Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Textured Surfels Visualization of Multi-Frame Point Cloud Data

David Schütz

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Thomas Ertl
Supervisor:	Patrick Gralka, M.Sc. Tobias Rau, M.Sc. Dr. Guido Reina
Commenced:	June 2, 2020
Completed:	January 4, 2021

Abstract

Laser scanning devices enable the capturing and storage of scenes and objects in the real world. The corresponding point datasets approximate surfaces where the laser beam of the scanner was reflected. Already today, there is a wide field of applications that visualize point clouds, ranging from preservation of cultural heritages to land surveying. With advances in technology, measuring techniques became increasingly precise and accurate. The associated growth of the captured point clouds offers many opportunities but also challenges. Visualizing billions of points can easily surpass the memory capacities of commercial computers. Furthermore, render times can increase, hindering the dynamic exploration and analysis.

In this work, we present a processing pipeline that reduces the complexity of point clouds. As a first step, planar regions of a point cloud are approximated by rectangles. Subsequently, calculated textures and displacement maps allow a more realistic representation of the classified points through rectangles. The generated data is stored in a data structure that supports different level of detail representations based on the camera position, as well as efficient culling of invisible points. Lastly, the data structure is rendered using the OSPRay render engine.

In an evaluation, different parameter configurations of our processing pipeline are examined. The collected data is analyzed in terms of render time, memory consumption, and image quality. Furthermore, all results are compared with a sphere-based reference method. Overall, a notable saving in memory consumption can be observed. Meanwhile, image quality and rendering times provide comparable results to the reference method, especially for distant to medium viewing distances. However, the achieved compression rate is dependent on the spatial properties of the point cloud. Datasets with large planar regions allow the consolidations of a high amount of points by a small number of rectangles. Meanwhile, regions with high curvature can cause overlapping geometry and a lower point reduction.

All in all, the presented approach enables the rendering of point clouds at different levels of detail. Approximated geometric shapes reduce memory consumption while preserving render times and image quality. Thus, complex point clouds can be visualized more efficiently on commercial computer systems.

Kurzfassung

Lasermessgeräte ermöglichen das Aufnehmen und Speichern von Szenen und Objekten in der realen Welt. Die zugehörigen Punktdatensätze approximieren Oberflächen, an welchen der Laserstrahl des Scanners reflektiert wurde. Bereits heute gibt es einen weiten Bereich an Anwendungen, welcher von der Erhaltung von kulturellem Erbe bis zur Landvermessung reicht. Mit technologischen Fortschritten wurden die Messtechniken immer präziser. Das verbundene Wachstum an Größe der aufgenommenen Punktwolken bietet viele Möglichkeiten, aber auch Herausforderungen. Das Visualisieren von Milliarden an Punkten kann schnell die Speicherkapazitäten von kommerziellen Computern übersteigen. Darüber hinaus kann die Renderzeit steigen, was die dynamische Exploration und Analyse erschwert.

In dieser Arbeit präsentieren wir eine Verarbeitungspipeline, welche die Komplexität einer Punktwolke reduziert. In einem ersten Schritt werden planare Regionen in einer Punktwolke durch Rechtecke approximiert. Anschließend ermöglichen berechnete Texturen und Displacement Maps eine realistischere Repräsentation der klassifizierten Punkte durch Rechtecke. Die erzeugten Daten werden dabei in einer Datenstruktur gespeichert, welche verschiedene Detailgrade basierend auf der Kameradistanz unterstützt, sowie auch nicht sichtbare Punkte effizient entfernt. Zuletzt wird die Datenstruktur in der OSPRay Engine gerendert.

In einer Evaluation werden verschiedene Parameterkonfigurationen unserer Verarbeitungspipeline untersucht. Die gesammelten Daten werden auf Renderzeiten, Speicherverbrauch und Bildqualität analysiert. Darüber hinaus werden alle Ergebnisse mit einer auf Sphären basierenden Referenzmethode verglichen. Insgesamt können beachtenswerte Einsparung im Speicherverbrauch ausgemacht werden. Währenddessen liefern die Resultate für Bildqualität und Renderzeiten, besonders für entfernt und mittlere Betrachtungsabstände, vergleichbare Ergebnisse mit der Referenzmethode. Jedoch ist die erreichte Kompressionsrate von den räumlichen Eigenschaften der Punktwolke abhängig. Datensätze mit großen planaren Regionen ermöglichen das Zusammenfassen von großen Punktmengen durch eine geringe Anzahl an Vierecken. Währenddessen können Regionen mit hoher Krümmung überlappende Geometrie und eine geringere Reduktion an Punkten verursachen.

Alles in allem ermöglicht der präsentierte Ansatz das Rendern der Punktwolken in verschiedenen Detailstufen. Die approximierten geometrischen Formen reduzieren den Speicherverbrauch, während Renderzeiten und Bildqualität bewahrt werden. Dadurch können auch komplexe Punktwolken auf kommerziellen Computern effizient dargestellt werden.

Contents

List of Figures

List of Tables

1 Introduction

Rendering large particle datasets is an important and highly studied field in computer visualization. Finding applications across and beyond scientific fields, point datasets can be acquired in different ways like per simulations or measuring devices. Common scanning devices that capture real-world scenes can be based on high-frequency light pulses. These laser scanners gather datasets containing point data. To do so, they scan objects in the real world with a high precision and store the depth position of each point. The resulting set of points approximates the surfaces of the observed environment. Merging multiple of these datasets from different angles and viewing distances results in an increasingly complex and dense point cloud. This opens up a wide range of use cases, whether it is the preservation of historical monuments, support in crime scene analysis, or quality control of technical components.

Visualizing a point dataset bears many challenges. Modern scanning devices can capture large point clouds including up to billions of points. Besides the three dimensional position, each point can contain additional information like color or velocity depending on the application area. This results in large datasets often surpassing the memory capacity of a graphics card or even the random-access memory. Furthermore, often the sheer computational power is too small to display billions of points simultaneously. Therefore, simply rendering each point as a sphere or point primitive quickly reaches limitations based on the size of the point cloud. The resulting high render times lead to a more cumbersome dynamic exploration of the point data. In addition to technical requirements, visualizing the whole point cloud has less impact on the resulting image quality as the camera distance increases. If more than one point is projected to a pixel, the user can no longer distinguish between the individual points. Since modern scanning devices achieve a high point frequency, measuring points less than a millimeter away from each other. This can result in thousands of points being summarized in one pixel. Additionally, when being zoomed in, large areas of the point cloud can be outside of the viewport and therefore not visible to the user. Varying point density and noise hide additional challenges in rendering and analyzing the data. Thus, preprocessing and simplifying the dataset is a necessary task, enabling the visualization of large point clouds on commercially available computer systems.

Traditional rendering approaches rely on a subsampling of the point cloud or the reconstruction of the point data to a triangulated mesh. Furthermore, spatial data structures can be used to accelerate the occlusion of invisible points. However, finding representatives reducing processing time and memory requirements while preserving image quality remains a challenging task.

In this work, a different hybrid approach will be explored. Rendering point cloud data via surface elements so-called surfels. This is achieved by summarizing sets of points through geometry elements. Here, a point cloud segmentation method enables the detection of planar surfaces in the point cloud. Subsequently, for each classified plane, a rectangle is approximated in the point cloud. To enable a more realistic representation, the color data of the classified points are combined into

textures. Furthermore, a level of detail is provided by displacement maps which incorporate the deviations of points from the approximated planes. Therefore, depending on the viewing distance, different level of detail representations can be rendered using the introduced data structure.

In the course of this work, we collaborated with the company FARO, in particular with Dr. Sebastian Grottel. Dr. Grottel provided various datasets for the evaluation as well as expert opinion.

Thesis Structure

The upcoming chapters are structured as follows:

Chapter 2 – Foundations: This chapter provides technical fundamentals on which the upcoming chapters are based on. Spatial data structures allowing the storage of point and geometry data are introduced and explained. Additionally, the basics and methods for an efficient finding of adjacent points are introduced. Finally, existing segmentation methods for classifying points are presented. These methods are later used to find point representatives as well as to further spatially classify point data.

Chapter 3 – Related Work: Existing approaches that allow the efficient visualization of point clouds are presented. Additionally, details on their procedure and implementations are provided. Furthermore, we outline the differences and distinctions to our proposed approach.

Chapter 4 – Methodology: An overview of our approach is introduced based on previously made observations. The basic steps and foundations of our approach are explained, leading from a point cloud to a data structure, allowing the dynamic visualization of the processed data.

Chapter 5 – Implementation: In this chapter, the implementation of the previously presented approach is explained. Algorithms as well as used technologies are illustrated in more detail.

Chapter 6 – Evaluation: To compare different preprocessing configurations the implemented approach is executed and measured for a predefined parameter space. Runtime, memory consumption, and image quality are examined. Subsequently, the different configurations are compared to each other as well as to a reference method.

Chapter 7 – Discussion: We examine and analyze the previously presented results. The advantages and disadvantages as well as the limitations of our approach are outlined.

Chapter 8 – Conclusion and Outlook: In this chapter, the fundamental steps, as well as the most important obtained results and information of this work are summarized. In a subsequent outlook, possible extensions and improvements are presented.

2 Foundations

This chapter presents methods and topics on which this work is based on. First spatial data structures like Bounding Volume Hierarchies and kd-trees are explained. The subsequently presented pkd-tree specializes in the efficient storage and processing of particle datasets. Afterward, clustering algorithms are explained, which segment a point cloud in different groups based on point properties. Last, we illustrate the Principal Component Analysis, which enables the dimensional reduction of a point dataset and finds Principal Component vectors representing directions with the highest variance.

2.1 Bounding Volume Hierarchy

A Bounding Volume Hierarchy (BVH) is a tree structure containing bounding boxes of objects in space. BVHs are used for a wide variety of tasks. Clark [Cla76] proposed to use a BVH to speed up processing times when rendering objects in space from an arbitrary viewpoint. Usually, leaf nodes of a BVH contain geometric primitives. Those leaf nodes are grouped and summarized by parent nodes with greater box values bounding the bounding boxes of all the child nodes. A popular use case is the acceleration of ray primitive intersections in ray tracing algorithms.

Different strategies to construct a BVH-tree exist. Most notable top-down, bottom-up, and insertion methods. Top-down algorithms [KK86] start with the root node of the tree and then split up the bounding boxes until all primitives in the leaf nodes are reached. Bottom-up approaches [GHFB13; WBKP08] start with all primitives and subsequently merge them until the root node is constructed. Often a hierarchical clustering method is used in these approaches to guarantee a better tree quality. Lastly, insertion methods [BHH14; GS87] create a BVH incrementally, inserting one node after another into the tree. This has the advantage that not all primitives need to be stored in memory at the same time.

Finding good partitions is an important task when building a BVH-tree. Inadequate partitioning metrics can cause unnecessarily many intersection tests caused by overlapping bounding boxes. A simple approach is to partition along the axis with the longest range between two centroids of all bounding boxes [KK86]. The surface area heuristic [GS87] is a popular partitioning scheme optimizing the tree for ray intersection tests. The heuristic minimizes the area of regions with a dense vertex distribution. Therefore, the probability of a random ray hitting this area is reduced avoiding further costly intersection tests with many nodes. Originally the proposed concept of Goldsmith and Salmon [GS87] created a BVH based on an insertion strategy. However, many state-of-the-art BVH construction methods use SAH together with a top-down approach [MB90; Wal07].

Traversing a BVH starts at the root node where a ray or another object is intersected with the bounding box of the node. If the ray hits the bounding box, the children of the nodes are traversed and the intersection is tested again. When the ray misses the bounding box the node and all its children can be neglected. Therefore, the tree traversal ends when either a leaf node is hit or all intersection tests miss. Since bounding boxes can overlap, the tree is traversed on multiple paths. All paths have to be processed in the algorithm until termination as there is always a possibility of finding a better fit.

2.2 Kd-Tree

Being invented in 1975 by Bentley [Ben75], the kd-tree is a multidimensional binary search tree. The spatial data structure finds use in nearest neighbor search or as an acceleration structure in computer graphics among other fields. In short, a kd-tree is a binary space partitioning (BSP) tree [FKN80] recursively dividing space into two convex subsets. In addition to a BSP tree, a kd-tree requires the intersecting hyperplanes subdividing the data to be aligned to the axes of the coordinate system.

Every level of a kd-tree splits the data along a cutting dimension. Bentley et al. proposed to repeat each dimension equally in a round-robin order. For this all nodes of the tree store one data point representing the split in the coordinate of the cut dimension. An illustration of a kd-tree is shown in Figure 2.1 a). A kd-tree is explored recursively. Traversing to the left child means finding data points with smaller values in the dividing dimension. Traversing to the right results in higher values.

2.3 Pkd-Tree

The pkd-tree, which was introduced by Wald et al. [WKJ+15], is a data structure based on the kd-tree supporting the rendering of large particle datasets. In their work Wald et al. distinguish between balanced and spatial kd-trees. Spatial kd-tree splits are determined through space usually dividing the tree into spatially equal subtrees. However, the resulting tree is oftentimes unbalanced which represents a disadvantage of this method. Balanced kd-trees divide the dataset into equal-sized subsets. This results in a tree being completely balanced except for its leaf nodes. Subsequently, the kd-tree can be stored without any additional memory overhead by simply sorting the data points. However, the tree can not be adapted to store other geometric primitives than spheres or points. Similar to a BVH, the kd-tree can be build using the surface area heuristic, optimizing ray primitives intersections.

Pkd-trees are based on balanced kd-trees. In contrary to the original algorithm from Bentley [Ben75], the respective dimensions splitting the tree are not chosen in a round-robin procedure. Instead, the dimension with the maximum extent of the bounding box is divided. Traversing a pkd-tree is alike to traversing a BVH-tree. Given two subtrees one can compute their bounding boxes by clipping the bounding box of the parent node at the intersecting line of the node. However, the radius of the particle stored in a node has to be considered. Therefore, a maximum radius is added to both bounding boxes. This results in two conservative bounding boxes overlapping each other at the center of the node's particle by a length of two times the maximum radius.

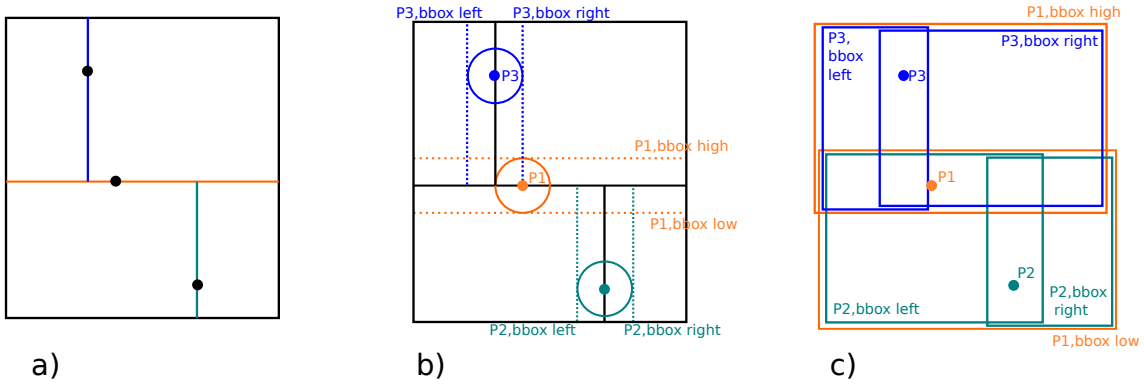

Figure 2.1: Particles stored in two different spatial structures. Image a) shows the spatial segmentation of a kd-tree. Here, the data structure is divided at splitting points, which are stored in the respective node. The points divide space parallel to the coordinate axes in for example a round-robin schedule. A pkd-tree in b) splits the data along the coordinates of single points. Additionally, a threshold value is added creating two conservative bounding boxes overlapping in the middle, as can be seen in c). Note that for presentation purposes a small epsilon value is added to the shown bounding boxes preventing visual overlap. Figures b) and c) are adapted from Wald et al. [WKJ+15].

Additionally, nodes contain exactly one splitting point. Correspondingly, the tree can be easily recursively traversed by testing the bounding boxes of both child nodes. An example of the maximum radius threshold can be seen in Figure 2.1 b). The associated conservative bounding boxes are shown in Figure 2.1 c). When the left bounding box is hit, the left child is traversed, or alternatively the right child when its bounding box is intersected. If the tested value lies in the area where both bounding boxes overlap, both children have to be traversed.

In practice, several adaptations can be done to optimize tree traversal, especially regarding ray intersections. First, the pkd-tree can be traversed in a front to back order allowing early termination when an intersection is found. However, it has to be taken into account, that the pkd-tree can have overlapping bounding boxes. Therefore, all sub-trees being recursively traversed at the same time need to continue their traversal, since they may find a closer intersection. In addition, computing the ray intersection with the whole bounding box each time results in unnecessary calculations. When moving from a node to its child only one of the six sides of the bounding box actually changed. Hence, only this side has to be ray intersection tested again.

In an evaluation Wald et al. [WKJ+15] compared the pkd-tree with an Embree quad-BVH-tree. Especially in memory requirements, the pkd-tree surpasses the BVH-tree since no additional space for pointers is needed. Therefore, the pkd-tree requires more than five times less memory. In contrast, the BVH-tree is 1.48 times faster, especially for close-up views, whereas this advantage is negated for farther viewing distances.

2.4 Nearest Neighbor Search

The nearest neighbor search finds all surrounding points of a query point in a dataset. There are different types of nearest neighbor search, changing search extent, and the size of the results. Elseberg et al. [EMSN12] mentions three different types of nearest neighbor queries.

K-nearest neighbor (k-nn) search, described by Cover and Hart [CH67] finds a number of k points that are closest to the query point. Especially in cases when an exact number of points is needed, k-nn search is of advantage. However, if applied in regions with sparse points, the k-nn search can lead to unwanted results. This is caused by far away points, which are included since the query point does not have many neighbors. Fixed radius nearest neighbor search solves this problem by returning all nearest neighbors lying in a fixed radius around the query point. With this method, the distance between points can not surpass a given threshold. Nevertheless, the number of found points is not fixed anymore. Sparse regions may result in little to no points, while dense regions can return many points, resulting in higher computation costs. The third query type, which is called ranged search combines a fixed radius nearest neighbor search with a k-nn search. Here, a number of k neighbors are searched in a region limited by a radius. However, according to Elseberg et al. [EMSN12] many nearest neighbors searching libraries do not provide this search strategy. Another notable query type is the approximate nearest neighbor search. This approach, as presented by Arya et al. [AMN+98], trades accuracy for efficiency. While approximate nearest neighbor search may not find all nearest neighbors of a query point, it exceeds other approaches in computation speed and memory savings.

Besides different search strategies, the nearest neighbor search can be implemented using various data structures. Most approaches use spatial trees like a kd-tree or octree. Meanwhile, the approximate nearest neighbor search relies on local sensitive hashing.

2.5 Principal Component Analysis

According to Jolliffe and Cadima [JC16], Principal Component Analysis (PCA) is a widespread and one of the oldest dimension reduction methods. Given a k-dimensional dataset PCA can reduce the dimensionality by projecting them to Principal Components. Principal Components are linear vectors maximizing the variance of the data in their direction. This is done by fitting a hyperplane onto the dataset that approximates the position the data points. Besides dimension reduction PCA additionally finds use in other tasks like approximating normal vectors, finding an eigenbasis, and describing the planarity of points.

Since PCA is sensitive to variances between different measurement units it is common to standardize each data point at the beginning. This is done by transforming the dataset to the center and dividing each value by its respective standard deviation. In the second step, the covariance matrix is computed. The covariance matrix is a symmetric squared matrix describing the variance and covariance between each variable. On the diagonal of the matrix, the variance along each axis of the datasets is described. Other values are the covariance between the two linear combinations. Therefore, the covariance indicates the correlation between two variables.

Lastly, eigenvalues and eigenvectors of the covariance matrix are calculated. Since the covariance matrix describes the variance between each variable, the eigenvector with the largest eigenvalue maximizes the variance of the dataset along its direction. Note, since the covariance matrix is symmetric all eigenvectors of the matrix are orthogonal. Thus, they can build a new orthogonal eigenbasis reducing the dimensionality of the datasets by a chosen number of principal components.

2.6 Point Cloud Segmentation

Before finding use in 3D point data many segmentation methods have their origin in 2D image processing. Image segmentation is defined as ”a process of partitioning pixels of an image to different regions based on specific information, which are normally intensity, texture or color” Liang et al. [LZB14, p. 847]. The same principle can be applied to 3D point data. Points in 3D space are clustered in different local regions sharing the same attributes. However, points do not lie equally spaced on a grid. Contrary to images, points in a point cloud can have varying distances to their neighbors, building dense and sparse regions. Thus, other attributes like the normal or curvature at each point gain more importance for the segmentation of 3D data.

Point cloud segmentation methods can be categorized into different groups. Nguyen and Le [NL13] distinguish between edge-based [BLHH; WA03], region-based [NBW12; VTLB15], attributes-based [BL08; VD01], model-based [GG04; SWK07], and graph-based methods [GF09; SRO10].

