int i = 0; i < number; i++)
{
int index = (*kdtree).getNeighbourPositionIndex(i);
std::cout << "index    " << index << std::endl;
std::cout << "x    " << MATpoints[index][0] << std::endl;
std::cout << "y    " << MATpoints[index][1] << std::endl;
std::cout << "z    " << MATpoints[index][2] << std::endl;
resultPoints.push_back({ MATpoints[index][0], MATpoints[index][1] ,MATpoints
outfile << MATpoints[index][0] << "," << MATpoints[index][1] << "," << MATp
}
output("Points").set(resultPoints);
outfile.close();
}
void VisiblePCbyRTree::process()
{
std::cout << "Visible PC query by RTree starts " << std::endl;
clock_t starttime, endtime, endtimeRTree;
starttime = clock();
///----------------------input-----------------------------//
Vector3D viewpoint = input("viewPoint").get<Vector3D>();
auto interior_MAT = input("interior_MAT").get<PointCollection>();
auto radii = input("interior_radii").get<vec1f>();
auto pc = input("original_pc").get<PointCollection>();
auto in_idx = input("in_index").get<vec1i>();
std::cout << "size of interior MAT" << interior_MAT.size() << std::endl;
std::cout << "size of in_idx" << in_idx.size() << std::endl;

std::cout << "input pc size:" << pc.size() << std::endl;
///-------------output ------------------------//
PointCollection visible_pc;
```

```
//std::vector<int> vec_result_id;
std::set<int>  vec_result_id2;

///——————————————process————————————————//
namespace bg = boost::geometry;
namespace bgi = boost::geometry::index;
typedef bg::model::point<float, 3, bg::cs::cartesian> point;
typedef bg::model::box<point> box;
typedef bg::model::segment<point> seg;
typedef std::pair<box, unsigned> value;

// create the rtree using default constructor
bgi::rtree< value, bgi::quadratic<16> > rtree;

for (int i = 0; i < interior_MAT.size(); i++)
{
box b(point(interior_MAT[i][0] - radii[i], interior_MAT[i][1] - radii[i], interior_MAT
rtree.insert(std::make_pair(b, i));
}
std::cout << "RTree node inserted done!" << std::endl;
endtimeRTree = clock();
std::cout << "Running time of RTree construction: " << endtimeRTree - starttime << std
for (int i = 0; i < interior_MAT.size(); i++)
{
std::vector<value> result_s;
seg query_seg(point(viewpoint[0], viewpoint[1], viewpoint[2]), point(interior_MAT[i][0
rtree.query(bgi::intersects(query_seg), std::back_inserter(result_s));

if (result_s.size() == 0)
std::cout << "no box intersection!" << "ID" << i << std::endl;
if (result_s.size() == 1)
std::cout << "only intersect with target pt itself!" << i << std::endl;

//int result_id = result_s[0].second;
int result_id = i;
float min_dis = PointToPointDis(viewpoint, Vector3D(interior_MAT[i][0], interior_MAT[i

for (auto item : result_s)
{
int id = item.second;
if (DistancePointToSegment(viewpoint, Vector3D(interior_MAT[i][0], interior_MAT[i][1],
{
float current_dis = PointToPointDis(viewpoint, Vector3D(interior_MAT[id][0], interior_M
if (current_dis < min_dis)
{
result_id = id;
min_dis = current_dis;
}
}
}
//vec_result_id.push_back(result_id);
vec_result_id2.insert(in_idx[result_id]);
}

//std::cout << "Size of vector result id: " << vec_result_id.size() << std::endl;
```

```
for (auto id : vec_result_id2)
{
if (id > pc.size())
{
id = id - pc.size();
//std::cout << "index out of range!" << std::endl;
}
visible_pc.push_back(pc[id]);
}

//std::cout << "Size of set: " << vec_result_id2.size() << std::endl;
std::cout << "Size of visible pc: " << visible_pc.size() << std::endl;
endtime = clock();
std::cout << "running time:" << endtime - starttime <<"ms"<< std::endl;
output("visible_pc").set(visible_pc);


}
void VisiblePC::process()
{
std::cout << "Visible PC query starts" << std::endl;
std::cout << "Using KD-tree:" << if_checkbox<<std::endl ;

clock_t starttime, endtime;
starttime = clock();
//--------------input-----------------//
Vector3D viewpoint = input("viewPoint").get<Vector3D>();
auto kd = input("KDTree").get<KdTree*>();
auto interior_MAT = input("interior_MAT").get<PointCollection>();
auto radii = input("interior_radii").get<vec1f>();


auto pc = input("original_pc").get<PointCollection>();
std::cout << "input pc size:" << pc.size() << std::endl;
//--------------output-----------------//
PointCollection visible_pc;
// ----------process ---------------------//