Edge-based methods detect edges splitting the point cloud data into different regions. Then areas lying in between these edges are classified. By searching the neighborhood and including points with similar attributes, region-based algorithms find clusters in a dataset. More precisely, a distinction is made between seeded and unseeded region growing methods. Seeded-region approaches start at a seed point, where a local region incrementally grows until all bounding points are found. Afterward, the process is repeated for the next seed point if the point has not already been classified. Unseeded methods use a top-down approach by first classifying all points to the same cluster and then dividing the region into smaller ones. In the following sections, a seeded region growing algorithm is explained in more detail, since the algorithm to find planar regions in this work is based on it.

Attribute-based methods use clustering algorithms based on precomputed attributes for each point. Meanwhile, graph-based methods arrange data points in a graph, with for example neighboring points sharing an edge. Subsequently, based on different graph operations cluster can be identified. At last, model-based algorithms try to fit simple mathematical shapes matching the point cloud. In a following step, particles are classified as shapes.

Furthermore, Xie et al. [XTZ20] distinguish between Point Cloud Segmentation algorithms, like the above-mentioned methods, and Point Cloud Semantic Segmentation. Point Cloud Semantic Segmentation is a topic of 3D Point Cloud segmentation growing popularity over the last years. Here, additional semantic information for each point, usually using machine learning approaches, are detected and stored. Works like PointNet [QSMG17] can classify and segment semantic structures like mugs, tables and, chairs in a 3D point cloud using a deep learning approach.

2.7 Region Growing

In "Robust Segmentation in Laser Scanning 3D Noisy Point Cloud Data", Nurunnabi et al. [NBW12] present a seeded region growing algorithm. Notable compared to other algorithms is the seed point selection based on their curvature and the comparison between PCA and robust PCA [HRB05] for normal and curvature calculation. In addition, region growing is applied based on three different criteria. These criteria are the orthogonal distance, euclidean distance, and the angle between normal vectors.

In the first step, the curvature and normal vector are computed for each point in the point cloud. This is done by either using PCA or robust PCA. When calculating the covariance matrix on a set of nearest neighbor points, a 3D plane is fitted into this subset of data. Here, the two eigenvectors with the biggest eigenvalues point in the two directions of the highest variance, building two perpendicular tension vectors of the plane. Meanwhile, the third eigenvector points in the direction of the horizontal variance of the fitted plane. Therefore, the third eigenvector can be used as an approximation of the normal vector. Depending on the used query type for the nearest neighbor search the fitted plane is averaged over more or fewer data points. Nurunnabi et al. use k-nn search for their normal estimation, arguing that their datasets have varying point density. The planarity or also called curvature σ of a point p_i is calculated as followed: $\sigma(p_i) = \frac{\lambda_0}{\lambda_0+\lambda_1+\lambda_2}$ [PGK02]. Here λ_0 is the smallest of the three eigenvalues. If λ_0 has a high value it implies a high variance towards the direction of the normal. This in turn indicates an uneven surface.

In a following step, region growing is performed on the point cloud. For this, the algorithm starts at the point with the least curvature. This point is pushed into a set of seed points, which is empty at the beginning. In the next step, the first point in the set of seed points is removed. If the curvature value of the point surpasses a fixed threshold value and the point is not classified yet, a k-nn search is performed. For each nearest neighbor point, three criteria are evaluated. If the criteria are met, the point is classified to the current cluster. To reduce noise and outlier data, the orthogonal distance (OD) of all neighboring points p_i is calculated by: $\mathrm{OD}(p_i) = (p_i - \overline{p})^T . \hat{n}$ with $\overline{p}$ being the median point and $\hat{n}$ being the unit normal of the fitted plane. For the second criteria, the euclidean distance from the current point to the seed point is calculated. The threshold distance is the median of the distance between the seed point and all its neighbors. At last, the angle θ between the seed point normal n_i and current point normal n_j is tested against a given threshold. The angle is calculated as follows: $\theta_{ij} = \arccos\left|\hat{n_i}^T \hat{n_j}\right|$. If all requirements are met the point p_i is pushed into the set of seed points. Afterward, nearest neighbor search is performed on the next point in the set of seed points. This process is repeated until the set of seed points is empty. If the number of all points classified to the current cluster exceeds a minimum quantity, the current cluster is accepted. Otherwise, the seed points are classified as noise. Subsequently, the next unclassified point with the lowest curvature is pushed into the empty set of seed points and the procedure is repeated for the next cluster until all points are classified.

In an evaluation, Nurunnabi et al. show that using robust principal component analysis makes the algorithm more robust to outliers. Therefore, robust component analysis delivers more correct results for noisy data.

2.8 DBScan

Clustering techniques find patterns in datasets through unsupervised learning and classify their data elements. DBScan scan is such a clustering algorithm and was first introduced by Ester et al. [EKSX96]. The algorithm is based on the density of data points. As opposed to other clustering algorithms like k-means [Mac67], there is no need to determine the number of clusters to be found. DBScan starts by iterating over a set of unclassified data points. Afterward, fixed radius nearest neighbor search is performed on the given points. If the number of found neighbors surpasses a designated threshold, a new cluster will be assigned to the point as well as all its neighbors. Subsequently, a nearest neighbor search is performed iteratively on the set of all unclassified found neighbors. Again, if the quantity of found neighbors of a seed surpasses the threshold all previously unclassified points join the cluster. Additionally, the points are included in the set of seed points. This procedure is repeated until no unclassified points are found. Adjustable parameters of DBScan are the minimum number of nearest neighbors found for a queued seed point and the radius which is used for the fixed radius nearest neighbor search.

3 Related Work

Visualizing point clouds is a well-studied topic in computer graphics. It covers many areas ranging from the visualization of molecular dynamics simulations to the reconstruction of surfaces. This work focuses on the rendering of large point clouds. In addition, the surface of a point cloud is approximated by geometrical shapes and simplified accordingly. In the following, the related work to our approach is presented for both fields. Furthermore, we argument how our work differs from the presented methods.

3.1 Rendering of Large Point Clouds

One of the first methods to render large point clouds is QSplat [RL00]. QSplat relies on a bounding spheres data structure. Firstly, the structure is built by preprocessing the point cloud to a triangular mesh. Afterward, normals for each vertex can be computed more easily. Furthermore, a maximum sphere size for a vertex is chosen based on the maximum distance to all connected vertices. To create a tree, vertices are split in a top-down approach along their longest axis. If only a single vertex is left, a sphere is spanned with the vertex being in its center. The bounding spheres data structure enables the use of visibility culling and level of detail control. When rendering, the algorithm traverses the created tree. If a node is not visible on the screen, the whole branch of the tree is skipped. Otherwise, all children of the tree are traversed until either a leaf node is met or the area of the sphere projected to the screen exceeds a set threshold. Subsequently, the spheres are visualized using a splat based rendering method. When the camera is not moved, the scene is rendered in successively higher detail, until each splat reaches the size of one pixel.

Equally to QSplat, Gobbetti and Marton [GM04] create a hierarchical data structure where each level of the data structure contains a further refinement of the previous tree level. Therefore, the closer a node is to the root node, the coarser is its representation. The data structure is based on a binary tree containing indices to a point array. Here, the root node points to the whole point cloud. Child nodes spatially subdivide the points of their parent node into two equally sized subsets. This process is executed until the nodes reach a predefined limit of points. Each node represents a subsampling of its contained points. Here, the root node contains the coarsest resolution while child nodes add a fixed amount of additional points to the representation. Cutting the tree at specific levels provides a corresponding level of detail depending on the cutting level.

Sequential point trees are a data structure allowing the adaptive rendering of point clouds solely on the graphics processing unit [DVS03]. In the first step, the point data is arranged in an octree. In a bottom-up approach, bounding disks are approximated for each node, representing the later splats used for rendering the node. Afterward, the tree can be traversed recursively. Based on different error metrics the deviation of all child disks projected to the bounding disk of the parent node can be calculated. When rendering the data structure, a level of detail is created, by defining an error

threshold. If the calculated error lies above the threshold all children of the node are traversed recursively. Afterward, each node meeting the requirements or being a leaf node is rendered as a splat. Since the rendering method is based on recursion it is not capable of being efficiently rendered on a graphics card. Therefore, the point tree is sequentialized and rearranged. A new more intuitive measure is calculated providing a camera distance interval at which the node meets the set error threshold. Subsequently, all nodes are stored in an array and are sorted by the maximum value in the computed interval. This array can be rendered on the GPU sequentially moving from coarse to fine level of detail.

Instant points [WS06] extend the concept of sequential point trees by reducing their memory overhead. Afterward, the so-called memory optimized sequential point trees (MOSPT) are combined with a nested octree. An outer octree allows view-frustum culling as well as multiple selectable levels of detail for the whole model. Each node of the outer octree contains a single internal octree. Meanwhile, each node of the internal octree contains a MOSPT. This data structure enables a splat based rendering approach visualizing large point clouds without the need for long preprocessing steps.

A more recent implementation using nested octrees is Potree [Sch16]. Potree is a web-based visualization tool rendering large point clouds. Each node of the octree subsamples its included point set. Thereby, higher-level nodes increase the number of points representing its spatial area. The subsampling strategy is based on Poisson-disk subsampling, which was first proposed by Cook [Coo86] in the context of computer graphics. This allows the rendering algorithm to calculate the level of detail on the fly based on the depth of the visible tree nodes. Point sizes of the rendered splats are determined adaptively adjusting the covered space between adjacent points.

Octree based rendering methods introduce some challenges when being rendered on a graphics card. Since they divide the data in equally spaced boxes each node can enclose an arbitrary amount of points. Therefore, dense regions lead to a higher maximum node level, while coarse regions provide little to no further refinement. This can lead to highly unbalanced trees. Furthermore, leaf nodes of an octree can contain a variable amount of points producing different vertex buffer object sizes. Goswami et al. [GZPG10] address these challenges by using a kd-tree data structure with a variable branching factor. Choosing this factor based on the size of the dataset makes it possible to pick a uniform amount of points per node and a favored tree depth. The sum of all leaf nodes equally subdivides the point data of the whole point cloud. In higher-level nodes points with similar normal directions are clustered and represented by additional calculated point representatives. Thereby each node contains the same amount of points similar to the point threshold of the leaf nodes.

Similar to the presented methods, we use a level of detail datastructure in our work. However, instead of subsampling points or calculating new point primitives for each level of detail, we utilize segmented geometric primitives as representatives. Subsequently, the datastructure has to be adjusted, handling various geometric shapes as well as unclassified point data.

3.2 Surface Reconstruction

With surface reconstruction, we summarize approaches that focus on the approximation of point cloud surfaces. Already in 1992 Hoppe et al. [HDD+92] addressed the topic of simplifying the surfaces described by points. Hoppe estimated plane normals in a k-neighborhood of nearest points

through PCA. Afterward, edges between points were weighted based on the normal deviation of both vertices. Based on these weights a minimal spanning tree was built. Finally, marching cubes was used to calculate a linear continuous approximation of the point surface.

Similar to Hoppe et al., Pauly et al. [PGK02] calculated normals through the eigenvector with the smallest eigenvalue in the covariance matrix. Additionally, a curvature value for each point based on the three eigenvalues was calculated. Subsequently, two methods to cluster point regions were presented. The first approach utilized a region growing algorithm to find point regions of a limited size to subdivide the point cloud in multiple patches. Another proposed method relied on adaptive hierarchical clustering splitting regions along their directions with the highest variance. Both methods generated clustered regions of a fixed maximum size. The higher the curvature within a region, the fewer points it contains. Finally, each detected region is approximated by a splat. Thereby the area of a splat increases with the number of represented points. Depending on the defined maximum amount of points for a patch, a different level of detail can be achieved, highly reducing the amount of rendered points.

There is a wide variety of different approaches to approximate the surface of a point cloud by a triangle mesh. Carr et al. [CBC+01] uses radial basis function interpolation. Here, a single radial basis function is fitted to the point cloud data. Subsequently, the surface can be either visualized via raytracing, or a marching tetrahedra variant can be used to approximate the isosurface of the function. Kazhdan et al. [KBH06] formulates the surface reconstruction as a spatial Poisson problem. All of these solutions assume that the point cloud describes a watertight surface. This means, that the resulting mesh consists of one closed surface without any holes.

Besides the previously presented smooth reconstruction algorithms, the surface of a point cloud can be approximated by fitting geometric primitives in the point cloud. As described in Section 2.6 there exists a wide range of algorithms to detect geometric primitives. The results of these algorithms can be used to represent point regions. Chen and Chen [CC08] use a method similar to region growing, clustering adjacent points with low normal deviation. Then boundary lines of a cluster are detected by projecting the point cloud in 2D space. Subsequently, boundary points are detected by computing distances from a point to all detected lines. In a final step, a polygon for each cluster is calculated. This method allows the reconstruction of sparse point clouds solely by polygons. Lafarge and Alliez [LA13] describe a hybrid algorithm consolidating detected planar shapes with unclassified point regions. For this, points representing planar shapes as well as sharp regions are resampled. Afterward, a Delaunay triangulation [Del34] for all vertices based on an error threshold is computed. The higher the threshold the more points are represented by planar shapes.

We use a point cloud segmentation method similarly to geometric primitive based surface reconstruction methods. However, our aim is to be able to handle larger point cloud sizes combining the segmentation with a level of detail datastructure. Additionally, the presented methods are only validated for point clouds containing less than three million points. Furthermore, many methods like Lafarge and Alliez [LA13] do not support color information in their presented approach. Finally, since many point clouds can contain holes and discontinuities, we do not assume water tight surfaces for our approach.

4 Methodology

This chapter outlines the methods and steps we used to process a point cloud. For this purpose, a visualized point cloud is observed more closely and performance bottlenecks are identified. Subsequently, a method is developed to circumvent the found challenges and to enable a more efficient visualization. Therefore, suitable regions are represented by detected geometric shapes, replacing point subsets. Afterward, a datastructure is designed allowing the efficient rendering of the resulting dataset based on the distance to the camera.

4.1 Observations

The visualization of point clouds underlies several restrictions in practice. These are mostly based on computer hardware limitations but also on human capabilities. At a certain distance, humans are no longer able to distinguish between two objects. This property is due to the physical conditions of the eye and differs between different persons and ages. In more detail, the angular resolution of the human eye is about one arc minute [Ver18]. This means that under ideal conditions a human can separate two lines which are at least in an angle of $\frac{1}{60^\circ}$. Thus, for a common viewing distance of 25 centimeters, the human eye can distinguish two lines that are at least 0.075 millimeters apart. For a viewing distance of one kilometer, the interval is already 0.03 meters apart. Therefore, for larger viewing distances the human eye can not separate adjacent points anymore.

Additionally, the used computer hardware especially the processing unit for rendering and the available memory cause restrictions. When the required memory of the point cloud surpasses the capacity of the random-access memory (RAM) or video random-access memory (VRAM), costly data transfers slow down the rendering speed significantly. Furthermore, the processing speed of the CPU or GPU is limited. The time to process billions of points can be not fast enough to produce an image in an acceptable time at 24 frames per seconds or higher. Therefore, reducing the number of rendered primitives can accelerate rendering times and diminish memory capabilities. Besides the limitations of the human eye, the resolution of the computer monitor restricts the number of points that can be distinguished in a given area. If two projected points are mapped into the same raster during the rasterization stage, both are mapped onto the same pixel. Hence, they can not be visually separated in the final rendered image.

To further analyze this, we implemented a simple point renderer coupled with a histogram. Given a point cloud dataset, the renderer draws every point as a point primitive. The size of each point is one pixel. Now, during the rasterization stage, each point falling into a pixel is counted. This results in a texture with the height and width of the program's resolution. Every texel in the texture represents a pixel and encodes the number of points falling into it. To visualize the texture every value is linearly scaled from zero to the maximum value in the texture. Then, the texture is rendered as a heatmap with a color scale from blue to yellow.

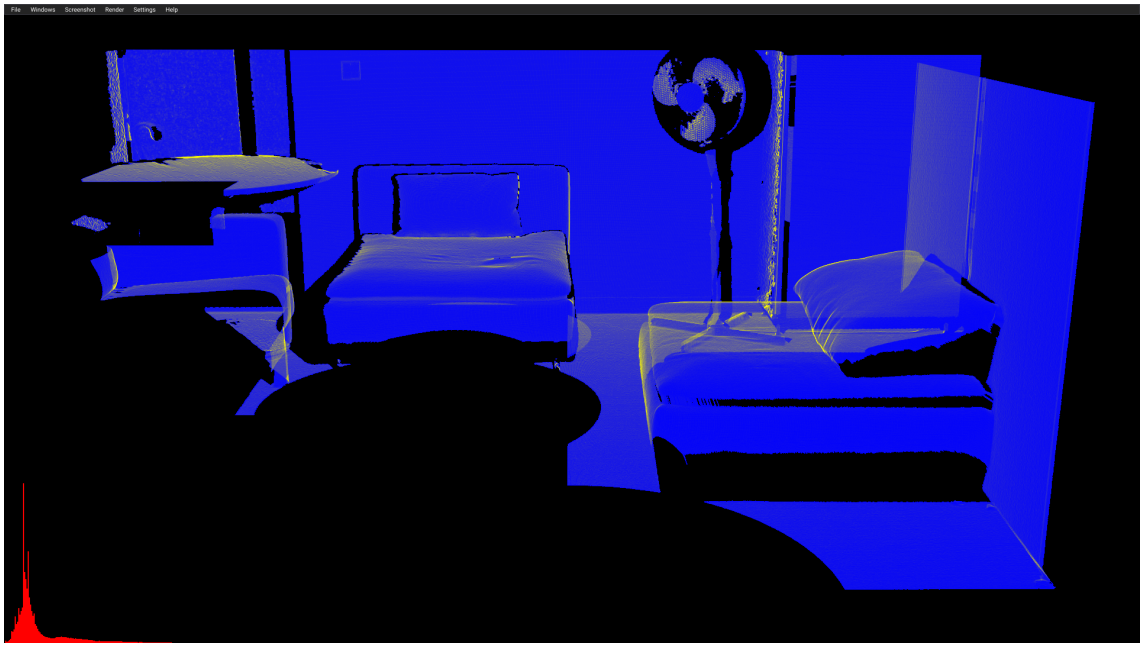

Figure 4.1: LiDAR dataset remodeling the interior of a room. A heatmap from blue to yellow encodes the number of points that are mapped to a single pixel. The values are linearly scaled from one point to 250 points per pixel. A histogram at the bottom further visualizes the point distribution among the pixels. With a number of 120131 most of the pixels contain 16 points. Planar areas perpendicular to the viewer build the densest regions.

A blue color encodes pixels with one or a few points in them. Yellow indicates a high amount of points with up to 250 points per pixel. Additionally, a histogram is generated. The x-values of the histogram are binned by the number of points falling into a pixel on the histogram. This means, all pixels with a certain range of points being mapped to them, are represented by one bar. The y-Axis encodes the number of pixels falling in the defined range of a bar. For example, if the bin size is one, the height of the most left bar would represent the number of pixels containing exactly one pixel. If the bin size is ten, the left bar encodes all pixels with one to ten points in them. Figure 4.1 shows the rendered picture using a point cloud acquired by a laser scanner. The dataset remodels the interiors of a room. Various observations can be made when analyzing the data.

A rather simple insight is that similar to the human eye, the number of points which can not be visually separated increase with the viewing distance. While standing exactly in front of the armchair, seen in the middle of the picture, most pixels contain one point. Moving half a meter away from the chair increases the mode to eight points per pixel. Finally, at one meter away most pixels include twenty points. From a far distance of 25 meters, 5000 points or more is the most frequent value, with a maximum value of twenty thousand points mapped to a pixel.

A second observation considers the planar regions of the room. When they are placed perpendicular to the viewer, many points are projected to the same pixel in the shape of a line. For example, if the surface normal of the seat pad points in the direction of the camera, all points representing the pad

are distributed along a large area of the screen. When rotated by ninety degrees, the area of the projected point subset diminishes. Solely a thin line is visible on the screen, with the vector of the highest variance of the planar subset pointing to the direction of the camera.

Last, hidden, invisible points increase the number of points per pixel. This can be seen on the right side of the image. Here, the chair should cover the stand of the fan and the floor. Not visible in the image but nevertheless important is that points outside the viewport are discarded. Especially for close viewing distances, only a small fraction of points is shown on the monitor. Therefore, the memory requirement and the processing time of the not visible points could be discarded until the camera moves.

4.2 Overview

To overcome the challenges previously proposed, an overall concept is introduced in this section.

It is important to consider that a point cloud is an approximation of objects in the real world. This type of representation finds its origin in the measurement of the scenes visualized by the point set. Laser scanners sample the environment and store depth data at points where the laser beam hits a surface. In contrast to a point cloud, objects consist of continuous surfaces, at least if one stays above viewing distance of the atomic space. These surfaces can also be described by other geometric primitives like cylinders, rectangles, or spheres. The advantage of these geometries is that individual primitives can describe a larger space than points. Planar surfaces, as for example identified in the previous chapter, consist of thousands of points. Instead, each surface can be described by a single rectangle, which requires only four vertices to store. If these points are replaced by the identified geometries, the memory requirements of the point cloud can be substantially decreased. Likewise, the render time could be reduced, since significantly fewer primitives have to be processed. There already exists a wide variety of algorithms, that identify geometric shapes in the point set. One of these algorithms can be used to segment points in the dataset based on the previously mentioned geometry.

Afterward, as can be seen in Figure 4.2, a distinction is made between classified and unrecognized points. Although the real world consists of many surfaces, not every region can be described by simple mathematical geometries. Some objects consist of more complicated structures, so that not all points can be classified. These points have to be processed to allow a simplified representation and a more efficient storage.

Likewise, the classified shapes have to be processed further, in order to create a realistic representation. Many point datasets include color information for each point. This allows to create a more realistic representation of the visualized objects. Therefore, a method has to be developed to transfer the color information of the classified points to their geometric shape representatives.

In the next step the unclassified points, as well as the detected geometric shapes, have to be stored. When rendering the dataset, hidden geometry or primitives out of the viewport can be discarded. This is based on the previous observation, that these objects are not visible on the screen. Therefore, the chosen data structure should support such culling techniques. Additionally, the data structure has to efficiently store the processed geometry and pass them to the render-engine. In the last step, the preprocessed is visualized on the computer screen.

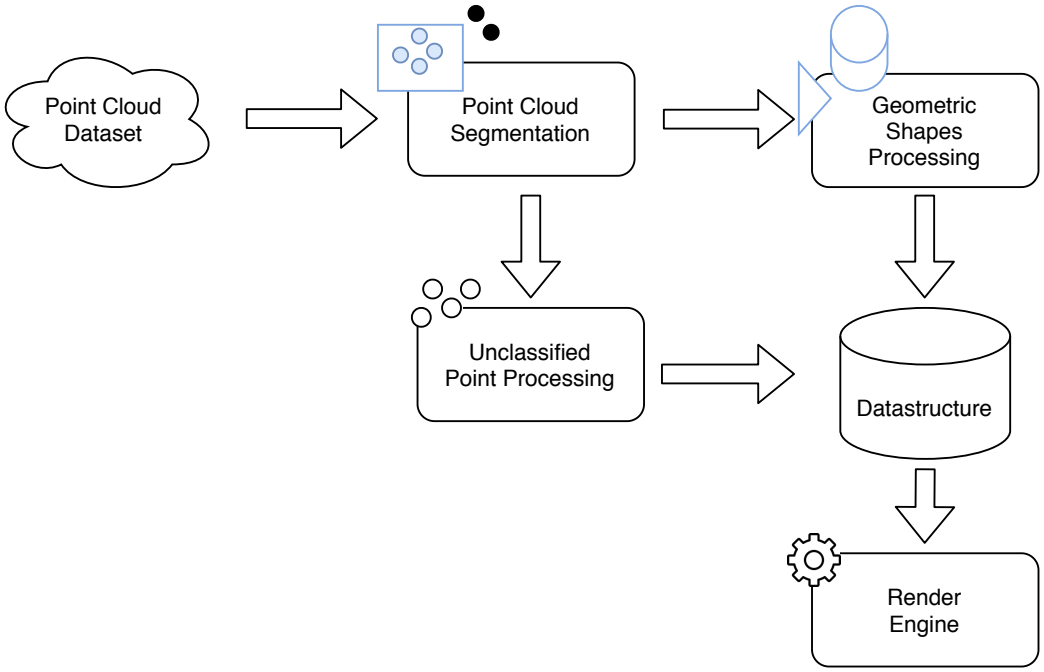

Figure 4.2: Overview of the presented processing pipeline. In the first step, geometric shapes are discovered in the dataset. Therefore, points lying in such shapes are classified and replaced by simple geometry like triangles, rectangles, cylinders, or spheres. Afterward, the approximated geometry has to be processed further to achieve a more realistic representation. Previously unclassified points have to be processed as well to match the level of detail of the other geometry. Afterward, both, unclassified points and geometric shapes are stored in a data structure. This data structure should support further processing steps to reduce the complexity of the dataset given a camera position. Finally, the data structure sends geometry to the rendering engine in order to visualize the processed point cloud.

4.3 Geometric Shape Processing

Likewise to a point cloud, it is important to recognize, that the detected shapes of a point cloud segmentation method are an approximation of real-world objects. Although, structural shapes such as planes, cylinders, and spheres occur, they are not given in perfectly regular shapes contrary to their mathematical representatives. Surfaces of objects can have different curvatures or even recesses. In addition, surfaces are not perfectly smooth. Depending on the material and condition they have irregularities such as indentations or bumps. These irregularities can be modeled by measuring the displacement of data points from the surface. Subsequently, choosing a finer tessellation level and then displacing vertices in the direction of the respective normal vector, creates a more detailed representation.

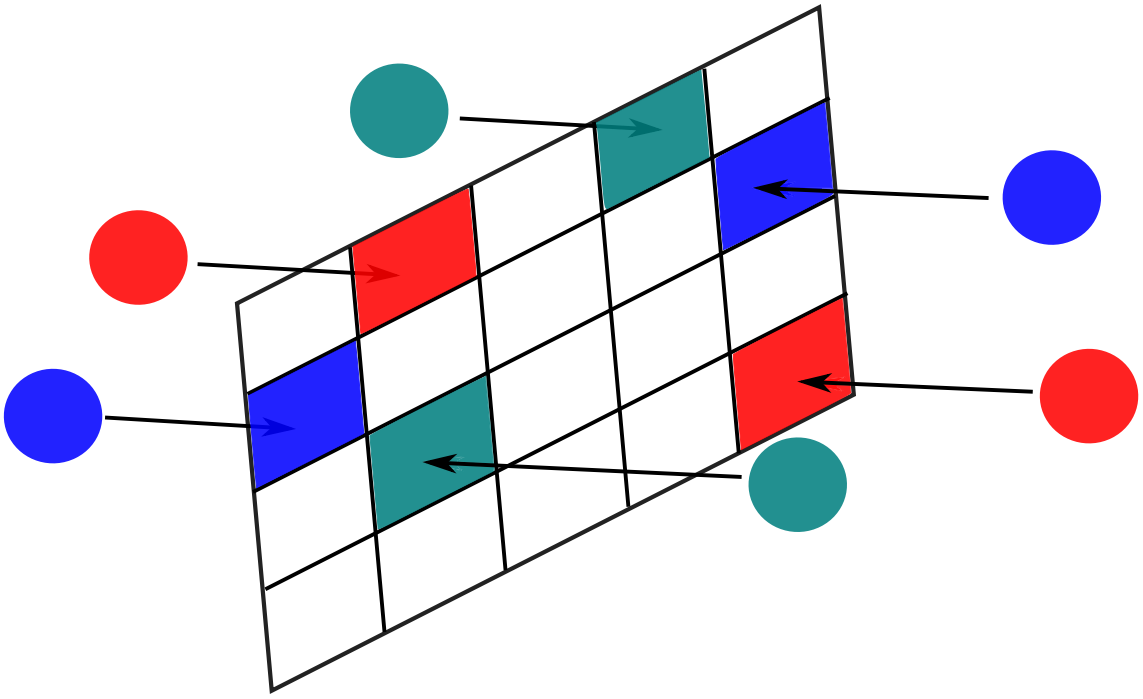

Figure 4.3: To create a texture, the colors of the classified points are mapped to a grid on the approximated shape. Then each cell encodes the color of a texel. From this information, a texture can be created.

Furthermore, the color of the surfaces must be described by their representatives. A common method in computer graphics is to save the color of geometries as a texture. To create these textures, the color-coding of the individual points must be projected onto their representatives. For this, a suitable uv-parameterization of the classified points has to be generated. In more detail, the uv-coordinates determine where each color point is located on the final texture. To achieve this, we use a method that constructs a grid over the area of the approximated geometry. Subsequently, the classified points can be projected into the different cells of the grid. Figure 4.3 illustrates this process.

4.4 Data Structure

Implementing a dedicated level of detail data structure is an important task. Since point cloud datasets can contain billions of points the dedicated data structure should support efficient storage of both the point data as well as other geometric primitives. Furthermore, in previous works, a data structure is used to process the point cloud out-of-core. When the size of the raw point data alone will not fit the RAM or VRAM of commercial computers, the data can be preprocessed. Afterward, only geometry visible in the current view is passed to the CPU or GPU. The rest of the point cloud is stored on the computer hard drive. When the view changes, the data structure has again to be traversed and the visible geometry is passed to the processing unit. An effective traversal strategy can therefore speed up the time until the geometry is rendered.

Another common method often combined with an out-of-core approach is to store different levels of detail. As stated in Section 4.1 the human eye is less capable of distinguishing adjacent shapes for further viewing distances. Additionally, the computer monitor is not able to visualize details smaller than the size of a pixel. Therefore, with an increasing distance to the camera, fewer details have to be shown. To realize this, precalculated representatives are chosen for a given distance range. Previous works like multiway kd-trees [GZPG10] or Potree[Sch16] take advantage of spatial data structures to implement this approach.

In contrary to the two previously mentioned methods, the representatives in our approach contain other geometric types than point primitives. Therefore, our data structure needs to support the storage of geometry types like cylinders, rectangles, or spheres. A BVH-tree [Cla76] is suitable for saving these shapes. This is due to the fact that the geometries can be well described by their bounding boxes. Going further, the number of primitives discovered is significantly less than the number of points in the point cloud. Therefore, the additional storage requirement for the pointers and the bounding box is negligible.

When building the BVH-tree, detected shapes build the leave nodes of the hierarchy. Here, the calculation method of the bounding boxes is dependent on the geometry type. For example, the bounding boxes of a rectangle can be identified by using the maximum coordinate values in each dimension of the four vertices. To differentiate between the various geometric types a parameter in the leaf nodes is used. Each stored representative can be visualized in different levels of detail. Therefore, for every node in the BVH, we store the bounding box and vertices of the representative. An additional byte is used to encode the geometric shape of the primitive. Otherwise one would not be able to recognize during the render phase which shape should be rendered solely based on the vertices of the geometry.

Another challenge arises in how to create a hierarchy and link the different nodes in the BVH-tree. With identified primitives lying in the leaf nodes of the tree, higher-level nodes have to aggregate two higher-level nodes. Therefore, a suitable method has to be found, which constructs the BVH-tree. With the underlying idea that parent nodes could represent coarser representatives summarizing two or more nodes, different strategies come into question. Here, the challenge lies in recognizing and assembling coherent primitives. Ideally, semantically related primitives such as a chair or a table are grouped first.

A common heuristic, when building a BVH-tree, is partitioning along the axis the largest range of centroids in a top-down approach is. While this approach is fast and simple, larger planes are less likely to be merged early even when other shapes lie near them. Another approach is to merge the two primitives with the shortest distance between each other in a bottom-up approach. The latter approach creates better representations, but the build time increases. Additionally, the BVH-tree is more likely to be highly unbalanced. To recognize semantically related structures, semantic segmentation methods can be used. However, this would require an additional processing step. The surface area heuristic (SAH) [GS87] optimizes the tree structure to reduce the necessary ray primitive intersection. Although SAH optimizes rendering times, it becomes more challenging to create higher-level representatives.

Besides geometric primitives like planes and cylinders, the data structure needs to be feasible to store the raw point data. The point data provides the most accurate representation of the real-world properties. Thus, for a detailed view, the user should always have the possibility to visualize this data.

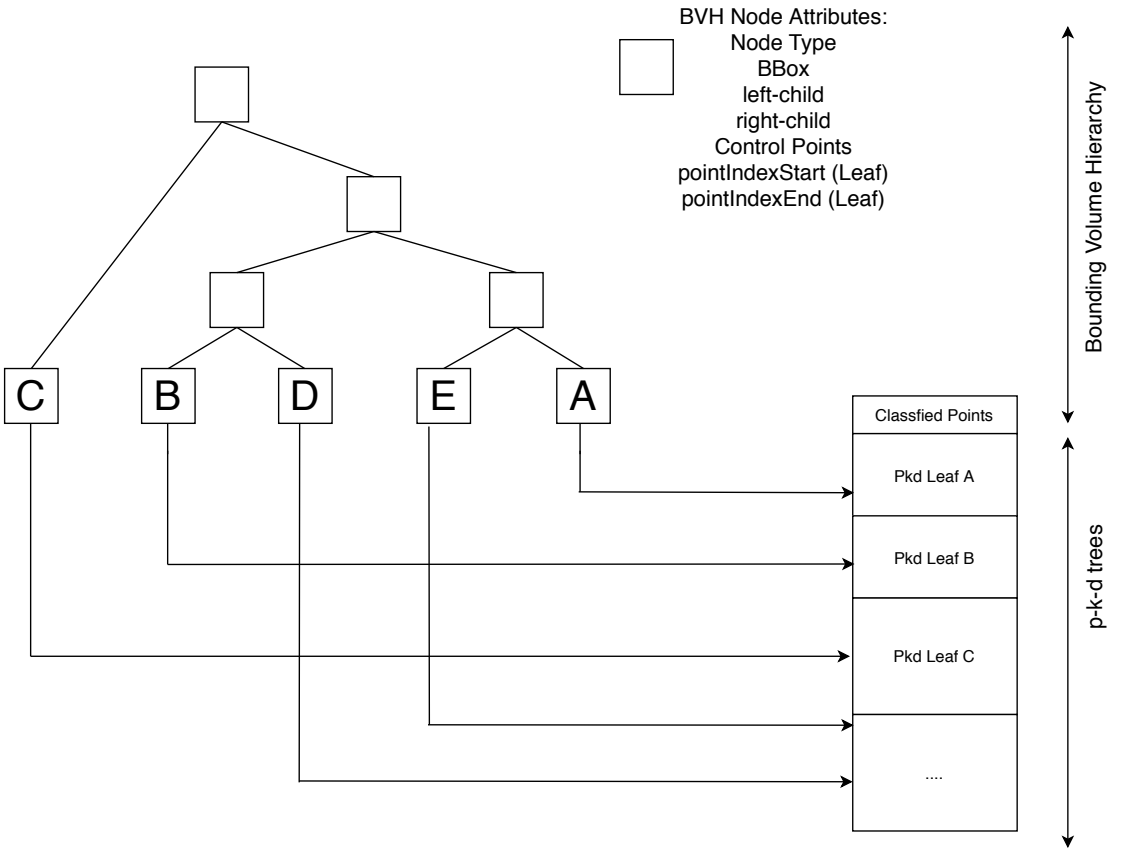

Figure 4.4: The level of detail data structure used in our approach. Approximated geometry is stored as a leaf node in a BVH-tree. Higher nodes can further summarize the approximated shapes. Each node contains a bounding box as well as the control points, and pointers to their children. Additionally, leaf nodes contain indices to their represented points. Each of these point subsets is processed as a pkd-tree.

In our approach pointers to the classified point data subsets of each geometric shape are stored in the leaf nodes. To minimize the required memory, we use pkd-trees to store these point subsets. The pkd-tree reduces memory overhead by only needing to sort the particle data. Then, after building each pkd-tree, the point indices of each point subset can be stored in the leaf nodes of the BVH-tree. An illustration of the presented data structure can be seen in Figure 4.4.

Lastly, we need to differentiate between classified and unclassified point data. Until now, we only considered classified points in our data structure. However, unclassified points have to be processed as well. These points undetected by our planar point cloud segmentation method may be freely distributed in the bounding box of the point cloud. In a similar approach to the previous method, an additional pkd-tree can be used to process this data. However, the unclassified points can be spread non-uniform over the entire bounding box of the whole point cloud. Unclassified points can be a result of regions with high curvature, sparse regions with insufficient points, noise, or points approximating not identified shapes. Therefore, a single pkd-tree may slow down rendering times. Thus, unclassified points have to be further subdivided. For this, we are using a clustering approach,

finding cohesive point regions. Afterward, an additional similar BVH-tree as introduced earlier can be constructed. However, the leaf nodes do not contain different geometric types compared to the previously presented BVH. Instead, the classified point clusters are stored in the leaf nodes. Dividing the unclassified point data from the approximated shapes can accelerate ray intersections, since less geometry is overlapping.

5 Implementation

This chapter focuses on the implementation of the previously presented concept. For this purpose, the used technologies are described in more detail. Furthermore, various implementation details, as well as design choices, are discussed. Therefore, each step of the implemented processing pipeline is introduced and described.

5.1 Used Technology

The presented approach was implemented as a plugin in the MegaMol framework [GBB+19]. MegaMol is a visualization prototyping framework implemented at the Visualization research center of the University of Stuttgart. The plugin is written in the C++ Programming Language. Additionally, some OpenGL code is used to visualize interim results. To process point cloud datasets, the Adaptable IO System(ADIOS) [LKS+08] is utilized. ADIOS allows to load, store, and adapt the data inside of MegaMol. For the nearest neighbor search, we used the nanoflann[1] header library. Nanoflann is based on a kd-tree to process two or three-dimensional point clouds. Nanoflann provides a k-nn search as well as a fixed radius nearest neighbor search. Approximate nearest neighbor and ranged nearest neighbor search are not supported. The STB image library[2] enables saving and loading textures as png files. To render the final result, the OSPRay renderer [WJA+17] is used. OSPRay is a CPU based ray-tracing framework developed to visualize scientific datasets. This work is based on OSPRay version 1.8 immigrated to the MegaMol framework [RKRE17].

5.2 Overview

A MegaMol plugin is composed of several modules communicating through data calls. Each module processes a point cloud which is loaded at the beginning. Afterward, the dataset is processed by the module next to the data loader and subsequently passed to the following module. Lastly, the dataset can be visualized by passing it to the view module at the end of the processing pipeline. An overview of all modules used in this work and their inner connections is shown in Figure 5.1.

In the beginning, a point cloud is loaded as an ADIOS file. Then normals, as well as curvatures, are calculated by the NormalEstimator module. These normals and curvatures are required for the following point cloud segmentation module. This module implements a region-based segmentation

[1] https://github.com/jlblancoc/nanoflann
[2] https://github.com/nothings/stb

algorithm. Here, planar shapes are identified and points corresponding to the detected shapes are classified. Afterward, the RegionGrowing module splits the dataset into classified and unclassified points.

The classified points are used to fit rectangles into the approximated planar regions. Afterward, an algorithm computes image textures combined with displacement values for each rectangle and stores them as image files in the TextureGenerator module. Subsequently, the Tessellation module combines the computed textures and rectangles by triangulating the given mesh based on the resolution of the texture and the stored displacement values. Furthermore, the uv-coordinates and vertices of the mesh are computed.

Meanwhile, the unclassified points are passed to the DBScan module. The DBScan algorithm enables the detection of spatially connected point regions. Thereupon, each cluster is segmented further by spatially subdividing the dataset in the Treelet module. For each of these subsets, a pkd-tree is build to accelerate the calculation of ray intersections. Finally, the created pkd-trees, are forwarded to the OSPRay renderer and stored in an OSPRay intern BVH-tree. Additionally, points neither classified by the region growing or DBScan algorithms are passed on to be added individually into the BVH-tree created by OSPRay. In the last step, the approximated planar shapes are visualized as triangles. The remaining points are rendered as sphere geometry.

5.3 Normal Estimation

The point cloud segmentation algorithm used in this work requires precomputed normals and curvatures for each point in the point cloud. Therefore, the normal estimation module calculates curvatures and normals of each point in the given dataset. For this, a simple approach using PCA is provided. This approach is based on the work of Pauly et al. [PGK02]. The PCA approach is fast, intuitive, and easy to implement. Nevertheless, for higher precision or noisy point cloud data other approaches like introduced by Mura et al. [MWP18] lead to more precise and robust results.

Typically, the point normal in a point cloud approximates the surface of a local point region. To approximate this region, adjacent points have to be found. Then the surface is estimated by fitting a plane in this set of points. Now, the normal of the plane estimates the surface at the position of the queried point. Given an unstructured point cloud, a plane can be fit in a set of arbitrary points using principal component analysis. This is done by calculating the covariance matrix of the chosen point set. Then eigenanalysis yields the three principal components. The two eigenvectors with the highest eigenvalue span the fitted plane in three-dimensional space. The third eigenvector is perpendicular to the two span vectors describing the divergence from the points to the fitted plane. Thus, the third eigenvector provides an approximation of the plane normal. Therefore, calculating the third eigenvector on a set of nearest neighbors estimates the local surface at the position of the points.

The region searched for nearest neighbors affects the resulting normal. As can be seen in Figure 5.2, fitting a plane in small regions is sensitive to small local fluctuations. In contrast, using a larger set of neighboring points averages small local changes. Thus, the calculated normals do not deviate much from adjacent points resulting in smoother but less precise transitions between points. In return, the calculated normals are more robust against noise.