/*int vetsize = pc.size();
std::vector<arr3f> threadvec1;
std::vector<arr3f> threadvec2;
std::vector<arr3f> threadvec3;
std::vector<arr3f> threadvec4;


std::for_each(begin(pc), begin(pc) + 0.25*vetsize, [&threadvec1](arr3f x) {
threadvec1.push_back(x);
});
std::for_each(begin(pc) + 0.25*vetsize, begin(pc) + 0.5*vetsize, [&threadvec
threadvec2.push_back(y);
});
std::for_each(begin(pc) + 0.5*vetsize, begin(pc) + 0.75*vetsize, [&threadvec
threadvec3.push_back(z);
});
```

```
std::for_each(begin(pc) + 0.75*vetsize, end(pc), [&threadvec4](arr3f v) {
threadvec4.push_back(v);
});

std::thread th[4];

th[0] = std::thread(VisiblePC::GetVisblePT, threadvec1, interior_MAT, radii, viewpoint
th[0].join();
th[1] = std::thread(VisiblePC::GetVisblePT, threadvec2, interior_MAT, radii, viewpoint
th[1].join();
th[2] = std::thread(VisiblePC::GetVisblePT, threadvec3, interior_MAT, radii, viewpoint
th[2].join();
th[3] = std::thread(VisiblePC::GetVisblePT, threadvec4, interior_MAT, radii, viewpoint
th[3].join();
*/

if (if_checkbox == true)
{
GetVisblePT(pc, kd, viewpoint, visible_pc);
}
else
{
GetVisblePTWithoutKD(pc, viewpoint, interior_MAT,radii,visible_pc);
}
endtime = clock();

//outfile.close();
//------------------set result -------------------//
std::cout << "visible_pc_done" << std::endl;
std::cout << "visible_points_size:" << visible_pc.size() << std::endl;
std::cout << "running_time:" << endtime - starttime << std::endl;

output("visible_pc").set(visible_pc);

}
void ParallelVector::process()
{
std::cout << "sight_vectors_starts" << std::endl;
//-------------input ------------------------//
Vector3D v1 = input("vector1").get<Vector3D>();
Vector3D v2 = input("vector2").get<Vector3D>();

float density = param<float>("Density");
float radius = param<float>("Radius");


//-------------output-------------------//
std::vector<Vector3D> Headvectors;
std::vector<Vector3D> Endvectors;

//---------------process-------------------//
Vector3D normal = (v2 - v1);
Vector3D perpen_normal(0, normal[2], -normal[1]);
auto u = perpen_normal.normalize();
std::cout << "Normalization:" << perpen_normal[0] << "," << perpen_normal[1] << "," <<
Vector3D v = crossProduct(perpen_normal, normal);
```

```
auto length = v.normalize();
//std::cout << "v:" << v[0] << "," << v[1] << "," << v[2] << std::endl;
// change radius and N to dynamic later
//float radius = 200;
//float density = 100;

float delta = radius / density;
float epsilon = delta * 0.5f;
//std::cout << "Test" << std::endl;

for (float y = -radius; y < radius + epsilon; y += delta)
{
for (float x = -radius; x < radius + epsilon; x += delta)
{
Headvectors.push_back(v1 + x * perpen_normal + y * v); // v1 is the point on
Endvectors.push_back(v2 + x * perpen_normal + y * v);
}
}
//std::cout << "Test done" << std::endl;

std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\Vectors_h
std::ofstream outfileVec(filepath, std::fstream::out | std::fstream::trunc);

for (int i = 0; i < Headvectors.size(); i++)
{
outfileVec << Headvectors[i][0] << "," << Headvectors[i][1] << "," << Headv
}
outfileVec.close();

output("Headvectors").set(Headvectors);
output("Endvectors").set(Endvectors);
std::cout << "number of parallel rays:" << Headvectors.size() << std::endl;
std::cout << "sight vectors done" << std::endl;
}
void VisiblePart::process()
{
std::cout << "Trying to get the visible part" << std::endl;
//-------------input ---------------- //
Vector3D viewpoint = input("viewpoint").get<Vector3D>();
auto targetPC = input("targetPC").get<PointCollection>();
auto kd = input("KDTree").get<KdTree*>();
auto interior_mat = input("MATpoints").get<PointCollection>();
auto radii = input("radii").get<vec1f>();
// ------------ output ----------------//
PointCollection vis_PC;
// ------------process ----------------//
for (auto pc : targetPC)
{
Vector3D v2(pc[0], pc[1], pc[2]);
Vector3D hit;