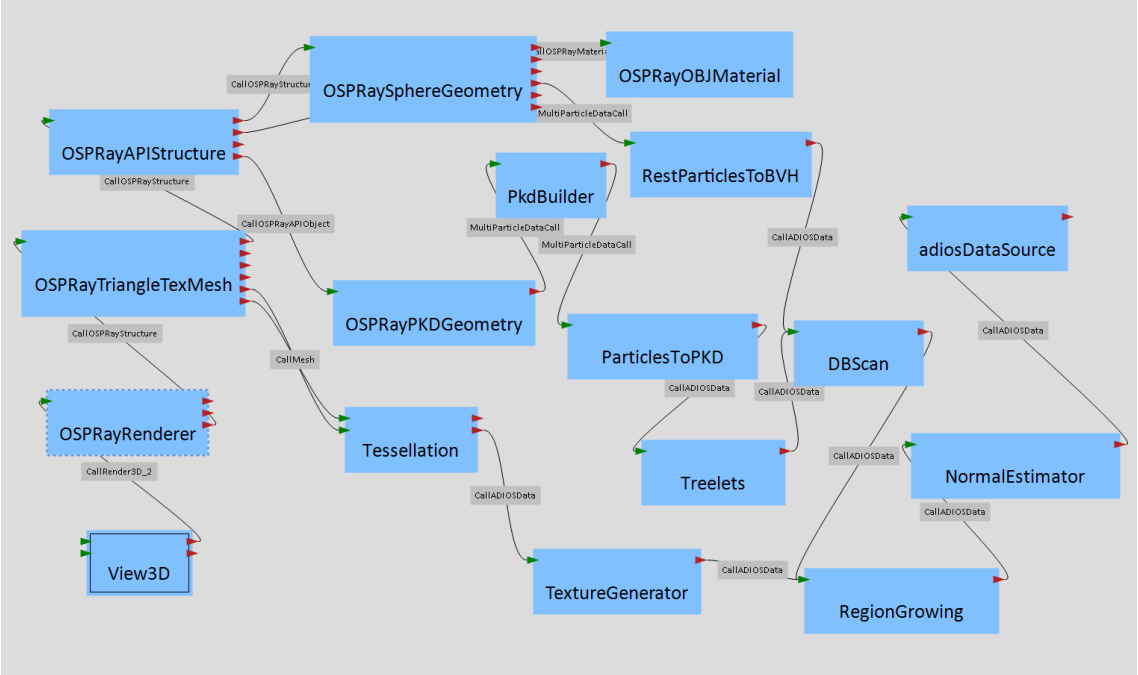

Figure 5.1: Overview of the implemented OSPRay modules. Modules are visualized as blue boxes, data calls are encoded by the connection between the modules. First, the point cloud is loaded by the adiosDataSource module. Afterward, normals and curvatures for each point are approximated. The computed results are utilized in our Region Growing algorithm finding planer areas in the dataset. Subsequently, the dataset is split into classified and unclassified points. Classified points are represented by rectangles. Related textures and displacements values are calculated in the TextureGenerator module. Based on the resolution of the resulting textures, the rectangles can be further tessellated in the Tessellation module. Finally, triangles of the resulting mesh are passed into the OSPRay internal BVH. Meanwhile the DBScan algorithm clusters unclassified points. The identified groups are then further subdivided into the Treelet module. Subsequently, a pkd tree for each cluster is created. Similar to the triangles, each pkd-tree is added to OSPRay's BVH-tree. Finally, points left unclassified after the DBScan algorithm are individually added to the BVH as sphere geometry.

Additionally, depending on the number of nearest neighbors and the size of the point set this computation can be time-consuming. To accelerate these calculations and the nearest neighbor search the algorithm is performed in parallel using OpenMP[3]. A smaller radius leads to a noticeable faster processing time since the nearest neighbor search and PCA have to process a smaller amount of points.

[3] https://www.openmp.org/

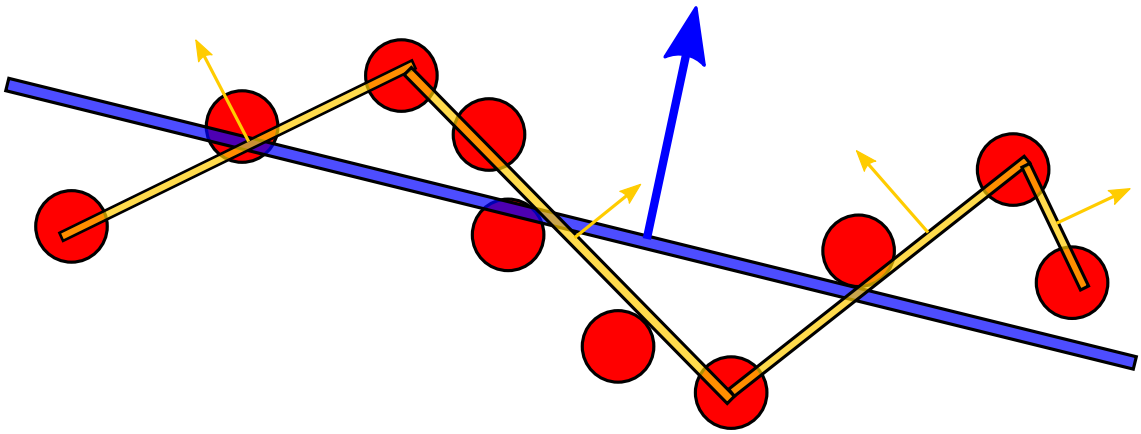

Figure 5.2: Fitting a surface in a larger set of neighbors results in a plane averaging the points' positions. Smaller regions provide more precise normals. However, they are less robust to noise.

5.4 Pointcloud Segmentation

There are different approaches to find regions with related properties in a point cloud. For this work, we chose a seeded planar region growing algorithm based on Nurunnabi et al. [NBW16]. The algorithm has the advantage of being easy to implement and modify. Furthermore, the algorithm has several adjustable parameters.

Given an unorganized point cloud with position, normal, and curvature for each point, we want to fit arbitrary planes in planar regions approximating the points lying in 3D space. Thus, adjacent points with their surface normal pointing into a similar direction are classified into the same cluster. In the first step, a set of seed points is sorted based on their curvature in ascending order. Then the point with the least curvature is added to a set of seed points. To find adjacent points a nearest neighbor search is performed. Afterward, the queried points are added or dismissed for the current cluster. There are four different options how adjacent planes can be separated from each other which can be seen in Figure 5.3. A sharp angle between adjacent normals indicates an edge between two fitted planes. These edges occur, for example, when two flat walls are connected and point in two different directions. These two surfaces can be distinguished by a maximum threshold angle θ between the adjacent normals n_i and n_j as seen in Figure 5.3 a). The angle is calculated by $\text{acos}|n_i.n_j| < \theta$. When the previously calculated normals are not aligned we need to calculate the absolute value of the dot product. This considers the ambiguity that the normals can point in the opposing direction.

However, while this heuristic covers many cases, it is not sufficient in itself. If there is a horizontal jump between two planes, as shown in figure Figure 5.3 b), the normals of all points would point in the same direction. Thus, we calculate a second condition limiting the horizontal shift by a threshold. As a third constraint, two planes can be separated by a certain distance shown in Figure 5.3 c). In contrary to Nurannabi et al. we do not calculate a maximum threshold of the median distance. However, we limit the radius of the nearest neighbor search, discarding points that exceed a maximum distance limit in the k-nn search. The last option is a curvy edge, seen in Figure 5.3 d). Having constant curvature in local regions can lead to a slowly arching surface.

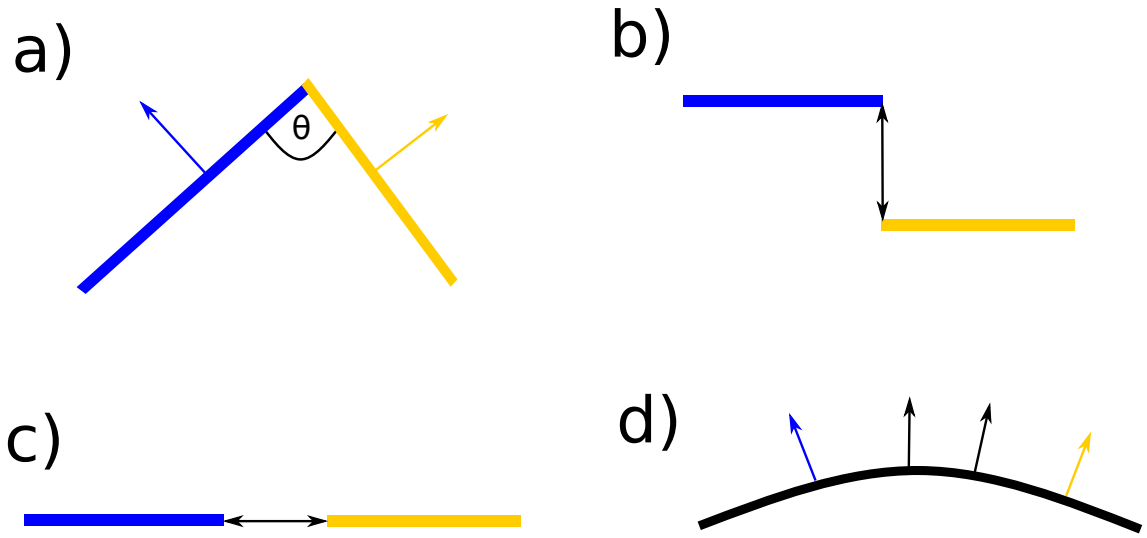

Figure 5.3: Four cases where two adjacent surfaces have to be differentiated. In a) the angle between the two planes exceeds a maximum distance. Furthermore, the horizontal distance between two planes can exceed a maximum horizontal distance limit. Besides the horizontal distance, the euclidean distance between two planes can be higher than a defined threshold. Lastly, a global angle between the current point and its seed point can surpass an angular limit.

When comparing two local normals in those regions, the angle between them may be smaller than the set normal threshold. Thus, the implemented region growing algorithm could additionally detect non-planar geometry like slowly arching cylinders or spheres. To exclude this, we compute the angle between the start seed point with the lowest curvature and the current point. This angle is then compared to a global angle threshold.

To find adjacent points we use nearest neighbor search. As already stated, we want to limit the radius of the nearest neighbor search by a maximum distance threshold. This makes fixed radius neighbor search a good option. However, the point density of the point cloud is not guaranteed to be uniform. That implies fluctuating numbers of found nearest neighbors. Inherently, the outcome of our algorithm does not change significantly with a variable number of neighbors, contrary to the normal estimation approach. When finding fewer nearest neighbors, their adjacent points again would be detected in the following nearest neighbor queries. Yet, when there is a thin edge between two bordering planes, finding more neighbors could skip this edge by also querying points beyond that edge. When using with a fixed radius search, we can exactly limit this skip. However, a disadvantage of fixed radius neighbor search is the performance. While in coarse regions a fixed radius may find only no to few neighbors, the same radius can result in thousands of neighbors in dense regions. In our algorithm, we add all found neighbors into a set of seed points. With more found points the set of seed points grows. Thus, the same points may be processed more often. Furthermore, the nearest neighbor search itself will become slower if more points are found. K-nn search diminishes this performance constraints. Nevertheless, we do not have a fixed radius limiting the maximum distance between two aligned planes. With ranged nearest neighbor search the maximum distance, as well as the number of nearest neighbors, is fixed. But here again, since

not all points lying in a radius are found, a thin edge of points is not guaranteed to be skipped over. Therefore, limiting both the number of nearest neighbors, as well as the search radius provided the reasonable results and runtime performance.

Overall, our implemented region growing algorithm has a wide range of adjustable parameters. These parameters are the minimum point size of a cluster, the horizontal distance threshold, the maximum local normal angle, and the maximum global normal angle.

5.5 Texture Generation

After the applied point cloud segmentation, a set of classified points results. Each cluster approximates a plane. Since these planes span an infinite two-dimensional space in three dimensions, we want to only visualize a finite subset of this space. In more detail, we want to find four corner points to create a convex quadrilateral. To reduce unnecessary intersection tests later, the quadrilateral should fit the set of points well. Large areas, where no points can be projected to the geometry should be avoided. Additionally, since we want to map a texture to the quadrilateral, we choose to diminish the choice of possible types of quadrilaterals to rectangles. With this, we facilitate uv-mapping later.

Given the normal of the fitted plane, we can project the three-dimensional points into two dimensions. Then, simply using the minimum and maximum x and y coordinates of the projected points as rectangle vertices, generates a convex set. However, the projected data points can be arbitrarily rotated at the center, resulting in a set of infinite possible bounding points. As stated before, we want to choose the four bounding points in an optimal way to reduce the area of the constructed rectangle. However, computation time has to be considered as well. The cost for tweaking the variables for a perfect fit exceeds the later rendering benefits at a certain point. By using principal component analysis, we find a good tradeoff between both factors. The principal component analysis provides the three principal components of the point set. Transforming these points to the eigenbasis of the covariance matrix enables a projection to two-dimensional space, by disregarding the vector with the smallest eigenvalue. Now, the x-axis points in the direction with the highest variance of the data. Again, we can use the minimum and maximum values in x and y direction to find a good uv parametrization. Since the eigenvectors of the covariance matrix are perpendicular, these values can be used as vertices of a rectangle. Thereby, the rectangle creates an approximation of the analyzed points as seen in Figure 5.4.

Furthermore, we can use the rectangle as a representative of a cluster in the level of detail data structure. However, until now we solely estimate the shape of the point cluster. In addition, all points have a color attribute, creating a more realistic representation. As already mentioned, with the points lying in two-dimensional space, we can create a texture approximating the color of all points. For this, we create a raster over our two-dimensional transformed dataset. Every cell of the raster represents a texel in the outcoming texture. The top left corner represents the texel with uv-coordinates of both zero plus half the size of a cell. The most right texel has a u-coordinate of one minus half the size of cell. The color of a texel is chosen by analyzing all points lying in the respective cell. For this step, we use color interpolation. To guarantee a better-resulting color, the color format of all points is first converted from RGB to HSV. A HSV color representation allows to separate saturation and luminance when interpolating. Nevertheless, we additionally implemented the color interpolation in RGB, allowing the user to choose the better fitting method.

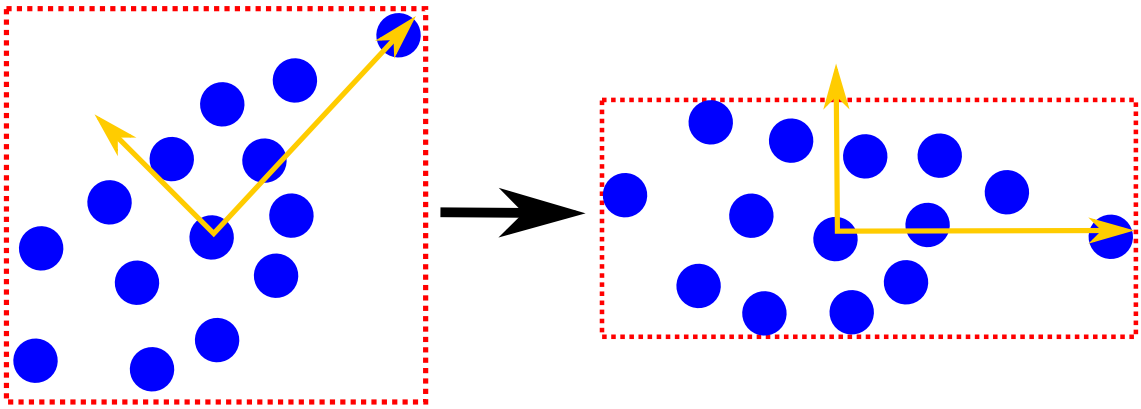

Figure 5.4: Principal Component Analysis is used to find the two perpendicular vectors with the highest data variance. Then transforming to the eigenbasis yields a good approximation of a fitted rectangle. Subsequently, a texture using uv-coordinates of the rectangle can be created by building a grid over the data points.

Afterward, independent from the chosen method all points are weighted equally and the color is linearly interpolated. In the end, we can store a texture based on the calculated texel values. Additionally, we can change the texture size by adapting the cell height and width of the raster. With bigger cells, more points are mapped to a texture and the texture has a lower resolution. If a too small cell size is chosen, high-resolution textures containing many transparent texels result. This is caused by the fact that many cells do not contain any points. To avoid this effect, a cell size larger than the point density of the point cloud should be selected.

Ideally, every cell contains at least one point. However, based on the shape of the point cluster, the cell size, and the chosen rectangle vertices, there can be cells enclosing no points. Additionally, the detected planes do not have to be continuous. Holes in the approximated surfaces can occur. To account for this, we add an opacity value to each color and the texture is stored with a red, green, blue, and opacity channel. If the opacity is zero, there are no points in the respective area. Thus, the texel has to be transparent. These translucent texels have to be addressed in the rendering stage later, by for example discarding fragments. Since the color information of each point is given as byte values, the resulting texture stores color with byte precision.

Another challenge that needs to be addressed is that the classified points do not have to be perfectly flat. Points can vary between a certain threshold in the normal direction of the fitted plane. By ignoring that fact, some planes may not fit together perfectly at the boundaries. Furthermore, the surface of the representative shapes is perfectly flat. However, surfaces described in a point cloud can additionally contain bumps and small deviations. Thus, while for coarser representatives this approximation is good enough, we want to add this detail for nearer viewing distances. To do this, we calculate a displacement distance in the direction of the respective plane normal for each point. Afterward, we store the calculated value as a displacement map. This displacement map can be used later to tessellate the rectangle mesh.

To calculate the displacement values, data points are not transformed into two-dimensional space when using PCA. Instead, the points are transformed to the three-dimensional eigenbasis. Subsequently, the third eigenvector is used to account for variation in the direction of the fitted surface normal. Therefore, the rectangle has to be translated to the average point value in direction

of the third eigenvector. With this information, the displacement for each point can be computed by calculating the coordinate value on the third principal axis. There exist various approaches to find the plane average. Using the statistical mode fits the plane so that most data points lie directly on the plane. Nevertheless, since our planar segmentation method limits the divergence in the normal direction, in the worst case, the maximum displacement value is two times the segmentation threshold. All in all, fewer points need to be relocated using this method. However, having only a limited amount of memory space to store the displacement value, the floating-point precision of the displacement is less precise. Another approach is to calculate the mean position of all points. Here, the maximum displacement value is smaller than by using the mode. Furthermore, most points lie near the fitted rectangle. Yet, more points must be shifted, unless all points lie directly on the rectangle. Last, calculating the midpoint between the minimum and maximum value yields the smallest maximum divergence. However, the rectangle fits all points less precisely, resulting in a high mean error.

Furthermore, the calculated displacement value for each point has to be stored. Using an additional displacement map would result in some memory overhead since we only store a binary value in the opacity channel of the previously calculated texture. Therefore, the displacement value is integrated into the texture. Having an eight-bit precision for opacity, one value is reserved to encode whether the texel is opaque. The remaining 255 values store the displacement information. With a zero point, we can shift all points with a floating-point precision of $\frac{1}{127}$ times the max displacement value in or opposing to the normal direction.

5.6 Tessellation

This section describes the implemented tessellation algorithm. Our approach triangulates identified rectangles based on the previously computed textures. In addition, vertices are shifted in direction of the normal vector of the rectangle based on the previously calculated displacement values. Using a GPU based rendering approach, tessellation and displacement are usually done on the fly given a camera distance. However, OSPRay, which is used as the rendering engine, does not support displacement mapping and on the fly tessellation. Thus, we provide a workaround by precalculating different level of detail representatives. Coarser tessellation and displacement lead to a lower level of detail with larger displacement deviations between two vertices. By contrast, finer tessellation results in more triangles and therefore a more detailed representation.

The algorithm starts with the vertices of a computed rectangle and its displacement map. The main idea lies in tessellating the rectangle for every texel in the texture. Then each vertex can be shifted by the displacement value of the texel. To do this, we start at a corner vertex of the rectangle. Subsequently, we process four texels which are adjacent and form a square on the surface. For each texel, its position on the rectangle is calculated. Thus, the two normalized tension vectors are multiplied with the u and v coordinate of the texel. Now, two triangles for every four points can be built. Afterward, we can move the vertices by the displacement value in the direction of the rectangle normal. The normal is retrieved from the previously implemented texture generation algorithm. Using the third eigencomponent avoids that a vertex is shifted in the opposite direction. This can occur because the normals in the data set are not directed. This method provides an easy and reliable tessellation. However, it can still be improved. Until now, a triangle is created for each texel. Since a texel can be transparent, some triangles are not visible in the rendered

image. Therefore, they can be already discarded when running the tessellation algorithm. If more than one vertex is transparent no triangle will be constructed for the given area. This approach enables the rejection of many transparent vertices. Note, as the vertex density relies on the texel size of the texture, the higher the previously defined resolution of texture, the finer is the resulting tessellation.

5.7 Data Structure

Since OSPRay relies on its internal data structure. The in Section 4.4 introduced level of detail data structure could not be completely realized. Similar to the proposed concept classified rectangles are stored as triangle mesh in the internal OSPRay BVH. However, since only a single BVH-tree is supported all unsegmented point cloud data has to be stored in this tree contrary to the proposed second BVH-tree. Before being added to the BVH-tree, all unsegmented point cloud data is further classified by the DBScan algorithm. Then the identified clusters can be stored as pkd-trees. However, the DBScan algorithm does not limit the maximum space a cluster can take. To prevent large clusters with inefficient bounding boxes the segmented regions are further subdivided.

For this, we use an approach presented by Gralka et al. [GWG+20]. First, each cluster is split into smaller segments of a fixed size. Gralka et al. divide nodes by splitting along the longest bounding box axis. Given the previous example of three connected lines, this approach can lead to an unbalanced tree. Instead of dividing the points of a cluster based on a spatial criterion, it is possible to split along the median data point of the longest bounding box. Now, a balanced tree is guaranteed. However, both cutting volumes no longer have the same spatial size. After splitting the data, pkd-trees for each point subset are created. These subsets are then added as pkd-geometry objects to OSPRay's BVH-tree. Finally, the points, that were not detected by either the point cloud segmentation method or the DBScan algorithm are added solely as point primitives to the BVH.

6 Evaluation

In this chapter, the implemented processing steps, as well as the rendered results, are evaluated. For this purpose, a parameter space is predefined. Subsequently, the processing steps are executed. Afterward, the obtained intermediate results will be presented and analyzed. For this purpose, we compare rendering times, memory consumption, and image quality with a sphere-based point rendering method in OSPRay.

6.1 Datasets

For the evaluation, two datasets provided by FARO are used. One of the datasets shows the cutout of an empty room. A large part of the room consists of unplastered walls that are covered with graffiti. With the walls and floor, the dataset contains large planar areas. However, most surfaces are rough and uneven. Furthermore, the point density is not uniform, which must be taken into account while running the preprocessing steps in our pipeline. In total, the dataset comprises 26130261 points.

The second dataset remodels a bicycle standing on the ground. With 14624303 points the dataset is smaller than the previously introduced point cloud. In contrast to the graffiti dataset, the point set contains significantly fewer and smaller planar areas. The modeled bike consists mainly of cylindrical shapes with high curvatures.

Besides point and color information, these two datasets contain precomputed normals. Nonetheless, we decided to use the self-calculated normal values presented in Section 5.3. This is based on the observation, that our normal calculation method allows more degrees of freedom. Here, the number of neighbors examined for the calculation can be determined by ourselves. Thus, the effect of precise and smooth normals can be analyzed.

6.2 Parameter Space

There is a wide range of parameters in our processing pipeline. Therefore, changing every parameter would result in too many combinations that need to be evaluated. To narrow this space further down we limit the parameter space to predefined configurations for each processing step. Table 6.1 below provides information about the most important parameter values used in the upcoming evaluation.

Since the implemented planar segmentation algorithm depends on the precomputed normals, we calculate the normals and curvatures for each dataset using two different radii. One radius is picked relatively small to include small bumps and changes. The second radius is twenty times the size of the previous parameter, providing a smoother normal distribution. For the planar segmentation algorithm, we explore two different configurations.

Table 6.1: There is a wide range of parameters that can be defined within the processing pipeline. Here, the 14 most important parameters are shown. Furthermore, the specific values used for the later carried out evaluation are listed.

Processing Step	Parameter	Description	Values
Normal and Curvature Estimation	rNorm	Radius threshold used for fixed radius neighbor search. Smaller values result in fewer neighborhood points being used to calculate the normal and curvature values.	Bike: {0.0005, 0.000025} Graffiti: {0.001, 0.00005}
Region Growing	Num Nearest Neighbors	Number of nearest neighbors used for the k-nn search.	Both: 40
	α	Local normal deviation threshold.	Both: {0.13, 0.35}
	β	Global normal deviation threshold.	Both: {0.13, 0.35}
	Minimum Points	Minimum number of points a cluster has to contain.	10,000
Texture Generator	Res	Cell size used to compute the texture.	Bike: {0.004 (High), 0.012 (Middle), 0.036 (Low)} Graffiti: {0.006 (High), 0.018 (Middle), 0.054 (Low)}
Tessellation	tess	Specifies, whether the given rectangles should be further tessellated.	Both: {True, False}
	delTransp	Specifies whether invisible triangles are removed from the mesh, when the mesh is tessellated.	Both: {True, False}
DBScan	ϵ	Radius parameter of the fixed radius nearest neighbor search.	Bike: 0.0005 Graffiti: 0.001
	Minimum Points	Minimum number of points a cluster has to contain.	Both: 100
Pkd Treelets	Maximum Size	Maximum number of points a treelet can comprise.	Both: 1000
Rendering	viewDist	Viewing distance to the rendered object.	Both: {Far, Middle, Near}

The first parameter set leads to a stricter segmentation. Here, only points with a small angular normal difference are classified together. The second configuration applies a higher angular limit. Thus, more points can be classified. However, the correlated points deviate more from the approximated surface. For the DBScan algorithm and treelet creation, we use a single parameter value. These configurations have been established as good practice for the given datasets and do not need to be compared further with other values. Since the two datasets have different point densities, the epsilon value of the DBScan algorithm changes. Furthermore, the values between both point clouds change for the texture generation step. Here, three different texture resolutions are used to investigate their impact on runtime and memory requirements. Lastly, three different configurations are likewise used for the tessellation processing step. First, the detected rectangles can be visualized solely or they can be tessellated in a preprocessing step. Additionally, transparent triangles detected during tessellation can be either discarded or kept.

6.3 Preprocessing

In this chapter, the results of the pre-processing steps in the conducted evaluation are presented. Furthermore, detailed reasoning behind the selection of the chosen parameter values is provided.

6.3.1 Normal Estimation

Besides the type of nearest neighbor search, the solely adjustable parameter for the implemented normal estimation algorithm is the number of nearest neighbors or the search radius. Using a k-nn search for the graffiti dataset introduced some challenges. Since the dataset contains regions with different point resolution, a normal calculation based on the number of nearest neighbors produced inconsistent results. For example, dense regions led to precise normals over a small area as the found nearest neighbors are near the query point. On the contrary, coarse regions led to more averaged normals calculated over a larger region. Here, the distance between the query point and a neighbor can be significantly higher. Therefore, we use a fixed radius nearest neighbor search neglecting this effect. This is because all nearest neighbors in a fixed space are searched. However, the number of found neighbors is not constant anymore leading a varying amount of points being processed.

When choosing a radius parameter value we wanted to be able to differentiate between highly averaged and precise normals. In the example of the graffiti point cloud, if the normals are calculated with a smaller radius, the wall joints have normals pointing in different directions to the wall. In contrast, with a larger radius, all normals are more average. Small deviations between two bricks are almost neglected. It is of interest to compare these two configurations later since the implemented point cloud segmentation method yields different results depending on the normal calculation.

6.3.2 Region Growing

The implemented region growing algorithm has the largest parameter space in our processing pipeline. To reduce combinations we chose a constant threshold of minimum points, a constant number of searched nearest neighbors, and a maximum distance between two adjacent points.

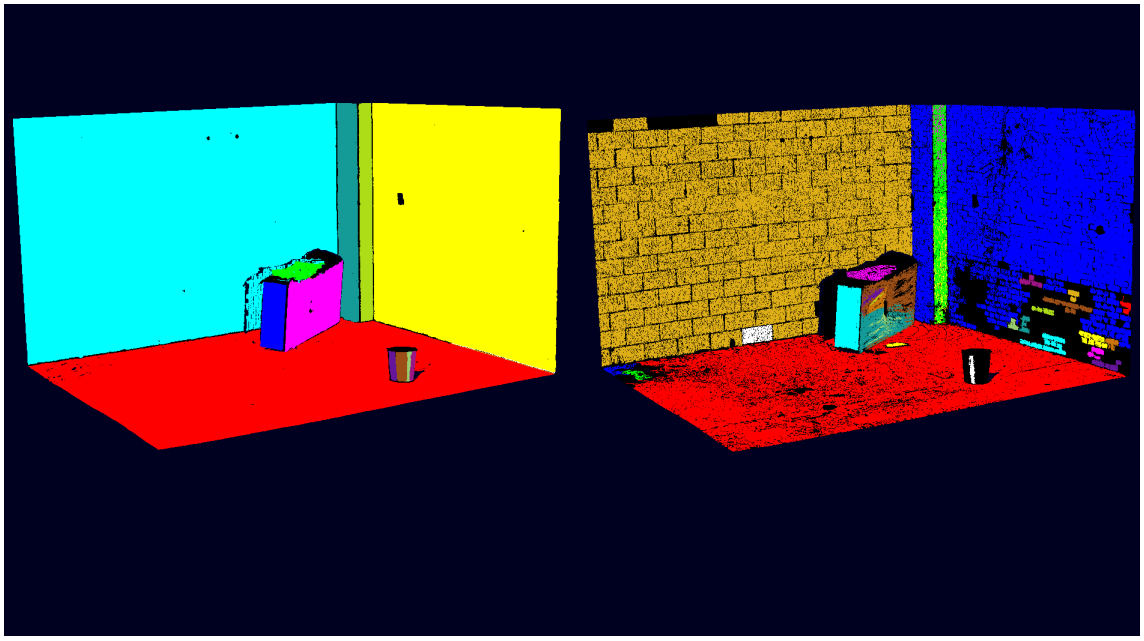

Figure 6.1: Two results of the implemented region growing algorithm. The left image used normals approximated over a space of 0.001 meters for each point. Additionally, with twenty degrees a high angle deviation limit was chosen. On the contrary, the right picture is based on normals over a range of 0.00005 meters and a maximum local and global angle of 7.5 degrees.

The selected values proofed to create good results. Furthermore, we neglected the horizontal distance parameter since it added no further value for the later processing steps. Here, the maximum search radius already provided a good limitation for the maximum distance between two points.

One goal of the later evaluation is to find out to what extent large detected areas influence the performance and image quality in contrast to smaller planar areas with a lower deviation from the point dataset. Therefore, we picked a small global and local angle of 7.5 degrees. In contrast, we introduced a large angular limit of twenty degrees. The result of both parameter values can be viewed in Figure 6.1. Especially for the rough and uneven surface of the walls, both values resulted in significant differences. A high angle threshold resulted in a significantly higher amount of classified points. Some inaccuracies can be observed on the green wall in the right image. Here, since the surface normals vary highly between adjacent points, two planar surfaces are classified over the same region, indicated by the purple and green color. Therefore, two overlapping planes result.

Furthermore, the outcome of the algorithm was different given different normal estimation parameter values. Averaged normals over a wider area allowed to detect coarser regions like walls and the floor in the graffiti dataset. Meanwhile, more precise normals made it possible to distinguish between smaller planar surfaces. When both parameters were low many adjacent points could not be detected. Here, a significant drop of classified points can be observed for the graffiti dataset. With 77.6% of all points, almost 5 million fewer points were detected.

Table 6.2: Results of the region growing algorithm for different parameter values. A higher radius for the normal calculation and a higher angular limit value lead to more classified points. In contrast, the standard deviation of the points in the normal direction of the approximated planes increases. In the bike dataset, fewer points are detected and more planes are approximated than in the graffiti dataset.

Dataset	Radius Normal Estimation	Global and Local Angle in Degrees	Detected Planes	Detected Points (Coverage)	Standard Deviation in Normal Direction
Graffiti	0.00005	7.5	35	20284261 (77.6%)	0.014099
Graffiti	0.00005	20	15	25059918 (95.9%)	0.018606
Graffiti	0.001	7.5	19	25185069 (96.4%)	0.018539
Graffiti	0.001	20	15	25870660 (99%)	0.021812
Bike	0.000025	7.5	49	8575954 (58.6%)	0.004492
Bike	0.000025	20	101	10752205 (73.5%)	0.011748
Bike	0.0005	7.5	82	9042540 (61.8%)	0.006378
Bike	0.0005	20	144	11871233 (81.2%)	0.019151

On average, the algorithm takes 180 seconds to process the graffiti dataset of 53 million points. This time is reasonable, however for larger point clouds exceeding over a billions of points processing times the implemented algorithm may has to be reconsidered. An octree-based implementation of the region growing algorithm [VTLB15] or a model-based segmentation method [SWK07] may provide better performance. As can be seen in Table 6.2 our implemented approach recognizes more than 58% percent of all points for each configuration. For the graffiti dataset, which consists mainly of planar surfaces, significantly more points are segmented, ranging from 77.6% to 99%. In contrast, for the bike dataset, significantly more small areas are detected. In addition, the standard deviation in the direction of the approximated normal was calculated for all configurations. For this the smallest distance of each point to its representative was calculated. Based on the nature of the region growing algorithm, a higher angle threshold, and smoother normals lead to a higher calculated standard deviation.

6.3.3 Texture Generation

The texture generation method fits rectangles in the previously classified points. Afterward, an image texture for each plane is created. Parameters for the algorithm are the cell height and width. A higher texture resolution is achieved by smaller cell sizes. Larger parameter values lead to blurrier textures that require less memory. With tessellation, the next processing step is dependent on the resolution of the texture. Therefore, we chose three different cell sizes encoding high, medium, and low texture resolutions. Since the average point density differs between the graffiti and bike dataset, individual values were picked for each point cloud. A resulting texture of this algorithm can be seen in Figure 6.2. The left image shows a high-resolution texture given a cell size of 6 millimeters. On the right, the same wall is shown using a cell size of 54 millimeters. The resulting texture of the image on the left requires 931 kilobytes of storage. The size of the right image is 14.6 kilobytes.

Figure 6.2: Two resulting textures of the implemented algorithm using different parameters. Both images show the texture of a wall generated from the graffiti dataset. The left image was generated using a cell size of 6 millimeters while we used 54 millimeters for the right one. Since the edges of the point cloud do not match the borders of the texture perfectly, some inaccuracies can be detected. These are most evident at the edges of a low-resolution texture. To avoid poorly visible areas the opacity values, describing the displacement of each pixel, were neglected for this picture.

The proposed PCA approach to determine an uv-mapping of the triangle showed good but not perfect results. Especially regions with diverging densities, shapes with straight edges or notches can result in artifacts when choosing a low image resolution. This can be seen on the boundaries of both textures shown in Figure 6.2. A lower resolution further magnifies this effect.

6.3.4 Tessellation

While the level of detail of our tessellation method is dependent on the resolution of the previously computed texture, two additional parameters can be selected. First, the user can choose whether the mesh should be tessellated. Otherwise, a single rectangle primitive is used for each detected planar region. Furthermore, when tessellating the mesh, transparent triangles, which are invisible to the user, can be discarded. In our evaluation, we decided to include all three possible combinations.

6.3.5 Unclassified Point Processing

Points not detected by our planar segmentation method are further clustered by the DBScan algorithm. Here, the goal was to classify a large part of the remaining points. However, too small clusters are dismissed to avoid pkd-trees containing a low amount of points. Therefore, the value 100 was chosen as the minimum number of nearest neighbors. For the search radius, a distinction was made between 0.001 for the Graffiti and 0.00005 for the Bike dataset. Running the algorithm on our datasets classified a majority of the remaining points. For the bike dataset, 99.4% to 99.8% of the previously unclassified points were detected. For the graffiti dataset, the algorithm classified

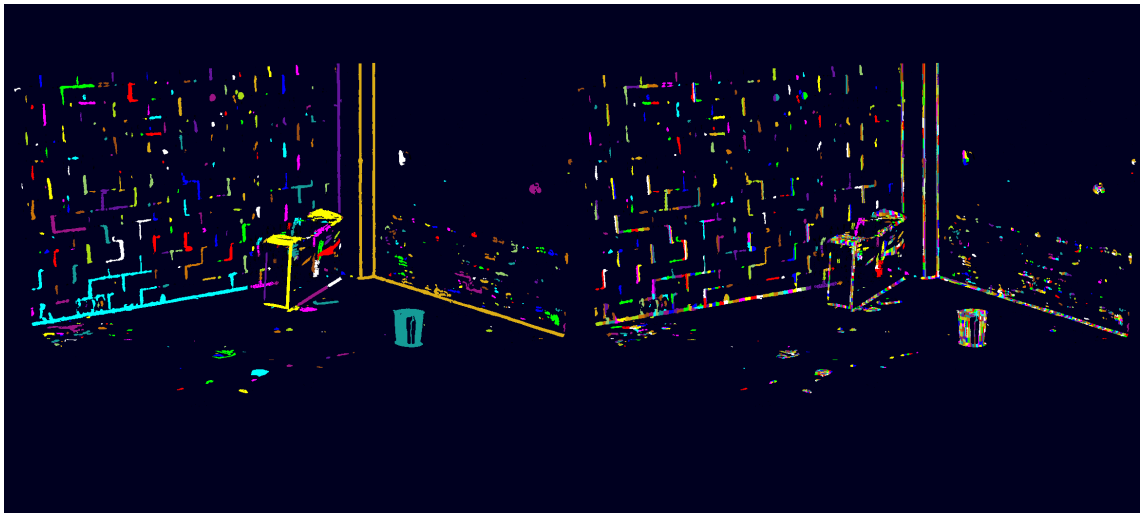

Figure 6.3: Processing of all unclassified of points in the graffiti dataset. The left image shows the result of clustered regions of the DBScan algorithm. The colored areas in the right image, encode how the detected clusters were further spatially divided.

on average a lower percentage of points in the range of 92.4% to 99%. Subsequently, all clusters were further spatially subdivided and partitioned into treelets. For this, a maximum point size of 1000 points per treelet was picked. The result of both processing steps can be seen in Figure 6.3.

6.4 Runtime Performance

To evaluate the runtime performance of our method, the presented parameter space was tested in MegaMol. All configurations were run on a computer with 16 GB RAM and a Ryzen 2600x, which has six cores and a 3.6 GHz base clock speed. Render times per frame were recorded in milliseconds for three different viewing distances. A far distance implies a camera position further away from the shown scene. Therefore, the dataset fills only a part of the viewport. A medium distance indicates a common distance for analyzing the whole scene. Here, the dataset almost fills the whole screen, while no part is cut out. Finally, a close view zooms into the dataset leaving a small part of the data at a close range visible on the screen.

These distances were determined for each of the two datasets and then used uniformly for all parameter configurations. To negate initial loading times, each configuration was first rendered for twenty frames. Afterward, the render time was recorded for the next hundred frames. All in all, our preprocessing parameter space includes 36 configurations for each dataset. Besides, each configuration was measured for the previously mentioned three different camera distances. Given the two datasets, this results in a total of 216 combinations. Since the 100 recorded values for each combination showed stable and only slightly fluctuating results, the mean rendering time for each combination is presented in the following bar charts. Subsequently, collected results were then compared to the results of a point rendering method. Here, more specifically, all point data was stored in a BVH and then each point was rendered as a sphere in OSPRay.

Configurations

Tess	del. Trans.	Res.	rNorm	α/β
True	True	0.054	0.001	7.5
True	True	0.054	0.001	20
True	True	0.054	5e-05	7.5
True	True	0.054	5e-05	20
True	True	0.018	0.001	7.5
True	True	0.018	0.001	20
True	True	0.018	5e-05	7.5
True	True	0.018	5e-05	20
True	True	0.006	0.001	7.5
True	True	0.006	0.001	20
True	True	0.006	5e-05	7.5
True	True	0.006	5e-05	20
True	False	0.054	0.001	7.5
True	False	0.054	0.001	20
True	False	0.054	5e-05	7.5
True	False	0.054	5e-05	20
True	False	0.018	0.001	7.5
True	False	0.018	0.001	20
True	False	0.018	5e-05	7.5
True	False	0.018	5e-05	20
True	False	0.006	0.001	7.5
True	False	0.006	0.001	20
True	False	0.006	5e-05	7.5
True	False	0.006	5e-05	20
False	False	0.054	0.001	7.5
False	False	0.054	0.001	20
False	False	0.054	5e-05	7.5
False	False	0.054	5e-05	20
False	False	0.018	0.001	7.5
False	False	0.018	0.001	20
False	False	0.018	5e-05	7.5
False	False	0.018	5e-05	20
False	False	0.006	0.001	7.5
False	False	0.006	0.001	20
False	False	0.006	5e-05	7.5
False	False	0.006	5e-05	20

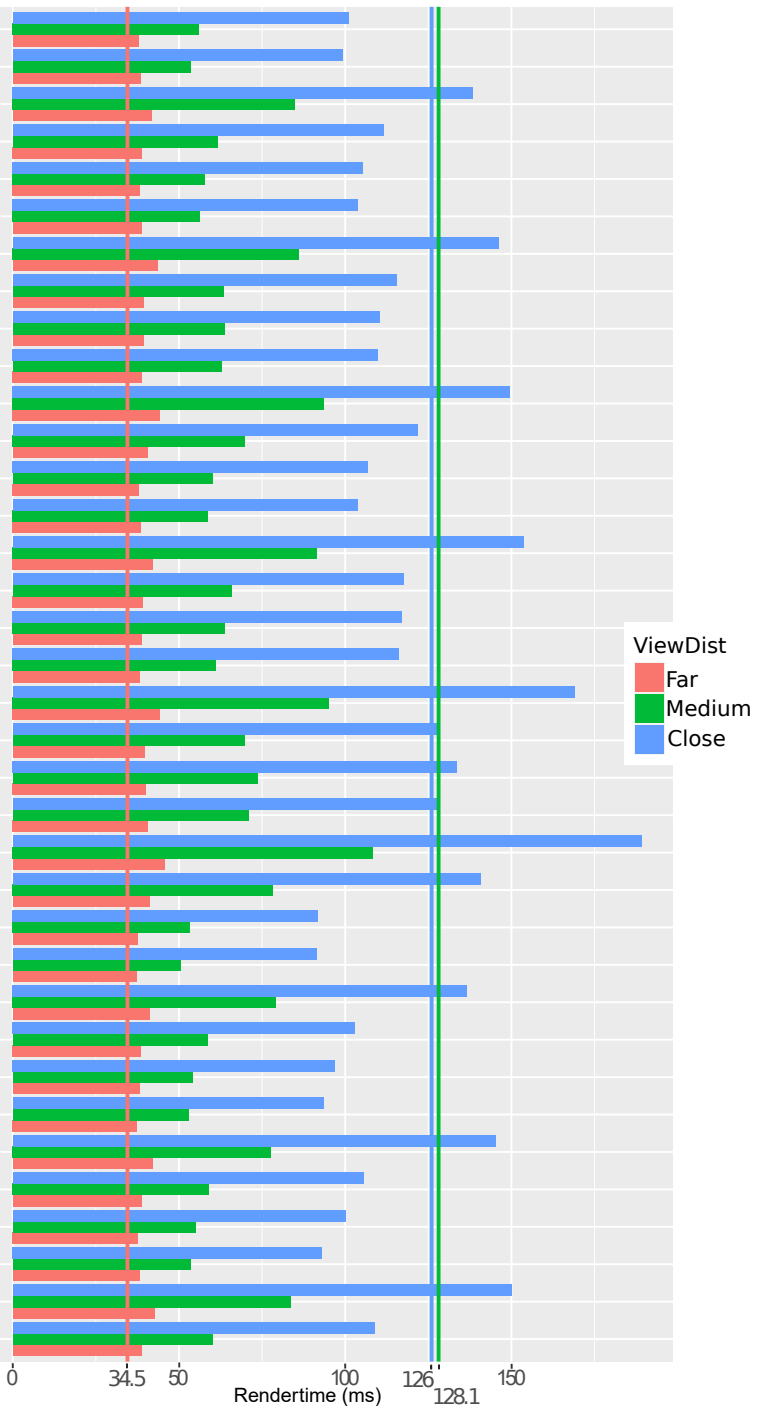

Figure 6.4: Bar chart showing rendering times of all parameter combinations for the graffiti dataset. Reference values of a sphere-based rendering approach are drawn as colored lines. Each color represents a respective viewing distance. Parameter values for each configurations are visualized in the table on the left of the image. For medium viewing distances our approach is significantly faster than the compared rendering time of a sphere-based rendering approach using a BVH. Dependent on the used configurations our approach can be below or above the reference value of 126 milliseconds for close distances.