//------------Chech each point visibility ---------//
bool inter = GetRaysResult::CheckLineBox((*kd).m_min, (*kd).m_max, viewpoint
if (inter==0)
{
vis_PC.push_back({v2[0],v2[1],v2[2]});
```

```
continue;
}
else
{
bool ifblock = 0;
for (auto ball : (*kd).m_allballs)                {

float dis = VisibiltyQurey::DistanceOfPointToLine(viewpoint, v2, ball.pos);
if (dis <= ball.radius) {
ifblock = 1;
break;
}
}
if (ifblock == 1)
{
continue;
}
vis_PC.push_back({ v2[0],v2[1],v2[2]});
}
}




output("visible_parts").set(vis_PC);
std::cout << "Visible part is done" << std::endl;
}

void VisibiltyQurey::process() {
std::cout << "Visibility Qurey starts" << std::endl;
//------------------input-----------------//
Vector3D viewpoint = input("ViewPoint").get<Vector3D>();
auto kdtree = input("KDTree").get<KdTree*>();
auto interior_mat = input("interior_MAT").get<PointCollection>();
auto interior_radii = input("interior_radii").get<vec1f>();
auto original_pc = input("original_pc").get<PointCollection>();
//------------output----------------------//
PointCollection visible_mat;

vec1f visible_radii;
vec1i visible_index;
// ------------process -------------//

//std::string filepath = "c:\\users\\tengw\\documents\\git\\Results\\Visible_MAT_out.
//std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);


for (int i = 0; i < interior_mat.size(); i++) {
bool visibility = true;
Vector3D target = { interior_mat[i][0], interior_mat[i][1],interior_mat[i][2] };
(*kdtree).queryLineIntersection(viewpoint, target, 10, 1, 1);
int number = (*kdtree).getNOfFoundNeighbours();
for (int j = 0; j < number; j++)
{
int index = (*kdtree).getNeighbourPositionIndex(j);
```

```
Vector3D s = { interior_mat[index][0],interior_mat[index][1],interior_mat[in
float dis = VisibiltyQurey::DistanceOfPointToLine(viewpoint, target, s);
if (dis <= interior_radii[index])
{
visibility = false;
break;
}
}
if (visibility == false)
{
continue;
}

visible_mat.push_back({ interior_mat[i][0], interior_mat[i][1],interior_mat[
visible_radii.push_back(interior_radii[i]);

//outfile << interior_mat[i][0] << "," << interior_mat[i][1] << "," << inter
}
//outfile.close();
output("Visible_MAT").set(visible_mat);
output("Radii_of_MAT").set(visible_radii);
//output("indices").set(visible_index);
std::cout << "Visiblity Qurey Done" << std::endl;
}

void WritePC2File::process()
{
auto pc = input("points").get<PointCollection>();
auto filepath = param<std::string>("filepath");
//std::string filepath = "c:\\users\\tengw\\documents\\git\\Reorien\\PC_poin
std::ofstream outfile(filepath, std::fstream::out | std::fstream::trunc);
for (auto point : pc)
{
outfile << point[0] << "," << point[1] << "," << point[2] << std::endl;
}
outfile.close();
}
void ReadNormal::process()
{
//----------input-------//
auto filepath = param<std::string>("filepath");
//----------output----------//
vec3f normal;
// ----------proccess--------//
std::ifstream infile(filepath);
std::string line;
if (infile)
{
while (getline(infile, line))
{
std::vector<std::string> result;
ReadNormal::split(line, ',', result);
float x = ReadNormal::str2num(result[0]);
float y = ReadNormal::str2num(result[1]);
float z = ReadNormal::str2num(result[2]);
normal.push_back({ x,y,z });
```

```
}
}
infile.close();
std::cout <<"read normal size:" <<normal.size() << std::endl;
output("normal").set(normal);

}

void TreeRemover::process()
{
std::cout << "Removing trees..." << std::endl;
//----------------input----------------//
auto pc = input("points").get<PointCollection>();
auto classification = input("classification").get<std::vector<int>>();
int id = param<int>("number_value");

//----------------output----------------//
PointCollection outpc;
//----------process----------------//

std::set<int> s;
for (int i=0;i<pc.size();i++)
{
s.insert(classification[i]);
if (classification[i] != id) outpc.push_back(pc[i]);
}
for (set<int>::iterator it = s.begin(); it != s.end(); it++)
{
std::cout << *it << " occurs " << std::endl;
}

output("points").set(outpc);
std::cout << "Tree removing done." << std::endl;
}
}
```