Figure 6.4 shows the evaluation of our method with the different parameter configurations on the y-axis. The baseline for close, medium, and far viewing distances of the BVH render method is drawn as a red, green, and blue line. The red color encodes a far, the green color a medium, and the blue color a close viewing distance. Viewing distances are similarly chosen in the bar chart, encoding the rendering times given a preprocessing parameter combination.

An immediately visible detail is that the rendering time increases for closer viewing distances in our method. Meanwhile, it decreases in comparison to the reference value from medium to close viewing distance. For the close viewing distance, the comparison method achieves slightly better results with 34.5 ms per frame than our method with an average of 39.8 ms. For medium distances, however, our method outperforms the reference value with an average of 67.8 ms as opposed to 128.1 ms. The render times achieved by our method for near but also for medium distances are highly dependent on the selected parameter space. For example, our method with a non-tessellated mesh, low-resolution textures, and high radius, as well as high angle thresholds for the region growing algorithm, took 91.3 ms to render for a near viewing distance. In contrast, many configurations with a highly tessellated mesh remained above the baseline of 126 ms. All in all, rendering times increased when a lower normal estimation radius or angle threshold was picked. Furthermore, higher rendering time occurred when using tessellation while not discarding invisible triangles. When using lower-resolution textures rendering times lowered especially for tessellated meshes.

For the bike dataset similar to the graffiti dataset, the evaluation results for far viewing distances are slightly higher than the comparative value of 36.1 ms. The measured values can be observed in Figure 6.5. This difference disappears for the medium viewing distance. Here, our method is faster than the reference value 91.5 ms with an average of 80.4 ms. However, the percentual advantage is noticeably smaller than for the graffiti dataset. For close viewing distances, our approach shows significantly higher render times than the comparison value. Only tessellated configurations in which transparent triangles are discarded provide slightly worse values than the baseline. All other configurations exceed the given comparison value. In contrary to the graffiti dataset, non-tessellated configurations did not provide the lowest render time. Instead tessellated meshes with removed transparent vertices were the most efficient.

6.5 Memory Usage

In addition to the runtime performance, memory requirements are an important factor when visualizing large datasets. When the dataset is too large to fit the RAM or VRAM capacity, costly data transfers result. Therefore, we measured the memory consumption of every configuration. In more detail, we recorded the entire memory usage of the MegaMol application given the preprocessed datasets. In this way, in addition to the memory requirements of the raw dataset, the RAM usage of further factors were taken into account. These are for example the overhead of the OSPRay data structure and the memory usage to run our plugin and the MegaMol application.

Figure 6.6 visualizes the required RAM of each configuration as a bar chart. All in all, our approach shows an advantage compared to a BVH consisting of point data. As a comparison, the BVH-tree required 3754 MB of memory. This value is 22.4 times larger than the configuration with the lowest memory consumption, which requires 174 MB of memory. However, even the maximum value of our approach remains with 843MB clearly below the reference value.

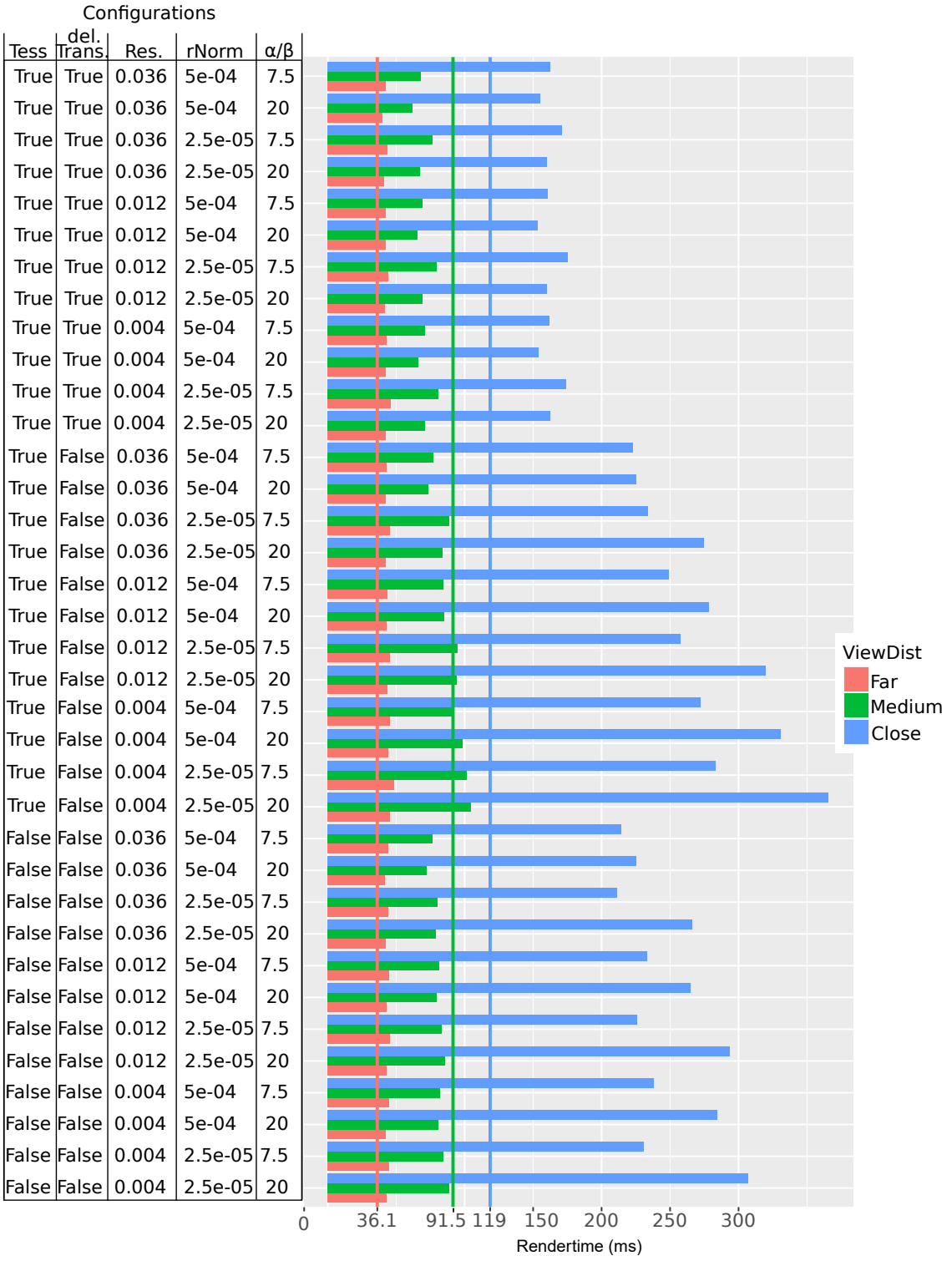

Figure 6.5: Render time in milliseconds for each parameter configuration shown as a bar chart. Furthermore, the orange, green, and blue lines encode rendering times of a sphere-based rendering approach using a BVH. While rendering times for far and medium viewing distances are similar, the reference method shows better rendering times for close views.

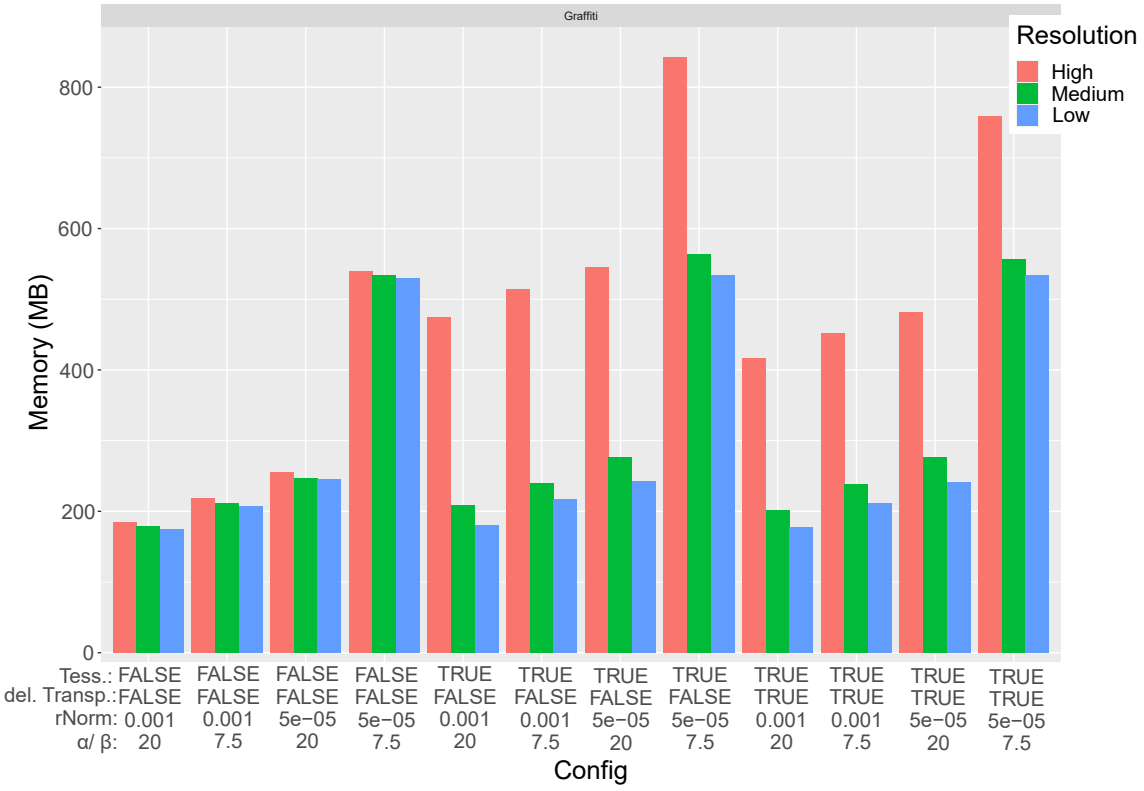

Figure 6.6: Memory requirements of the graffiti dataset for different preprocessing configurations. With an increasing texture resolution the required memory increases. This effect is amplified especially for high-resolution textures when tessellating the mesh. The dataset with the lowest thresholds for the point cloud segmentation requires significantly more memory than the other configurations.

Since a BVH based point rendering method allows fast render times but has significant overhead in additional memory requirements, we additionally measured the memory consumption of a pkd-tree for our two datasets. Here, our approach stays below the 962 MB memory requirement of the pkd-tree.

Comparing the different configurations of our approach, the non-tessellated approach requires the least amount of memory. Furthermore, the memory requirements increase significantly less between the different texture resolutions than for the tessellated methods. Here, especially high-resolution textures increase memory consumption. Additionally, deleting invisible triangles in the tessellated mesh reduces the memory requirements further. Evaluating the parameters for the point cloud segmentation method, the approach with the lowest angle and radius limit consumes significantly more memory than all other approaches.

Furthermore, our approach provides better memory consumption values for the bike point dataset than the BVH or pkd-tree variant. However, the compression rate remains below that of the graffiti dataset. The configuration with the lowest memory capacity of 279 MB is 6.6 times smaller than the reference value. On the other hand, the combination with the largest memory consumption of 533 MB is still 3.5 times smaller than the comparison. Again all measured values additionally lie below the memory consumption of the pkd-tree with 640 MB.

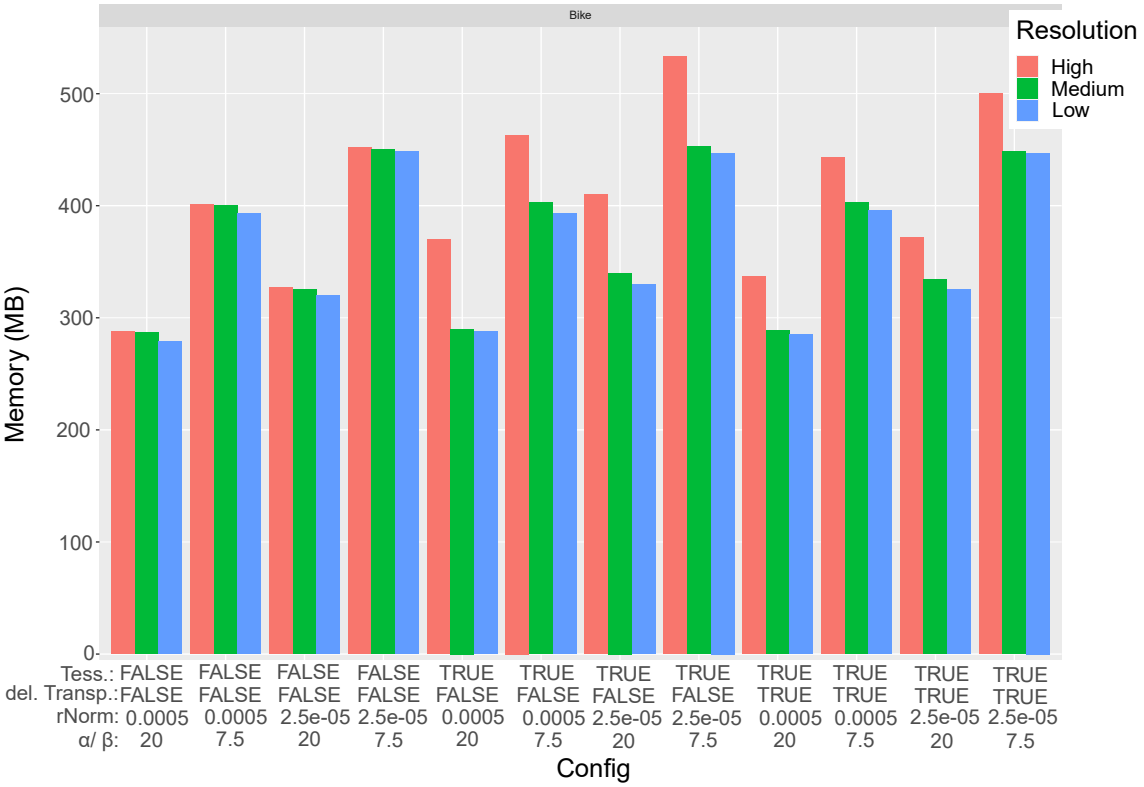

Figure 6.7: Memory usage of the bike dataset given parameter values of our processing pipeline. Similar to the graffiti dataset memory usage increases with higher resolution textures. Compared to the graffiti dataset leads tessellation to a lower rise in memory. Lastly, configurations with an angle threshold of 7.5 degrees require the most storage.

Not tessellating the detected rectangles again provides the representation with the least memory requirements on average. However, contrary to the graffiti dataset, this advantage diminishes for medium- and low-resolution textures. Furthermore, unlike in the Graffiti dataset, the normal calculation has a smaller impact on memory requirements. However, configurations with a high angle threshold still provide the highest compression rates.

Another important reference is to compare the memory requirements needed to store the raw dataset compared to the representatives in our approach. To store a rectangle four vertices, six indices and four uv-coordinates have to be contained in memory. Furthermore, a texture for each rectangle has to be saved. Here, each set of points mapped to the same texel is described by four bytes. Calculating the raw memory requirements of all primitives allows us to give further inside into the memory requirements of our approach. The implemented segmentation algorithm detected 99% of all points when choosing the two higher preprocessing thresholds. Therefore, 25870660 points are replaced by 15 rectangles. Calculating the memory requirements of all points in the dataset for these points, they consume $26130261 * (4 * 1\text{byte} + 3 * 4\text{byte}) = 398.71$ MB of memory. This result is based on the assumption that each color channel requires one byte and all three coordinates of a point are stored in four-byte precision. In contrast, the 15 rectangles consume $15(3 * 4\text{byte} + 4 * 4\text{byte} + 2 * 4\text{byte}) = 540$ byte. In addition, there are 3.96 MB required for unclassified points and 3.75 MB for the high-resolution textures. The consumed memory for low-resolution textures totals

68.5 KB. Therefore, the requirements are up to 99 times lower than when storing the raw point data. It is important to note, that this represents one of the best cases of all of our configurations. However, even for the bike dataset at least 58.6% of all points were detected. This replaces 8575954 points which have a memory requirement of 130.86 MB by 49 rectangles requiring a total of 1.7 KB for rectangles and up to 918KB for the storage of the textures.

6.6 Image Quality

Beyond the previously evaluated technical requirements, image quality serves as an important factor. It is of little advantage for an approach to achieve good render times and low memory usage, when the rendered image deviates too much from the original scene. For this, the representatives should simplify the rendering requirements while still providing a realistic illustration. As common in level of detail data structures, the image quality may decrease for objects further away, since less detail is visible on the screen. To determine the image quality of our approach, a screenshot was taken for each of the configurations evaluated. Because the removal of transparent triangles does not influence the resulting image quality, this parameter was neglected. As a result, there were a total of 18 configurations per dataset. For each of these configurations, a screenshot was taken from a close, medium, and far viewing distance. As reference images, the results were compared with the sphere-based rendering method used previously. Here, each point of the point cloud was rendered as a colored sphere in OSPRay. To compare the results of our method with the reference images the structural similarity index measure (SSIM) [WBSS04] was used. To determine the SSIM value between a screenshot of our approach and the reference image, the SSIM implementation of scikit-image [WSN+14] was utilized. The resulting values indicate the similarity of the two images in percent. A value of one implies two completely matching images. In contrast, low values indicate large differences. The minimum value of minus one implies two completely different images.

An overview of the SSIM values obtained for the Bike dataset can be observed in Figure 6.8. For far viewing distances, near-perfect comparison values were determined for all configurations. The SSIM results range from 0.997 to 0.9999. For medium distances, the SSIM values decreased slightly. Here, the different textures led to the biggest differences. High-resolution textures achieved values between 0.974 and 0.993. Medium-resolution textures, on the other hand, have a maximum SSIM value of 0.985 and a minimum value of 0.964. The lowest results were achieved by low-resolution textures within a range of 0.94 to 0.97. This trend is amplified for the near viewing distance. Additional notable here are the significant differences between the parameters used for the point cloud segmentation method compared to the other viewing distances. All in all, the results vary highly for close viewing distances depending on the configuration. The SSIM values range from 0.719 to 0.965. The computed SSIM values of the graffiti dataset provided comparable values to the results already presented. Merely the tessellation of the triangle mesh led to a higher increase than is the case for the bike dataset. For a close viewing distance and high-resolution texture, this yielded a rise of up to 0.1. The corresponding bar chart can be seen in Figure A.1 in the appendix.

To investigate the origin of the image quality differences in more detail, the SSIM map provided by the scikit-image SSIM-method were analyzed. The maps highlights region, where deviations from the reference image could be detected in the SSIM algorithm. An example of a reference image and the image using our approach, as well as the resulting comparison image can be viewed in Figure 6.9. The displayed bicycle can be seen at a medium viewing distance.

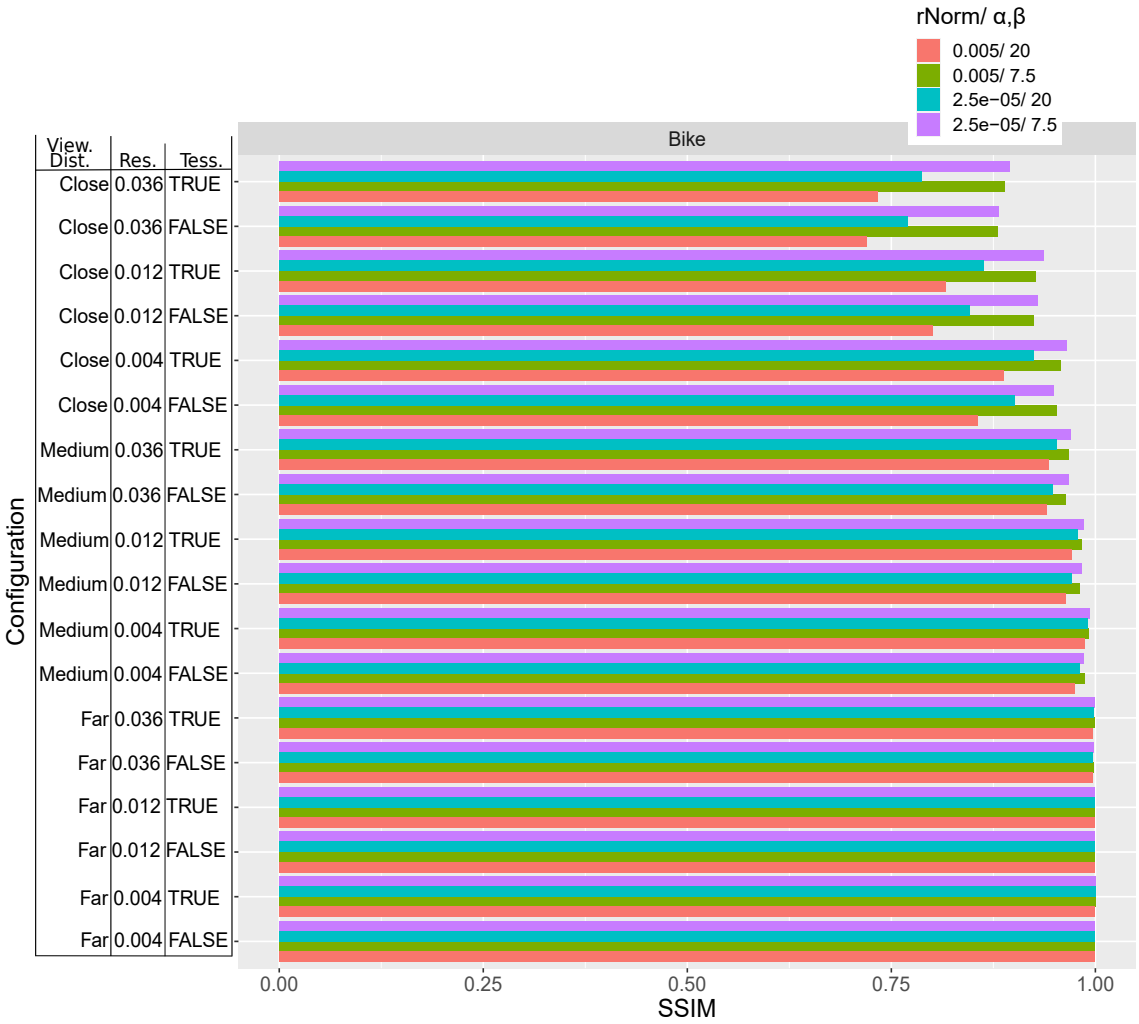

Figure 6.8: In this image, a bar chart is presented showing the structural similarity index measure (SSIM) between configurations of our approach and a reference image. The different configurations of the point cloud segmentation method are encoded by four colors. As the viewing distance lowers, the similarity value decreases. Furthermore, the four different point cloud segmentation configurations have an high impact on the resulting values, especially for the near viewing distance.

The parameters selected for our approach are a radius of 0.000025 for normal calculation, an angular limit of 20 degrees, and a high-resolution texture. Thus, 0.73.5% of all points were replaced by 101 rectangles. Furthermore, the rendered triangles were additionally tessellated and displaced. The calculated SSIM value of this configuration is 0.992. Observing the comparison image, especially the floor color introduced consistent differences. Furthermore, regions with lower point densities, as can be seen on the bottom left and right on the SSIM map, lead to further deviations. As can be seen in Figure 6.10, the results of the graffiti dataset showed similar sources of differences. Here, a low point point density area behind the shown mattress introduced the highest deviations. Furthermore, the edges of the wall with no adjacent geometry caused differences. The respective SSIM value is 0.994.

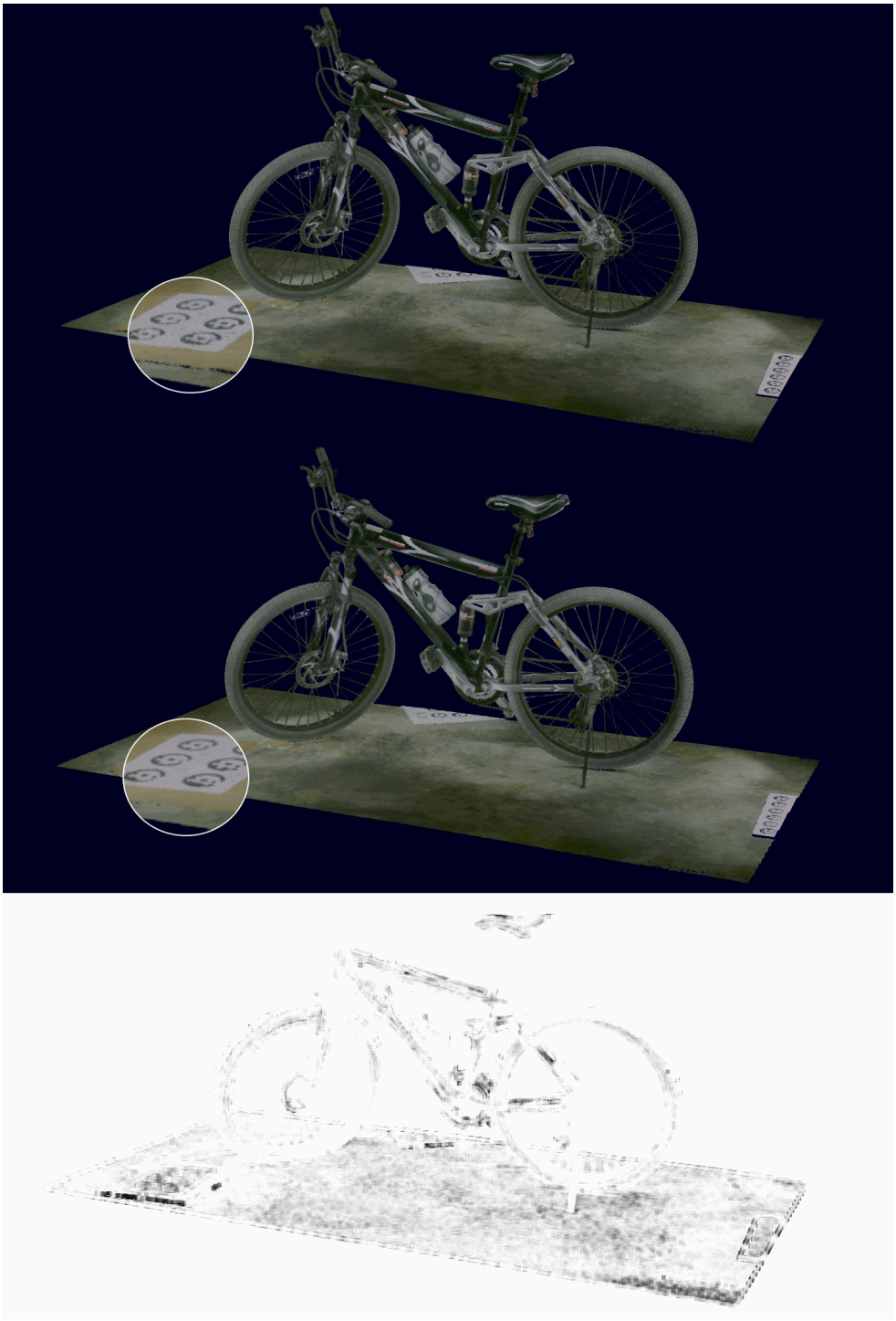

Figure 6.9: This picture shows the difference between a reference image using a sphere-based rendering method on top. The rendered image of our approach is shown in the middle. Regions with deviations found by the SSIM algorithm are indicated in the image at the bottom. Our method replaced 73.5% of all points in the bike dataset with 101 rectangles. Furthermore, high-resolution textures were used and the rectangles are tessellated. Most differences can be identified across the floor and at the edges of the approximated rectangles.

Figure 6.10: The resulting images of the sphere-based reference method (top) and our approach (middle) showing the graffiti dataset. Our approach replaced 99% of all points with 15 rectangles. To increase image quality the rectangles were tessellated and a high-resolution texture was used. The identified differences by the SSIM algorithm can be observed at the bottom image. Most deviations occur in regions with low point density and at the edges of the fitted rectangles.

7 Discussion

In this chapter, the results of the conducted evaluation are reviewed and interpreted. Therefore, possible explanations and reasons of observed effects regarding runtime performance, memory requirements and image quality are provided. Furthermore, limitations and scalability of our approach are discussed.

7.1 Runtime Performance

Observing the render times of the graffiti dataset of our approach, many of the parameter settings yielded the expected effect. Visualizing the rectangles without tessellation showed to deliver the best rendering times. This behavior is expected since the rendering engine has to process viewer triangles. For the same reason, removing invisible triangles when tessellating reduced the overall render times in both the graffiti and bike dataset. In the end, there were only slight differences in the results between the different texture resolutions when the configuration was not tessellated. Since the size of both datasets did easily fit the RAM of the tested system, the time overhead for sampling a high-resolution texture in OSPRay seems to have only a low impact on performance. Nevertheless, we expect a higher impact on performance, if the available memory is more limited for example when using a larger dataset. Furthermore, combining a higher resolution texture with tessellation increased rendering times by a higher factor. This was to be expected since a higher resolution texture refines the tessellation. Thus, more triangles have to be visualized. Lastly, the two parameters affecting our point cloud segmentation method showed an impact on render times. Here, a similarity to the number of classified points can be observed. The more points are left unclassified the more the render time increases. This was to be expected since all unclassified points have to be rendered instead of summarizing them by single planes.

Comparing render times of both datasets revealed differences. Interestingly, the bike dataset achieved not the best rendering times when the mesh was not tessellated. Instead, the method that tessellated the mesh and discarded invisible triangles showed significantly better results. One possible reason for this could be found in the planar areas discovered in the dataset. While the graffiti dataset mainly contains rectangular planar areas, the bike dataset includes more curvy regions. This results in rectangles being filled with fewer points. An example of this is the surface of the tires. Our region growing algorithm discovers a planar surface on the two sides of them. Building a circle shape in the texture the middle, as well as corner areas, are completely invisible. Visualizing this region as a rectangle therefore results in an additional intersection test. This effect is further enhanced when several such shapes overlap. Here, tessellation improves rendering times by eliminating the invisible triangles. Solutions to this problem can also be found in the segmentation algorithm. Being able to detect more shapes than planes, improves the density of points in a representative. For example, the previously mentioned bicycle tire could be described as a torus.

Overall, our approach achieved better results for the graffiti dataset than the bike dataset. A possible explanation for this behavior lies in the results of the point cloud segmentation method. For the bike dataset, each configuration classified fewer points than for the graffiti dataset. As already mentioned, rendering the unclassified point data demands additional processing power than representing a large number of points by a few rectangles. Furthermore, in the graffiti dataset detected planes were larger and covered a wider area. The reason for this is that the bike dataset mostly consists of areas with high curvature and non-planar regions. On the contrary, the representatives of the bike dataset consist of a lot more but smaller rectangles. Here, good parameter values of the region growing algorithm have to be found, balancing between overlapping rectangles and points that are left unclassified. For our configurations, classifying the most points of the dataset still provided the best rendering times. All in all, it is important to notice, that the planarity of the point dataset has a high impact on rendering times for our algorithm. The bike dataset was particularly chosen to represent a worst-case for our segmentation method.

Comparing the results of our approach with the reference values of the sphere-based rendering method showed good results. Especially for medium viewing distances lower rendering times could be achieved. A medium viewing distance describes a camera distance where no point is located outside the viewport. Meanwhile, the area of the space occupied on the screen of the dataset is optimized. Therefore, the highest amount of points have to be rendered in the reference method. This number is significantly reduced by our approach and replaced by a small number of rectangles. When zooming in further, the sphere BVH-tree can be more efficient, since it offers very fast and simple viewport culling operations. Our approach likewise relies on a BVH for storing geometric primitives. However, the sphere primitives are smaller than rectangles, allowing more precise filtering of the dataset. Thereby, the reason that the sphere-based method provides better render times for a close viewing distance can be explained. Furthermore, the sphere-based method yields significantly better results for close viewing distances visualizing the bike dataset. However, here the render times are further slowed down by the previously mentioned overlapping triangles. Especially for planer regions, our approach can still improve render times for close distances at specific configurations even with high-resolution textures. Lastly, even though our approach yields higher rendering times for far camera distance, the observed differences are small. Here, the other factors like memory requirements and image quality could provide a more important point of reference.

It is important to note that the size of the datasets limits the results of our evaluation. The impact of larger datasets for both compared render methods still needs to be explored. Here, our method could offer greater advantages, since significantly fewer primitives have to be processed than in the reference method. Further limitations can be found in the chosen rendering engine. Our proposed data structure had to be modified to match the internal OSPRay architecture. Therefore, classified and unclassified points had to be processed in the same BVH-tree. This can lead to more required intersection tests since many points overlap with the approximated rectangles. Furthermore, a dynamic level of detail could not be implemented, since on the fly tessellation and displacement maps are not supported. While a constant level of detail provides good insight into the performance of certain parameters, it does not reflect an ultimate use case. Dynamic selection of representatives based on camera distance is an important aspect of many existing point render applications. As a result, our performance is only hardly comparable with them. Another important aspect that complicates this comparison is that OSPRay is based on ray tracing. While ray tracing represents light bounces and reflections more realistically, a rasterization-oriented approach implemented on the GPU can provide better render times.

7.2 Memory Requirements

Likewise to the rendering times, the memory requirements of our approach decreased the more points were classified. This is to be expected since individual rectangles replace a large number of points. The different texture resolutions had only a low impact on memory requirements. Due to their low storage requirements compared to other factors, the additional space required is of minor significance. Nevertheless, a finer tessellation leads to a significant memory increase. Especially in the graffiti dataset tessellating the large rectangles had a high impact. On the contrary to the bike dataset. Here, most of the memory requirements are taken by the unclassified points. Therefore, the ratio between tessellation and total memory consumption is lower. Furthermore, the number of additional vertices scales quadratically with the chosen cell size. Thus, the increase from medium to high-resolution textures is significantly higher than it is the case for low to medium textures.

Comparing the memory consumption of our approach with that of the reference method shows the great strength of our approach. A BVH has a significant memory overhead for bounding box and pointer. For a large number of elements, as is the case with point clouds, this overhead exceeds the memory consumption of the raw dataset. Therefore, the comparison between each other has only limited significance. Data structures for large point sets rely on other memory structures like an octree or kd-tree introducing lower overhead. However, our approach still offers a memory advantage over a pkd-tree, which itself has almost no memory overhead. After all, this is the main purpose of representatives in a level of detail data structure. The comparison with other strategies, such as Poisson subsampling would be of interest. However, this is difficult to compare. The compression rate of our approach depends on the spatial properties of a point cloud. If we take the compression rate of the graffiti dataset with high segmentation thresholds, the dataset can be reduced up to a factor of 99. Assuming a similar factor for a dataset that requires 50 GB of memory, it could be reduced to 610 MB. However, while modeling room interiors with many planar areas this may be a good reference value, other structures like trees can only be remodeled to a limited extent. Here, additionally recognizing geometries like cylinders or spheres would provide a further advantage. Furthermore, a level of detail method is solely presented for classified points. Simplifying unclassified points would make our method much less dependent on the spatial structure of the point cloud.

7.3 Image Quality

While SSIM is a good approach to measure the similarity between the two pictures, it introduces some challenges for our use case. Finding a good reference value for the comparison of a screenshot taken from our approach is not a trivial task. Optimally the rendered images are compared with the real-world environment that the captured point cloud represents. However, images would have to be taken from the same camera position. Additionally, influences like sunlight make this approach nearly impossible. We decided to compare our results with the images of the previously presented reference method. Visualizing a point cloud by small spheres creates high-resolution images that are close to the depicted reality. Hereby, the point cloud is displayed with the highest possible resolution. However, especially for close viewing distances, individual spheres may be visible in the image. The displayed points are projected onto the screen as filled circles if they cover more

than one pixel. Nevertheless, this difference only leads to small inaccuracies for medium and close viewing distances. Another advantage is that this difference can be easily tracked by comparing the divergence image displaying region with high dissimilarities.

Overall, we achieved high SSIM values for far and medium viewing distances. For close distances, especially configurations in which many points were replaced showed deviations. This was to be expected since the representatives are the only source of error for our approach. Unclassified points visualized as spheres are creating an identical representation to our reference method. If we detect zero points in our point cloud segmentation, we would obtain an identical image to our reference. However, inferring from SSIM values to the image quality observed by humans is not a simple issue. Flynn et al. [FWAP13] conducted a study observing at which SSIM values participants were able to detect a distorted image. The results showed, that four images were indistinguishable starting at a SSIM value of 0.96. Two other images, on the other hand, could not be separated from the reference image at a SSIM value of 0.92 and above. These values are reached for all far and medium viewing distances in our approach. Furthermore, some configurations achieved these values for close distances. Based on the collected values, it can be concluded that low-resolution textures are sufficient for our chosen far camera distance. Furthermore, for medium viewing distances, the medium texture resolution showed to provide sufficient results. However, high-resolution textures still enable measurable improvements. Due to the significant differences, only the calculated high-resolution textures in our evaluation should be selected for close-up views. Here, the rectangles should be additionally tessellated since it provided a noticeable increase in the SSIM values.

However, one should be careful with the interpretation of the SSIM values. Although the calculated results indicate the differences between the two images, the calculated values provide only limited information about the perceived quality of the images [NA20]. Therefore, we additionally reviewed the created SSIM maps. The SSIM map highlights regions in the image where the algorithm identified deviations from the reference picture. This allows further conclusions about the factors that lead to the measured differences.

Analyzing the SSIM map, we were able to identify variable sources of errors. As seen in the bike dataset, high constant dissimilarities can be detected on the floor where the bicycle stands. Three major sources are contributing to this deviation. First, the texture resolution may not fit the local point density of the point cloud at the represented region. Choosing a texture resolution in our preprocessing, the size of a texel is constant for the whole dataset. However, point clouds can have varying point densities. Being close enough, these differences can be identified if two or more distinguishable spheres are represented by one texel. In addition, a varying point density makes it difficult to choose an optimal cell size into which the points are projected. While the resulting texture resolution may be too coarse for some areas, noticeable artifacts due to undersampling may appear in other regions. To reduce this error, a dynamic texture resolution based on the point density of each rectangle could be calculated in the preprocessing step. However, the error would remain for varying resolutions within a rectangle. Another approach would be to reduce the artifacts of undersampling. For this purpose, the color of texels without points could be interpolated based on the neighboring texels. However, this approach is limited to slightly undersampled textures. Above a certain resolution, it would no longer be possible to distinguish between holes appearing in the sampled environment and undersampled regions.

Another source of error that is limited to close viewing distances is the shape of the representatives. While spheres projected to the screen are represented by a circle, a texel can be recognized as a square. Therefore, differences between both representations can be identified. Conclusions about

which representation reflects the reality more accurately can not be made since this is a basic limitation of a point cloud. However, with modern scanning devices measuring points at one-tenth of a millimeter apart, the use case of such close observation distances is very limited. Here, a point cloud simply encounters limitations.

The last limitation is the color of a texel. In our approach, colors of all points mapped to a texel are interpolated. The interpolated color represents a tradeoff between the color of all represented points. The final resulting color may not even appear in the actual dataset. Visualizing a point cloud with spheres, the color of the spheres are not interpolated. In contrast, only the colors of the spheres are shown, which are not occluded by other spheres. Depending on the viewing angle the visible spheres and colors may vary. We believe this leads to the largest differences between our approach and the reference images. Differences can become visible, especially in the case of strongly locally fluctuating color values. However, it remains to be analyzed how the color differences influence an analysis. For our results, the color difference was hardly noticeable limited to small areas.

Besides differences within the textures, deviations could additionally be detected at the edge of the approximated rectangles. Particularly in images of the Graffiti dataset, these differences are visible. Furthermore, as the resolution decreases, the observed deviations intensify. This points to an issue that was already noticeable when creating the textures. If even edges of the point cloud are not mapped parallel to the border of the texture slight artifacts can be observed. This effect was shown in Figure 6.2. A higher resolution diminishes these artifacts since the size of a texel is smaller. Further improvements can be achieved by optimizing the calculations of the corner points of a rectangle. However, if the detected planar region has more or less than four edges, a more advanced uv-mapping has to be used.

Interestingly, tessellation had limited effects on the calculated SSIM values. On the other hand, the differences were clearly visible for medium and close distances when comparing. We assume this is due to the fact that the human eye can recognize differences in shape more easily than color differences. Unaligned rectangles lead to unnatural forms and empty spaces. We can easily recognize these effects since they can not occur in the real world. Color differences, on the other hand, are often difficult to detect as long as the base color does not change. The detected differences obtained by interpolation in our approach lead solely to different brightness levels. Therefore, we conclude that the tessellation step is very important for the quality of the later result.

All in all, it is hard to tell which configuration provides the best overall performance considering memory requirements, render times, and image quality. Configurations classifying fewer points lead to worse render times and higher storage capacities. Meanwhile, the calculated SSIM value is higher. However, as mentioned above the SSIM does not form a precise indication of the perceived image quality. Significant differences nevertheless give a general indication. Here, it would be of further interest to interview users to form a stronger established opinion. If there are strong similarities with the SSIM value, an automatic selection of the parameters could be implemented optimizing technical requirements and image quality. Furthermore, certain parameter values could be adjusted manually based on user needs. An example of this could be the adjustment of tessellation levels, texture resolutions, or the level of detail shown based on defined camera distances. Likewise, the distance from which the raw point data is displayed to the user is still to be determined. The representation of all points still allows the most detailed and realistic visualization. However, the continuous surface of the approximated surfaces can fill the otherwise empty space between the points. Again, this would require the investigation of possible use cases.

8 Conclusion and Outlook

Rendering point clouds quickly encounters limitations as the amount of data grows. In order to display these datasets on commercial hardware further processing steps have to be conducted. In this work, a processing pipeline was introduced simplifying point clouds by geometrical shapes. In a first step, a normal estimation algorithm based on principal component analysis approximates the normals of the point cloud surfaces. Subsequently, a region growing algorithm estimates planar regions by planes. To be able to render these planes, rectangles are fit into the classified point regions. This is done by reducing the planes into two dimensions and finding minimum and maximum coordinates that can be used as vertices. Considering that point clouds can further contain color information grids are created over the rectangles and points are mapped to respective cells. With each cell describing a texel, textures of variable resolutions can be created. Here, the resolution depends on the arbitrary adjustable cell size. Allowing to create various levels of details, the approximated rectangles can be further refined. Calculated displacement maps describe the deviation of points projected to each cell from the geometrical shape. Afterward, the mesh of each rectangle can be tessellated and each vertex is shifted by the stored displacement value. With increasing mesh refinement a more detailed representation can be achieved. To dynamically choose a level of detail based on the viewpoint, a data structure is introduced. The data structure stores detected geometry in a bounding volume hierarchy. Besides rectangles, each node is designed to be able to store further geometry types like cylinders or spheres. Each leaf node contains indices to an array storing the point data represented by each shape. Thereby for each point subset, a pkd-tree is created and the array containing the point data is sorted accordingly. Besides classified point data, all points not classified by our point cloud segmentation method are processed further. Here, the DBScan algorithm is used to find related dense point regions. Afterward, the found clusters are further spatially subdivided by the median point along the largest bounding box dimension. Finally, each point segment is stored as a leaf node in a bounding volume hierarchy.

In a following evaluation, different parameter values in the processing pipeline were proposed, tested, and analyzed. The resulting datasets have been rendered using the OSPRay ray tracing engine. All results including up to 216 different configurations are presented and analyzed in terms of render times, memory usage, and image quality. Furthermore, all values are compared to a sphere-based rendering approach using a bounding volume hierarchy. Here, our approach shows comparable render times especially for far and medium viewing distances, while providing almost non-distinguishable render images from the reference method. Noticeable improvements were seen in particular for medium viewing distances, where the rendering time could be more than halved for some configurations. A major strength of our approach lies in the reduction of storage requirements. While a point-based bounding volume hierarchy introduces a lot of memory overhead, our approach reduces memory requirements lower than the original dataset. The memory compression rate highly depends on the spatial arrangements of the points. For an indoor scene showing a vacant building up to 99% of all points could be replaced by only fifteen rectangles. Another dataset consisting mainly of regions with high curvature yielded 58.6% to 81.2% of all points being replaced. However, the

resulting many small and overlapping rectangles led to higher rendering times which were especially noticeable at close viewing distances. Comparing the different preprocessing parameters showed visible differences in render times, memory usage as well as image quality. Classifying more points with fewer planes produces the best overall results, however, the dataset must contain distinct planar regions to achieve this. To gain insights into the image quality of the rendered results, images of the sphere-based reference method were compared with our approach. For this, the structural similarity index was calculated. The retrieved results indicated almost non-distinguishable results for far and medium viewing distances. Additionally, SSIM maps were compared to further investigate sources of errors. Here, most deviation resulted from small color differences and texture resolutions not matching local point densities.

Outlook

Right now there are limiting factors regarding the current implementation and evaluation of our approach. With the OSPRay rendering engine, dynamic tessellation, as well as displacement mapping, is not supported. The precalculated constant level of detail only partially represents a later use case. Here, a rendering approach based on the GPU can be implemented to achieve on-the-fly tessellation and displacement mapping based on the viewing distance. Furthermore, rendering times could be increased compared to CPU-based implementations. Likewise, the significantly reduced memory requirements of our method could once again speed up render times considerably in comparison to visualizing the whole point cloud.

This approach could be additionally improved by exploring different point cloud segmentation methods. While the currently used region growing algorithm provides reasonable results, the large parameter space, as well as possible overlapping segmented regions, can affect the later visualization. Furthermore, the detection of additional geometrical shapes like cylinders or spheres can increase the number of classified points as well as decrease the average deviation of all points from the fitted geometry. Likewise, the representation of planer points by tensor product surfaces could better fit non-rectangular shapes and therefore decrease overlap and intersections between the fitted planes.

In this work, the presented level of detail approach is limited to the classified points of the point cloud segmentation method. Although unrecognized points are processed, they are all rendered in the final view, if they are not occluded. A further approach aggregating these point regions with high curvature could reduce rendering times and memory requirements based on the distance to the viewport. This would additionally lead to a more consistent representation in which equally distant objects are visualized uniformly.

Finally, the lowest level of detail in our approach is achieved by visualizing each planar region by one rectangle. These rectangles are stored in the leaf nodes of a BVH-tree. However, the representatives of the nodes can be further aggregated by their parent nodes. This allows to simplify two or more detected geometries, enabling a higher compression rate. Nevertheless, finding a good method to compute these representatives requires a more advanced heuristic aggregating nodes in the BVH-tree. Furthermore, the technique has to incorporate the color information of the point cloud.

Bibliography

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Y. Wu. "An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions". In: *J. ACM* 45.6 (Nov. 1998), pp. 891–923. ISSN: 0004-5411. DOI: 10.1145/293347.293348. URL: https://doi.org/10.1145/293347.293348 (cit. on p. 16).

[Ben75] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007 (cit. on p. 14).

[BHH14] J. Bittner, M. Hapala, V. Havran. "Incremental BVH construction for ray tracing". In: *Computers & Graphics* 47 (Dec. 2014). DOI: 10.1016/j.cag.2014.12.001 (cit. on p. 13).

[BL08] J. M. Biosca, J. L. Lerma. "Unsupervised robust planar segmentation of terrestrial laser scanner point clouds based on fuzzy clustering methods". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 63.1 (2008). Theme Issue: Terrestrial Laser Scanning, pp. 84–98. ISSN: 0924-2716. DOI: https://doi.org/10.1016/j.isprsjprs.2007.07.010. URL: http://www.sciencedirect.com/science/article/pii/S0924271607000809 (cit. on p. 17).

[BLHH] B. Bhanu, S. Lee, C.-C. Ho, T. Henderson. "Range data processing: Representation of surfaces by edges". In: (cit. on p. 17).

[CBC+01] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, T. R. Evans. "Reconstruction and Representation of 3D Objects with Radial Basis Functions". In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 67–76. ISBN: 158113374X. DOI: 10.1145/383259.383266. URL: https://doi.org/10.1145/383259.383266 (cit. on p. 23).

[CC08] J. Chen, B. Chen. "Architectural Modeling from Sparsely Scanned Range Data". In: *Int. J. Comput. Vision* 78.2–3 (July 2008), pp. 223–236. ISSN: 0920-5691. DOI: 10.1007/s11263-007-0105-5. URL: https://doi.org/10.1007/s11263-007-0105-5 (cit. on p. 23).

[CH67] T. Cover, P. Hart. "Nearest neighbor pattern classification". In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27. DOI: 10.1109/TIT.1967.1053964 (cit. on p. 16).

[Cla76] J. H. Clark. "Hierarchical Geometric Models for Visible Surface Algorithms". In: *Commun. ACM* 19.10 (Oct. 1976), pp. 547–554. ISSN: 0001-0782. DOI: 10.1145/360349.360354. URL: https://doi.org/10.1145/360349.360354 (cit. on pp. 13, 30).

[Coo86] R. L. Cook. "Stochastic Sampling in Computer Graphics". In: *ACM Trans. Graph.* 5.1 (Jan. 1986), pp. 51–72. ISSN: 0730-0301. DOI: 10.1145/7529.8927. URL: https://doi.org/10.1145/7529.8927 (cit. on p. 22).

[Del34] B. Delaunay. “Sur la Sphere Vide”. In: *Izvestia Akademia Nauk SSSR* 7 (1934), pp. 793–800 (cit. on p. 23).

[DVS03] C. Dachsbacher, C. Vogelgsang, M. Stamminger. “Sequential Point Trees”. In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 657–662. ISSN: 0730-0301. DOI: 10.1145/882262.882321. URL: https://doi.org/10.1145/882262.882321 (cit. on p. 21).

[EKSX96] M. Ester, H.-P. Kriegel, J. Sander, X. Xu. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: KDD’96. Portland, Oregon: AAAI Press, 1996, pp. 226–231 (cit. on p. 19).

[EMSN12] J. Elseberg, S. Magnenat, R. Siegwart, A. Nuchter. “Comparison on nearest-neigbour-search strategies and implementations for efficient shape registration”. In: *Journal of Software Engineering for Robotics (JOSER)* 3 (Jan. 2012), pp. 2–12 (cit. on p. 16).

[FKN80] H. Fuchs, Z. M. Kedem, B. F. Naylor. “On Visible Surface Generation by a Priori Tree Structures”. In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 124–133. ISSN: 0097-8930. DOI: 10.1145/965105.807481. URL: https://doi.org/10.1145/965105.807481 (cit. on p. 14).

[FWAP13] J. R. Flynn, S. Ward, J. Abich, D. Poole. “Image Quality Assessment Using the SSIM and the Just Noticeable Difference Paradigm”. In: *Proceedings, Part I, of the 10th International Conference on Engineering Psychology and Cognitive Ergonomics. Understanding Human Cognition - Volume 8019*. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 23–30. ISBN: 9783642393594 (cit. on p. 62).

[GBB+19] P. Gralka, M. Becher, M. Braun, F. Frieß, C. Müller, T. Rau, K. Schatz, C. Schulz, M. Krone, G. Reina, T. Ertl. “MegaMol – A Comprehensive Prototyping Framework for Visualizations”. In: *The European Physical Journal Special Topics* 227.14 (Mar. 2019), pp. 1817–1829. ISSN: 1951-6401. DOI: 10.1140/epjst/e2019-800167-5. URL: https://doi.org/10.1140/epjst/e2019-800167-5 (cit. on p. 33).

[GF09] A. Golovinskiy, T. Funkhouser. “Min-cut based segmentation of point clouds”. In: *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. 2009, pp. 39–46. DOI: 10.1109/ICCVW.2009.5457721 (cit. on p. 17).

[GG04] N. Gelfand, L. J. Guibas. “Shape Segmentation Using Local Slippage Analysis”. In: *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. SGP ’04. Nice, France: Association for Computing Machinery, 2004, pp. 214–223. ISBN: 3905673134. DOI: 10.1145/1057432.1057461. URL: https://doi.org/10.1145/1057432.1057461 (cit. on p. 17).

[GHFB13] Y. Gu, Y. He, K. Fatahalian, G. Blelloch. “Efficient BVH Construction via Approximate Agglomerative Clustering”. In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG ’13. Anaheim, California: Association for Computing Machinery, 2013, pp. 81–88. ISBN: 9781450321358. DOI: 10.1145/2492045.2492054. URL: https://doi.org/10.1145/2492045.2492054 (cit. on p. 13).

[GM04] E. Gobbetti, F. Marton. “Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models”. In: *Computers & Graphics* 28 (Dec. 2004), pp. 815–826. DOI: 10.1016/j.cag.2004.08.010 (cit. on p. 21).

[GS87] J. Goldsmith, J. Salmon. “Automatic Creation of Object Hierarchies for Ray Tracing”. In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 14–20. DOI: 10.1109/MCG.1987.276983 (cit. on pp. 13, 30).

[GWG+20] P. Gralka, I. Wald, S. Geringer, G. Reina, T. Ertl. “Spatial Partitioning Strategies for Memory-Efficient Ray Tracing of Particles”. In: *Proceedings of IEEE Symposium on Large-Scale Data Analysis and Visualization*. 2020 (cit. on p. 41).

[GZPG10] P. Goswami, Y. Zhang, R. Pajarola, E. Gobbetti. “High Quality Interactive Rendering of Massive Point Models Using Multi-way kd-Trees”. In: *Pacific Conference on Computer Graphics and Applications* 0 (Sept. 2010), pp. 93–100. DOI: 10.1109/PacificGraphics.2010.20 (cit. on pp. 22, 30).

[HDD+92] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle. “Surface Reconstruction from Unorganized Points”. In: *SIGGRAPH Comput. Graph.* 26.2 (July 1992), pp. 71–78. ISSN: 0097-8930. DOI: 10.1145/142920.134011. URL: https://doi.org/10.1145/142920.134011 (cit. on p. 22).

[HRB05] M. Hubert, P. J. Rousseeuw, K. V. Branden. “ROBPCA: A New Approach to Robust Principal Component Analysis”. In: *Technometrics* 47.1 (2005), pp. 64–79. DOI: 10.1198/004017004000000563. eprint: https://doi.org/10.1198/004017004000000563. URL: https://doi.org/10.1198/004017004000000563 (cit. on p. 18).

[JC16] I. T. Jolliffe, J. Cadima. “Principal component analysis: a review and recent developments”. In: *Philosophical Transactions of the Royal Society of London Series A* 374.2065 (Apr. 2016), p. 20150202. DOI: 10.1098/rsta.2015.0202 (cit. on p. 16).

[KBH06] M. Kazhdan, M. Bolitho, H. Hoppe. “Poisson Surface Reconstruction”. In: *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*. SGP ’06. Cagliari, Sardinia, Italy: Eurographics Association, 2006, pp. 61–70. ISBN: 3905673363 (cit. on p. 23).

[KK86] T. L. Kay, J. T. Kajiya. “Ray Tracing Complex Scenes”. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’86. New York, NY, USA: Association for Computing Machinery, 1986, pp. 269–278. ISBN: 0897911962. DOI: 10.1145/15922.15916. URL: https://doi.org/10.1145/15922.15916 (cit. on p. 13).

[LA13] F. Lafarge, P. Alliez. “Surface Reconstruction through Point Set Structuring”. In: *Computer Graphics Forum* 32.2pt2 (2013), pp. 225–234. DOI: https://doi.org/10.1111/cgf.12042. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12042. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12042 (cit. on p. 23).

[LKS+08] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, C. Jin. “Flexible IO and Integration for Scientific Codes through the Adaptable IO System (ADIOS)”. In: *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*. CLADE 08. Boston, MA, USA: Association for Computing Machinery, 2008, pp. 15–24. ISBN: 9781605581569. DOI: 10.1145/1383529.1383533. URL: https://doi.org/10.1145/1383529.1383533 (cit. on p. 33).

[LZB14] Y. Liang, M. Zhang, W. N. Browne. “Image Segmentation: A Survey of Methods Based on Evolutionary Computation”. In: *Simulated Evolution and Learning*. Cham: Springer International Publishing, 2014, pp. 847–859. ISBN: 978-3-319-13563-2 (cit. on p. 17).

[Mac67] J. Macqueen. “Some methods for classification and analysis of multivariate observations”. In: *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*. 1967, pp. 281–297 (cit. on p. 19).

[MB90] D. J. MacDonald, K. S. Booth. “Heuristics for Ray Tracing Using Space Subdivision”. In: *Vis. Comput.* 6.3 (May 1990), pp. 153–166. ISSN: 0178-2789. DOI: 10.1007/BF01911006. URL: https://doi.org/10.1007/BF01911006 (cit. on p. 13).

[MWP18] C. Mura, G. Wyss, R. Pajarola. “Robust Normal Estimation in Unstructured 3D Point Clouds by Selective Normal Space Exploration”. In: *Vis. Comput.* 34.6–8 (June 2018), pp. 961–971. ISSN: 0178-2789. DOI: 10.1007/s00371-018-1542-6. URL: https://doi.org/10.1007/s00371-018-1542-6 (cit. on p. 34).

[NA20] J. Nilsson, T. Akenine-Möller. *Understanding SSIM*. eng. Tech. rep. arXiv.org, June 2020. URL: https://arxiv.org/abs/2006.13846 (cit. on p. 62).

[NBW12] A. Nurunnabi, D. Belton, G. West. “Robust Segmentation in Laser Scanning 3D Point Cloud Data”. In: *2012 International Conference on Digital Image Computing Techniques and Applications (DICTA)*. 2012, pp. 1–8. DOI: 10.1109/DICTA.2012.6411672 (cit. on pp. 17, 18).

[NBW16] A. Nurunnabi, D. Belton, G. West. “Robust Segmentation for Large Volumes of Laser Scanning Three-Dimensional Point Cloud Data”. In: *IEEE Transactions on Geoscience and Remote Sensing* 54.8 (2016), pp. 4790–4805. DOI: 10.1109/TGRS.2016.2551546 (cit. on p. 36).

[NL13] A. Nguyen, B. Le. “3D point cloud segmentation: A survey”. In: *2013 6th IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. 2013, pp. 225–230. DOI: 10.1109/RAM.2013.6758588 (cit. on p. 17).

[PGK02] M. Pauly, M. Gross, L. Kobbelt. “Efficient simplification of point-sampled surface”. In: vol. 1. Dec. 2002, pp. 163–170. ISBN: 0-7803-7498-3. DOI: 10.1109/VISUAL.2002.1183771 (cit. on pp. 18, 23, 34).

[QSMG17] C. R. Qi, H. Su, K. Mo, L. J. Guibas. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 2017. arXiv: 1612.00593 [cs.CV] (cit. on p. 17).

[RKRE17] T. Rau, M. Krone, G. Reina, T. Ertl. “Challenges and Opportunities using Software-defined Visualization in MegaMol”. In: *7th Workshop on Visual Analytics, Information Visualization and Scientific Visualization*. 2017 (cit. on p. 33).

[RL00] S. Rusinkiewicz, M. Levoy. “QSplat: A Multiresolution Point Rendering System for Large Meshes”. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 343–352. ISBN: 1581132085. DOI: 10.1145/344779.344940. URL: https://doi.org/10.1145/344779.344940 (cit. on p. 21).

[Sch16] M. Schütz. “Potree: Rendering large point clouds in web browsers”. In: (2016) (cit. on pp. 22, 30).

[SRO10] J. Strom, A. Richardson, E. Olson. "Graph-based segmentation for colored 3D laser point clouds". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 2131–2136. DOI: 10.1109/IROS.2010.5650459 (cit. on p. 17).

[SWK07] R. Schnabel, R. Wahl, R. Klein. "Efficient RANSAC for Point-Cloud Shape Detection". In: *Computer Graphics Forum* 26.2 (2007), pp. 214–226. DOI: https://doi.org/10.1111/j.1467-8659.2007.01016.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01016.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01016.x (cit. on pp. 17, 47).

[VD01] G. Vosselman, S. Dijkman. "3D building model reconstruction from point clouds and ground plans". In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* XXXIV (Jan. 2001) (cit. on p. 17).

[Ver18] G. Verhoeven. "Resolving some spatial resolution issues – Part 1: Between line pairs and sampling distance". In: 57 (Oct. 2018), pp. 25–34. DOI: 10.5281/zenodo.1465017 (cit. on p. 25).

[VTLB15] A.-V. Vo, L. Truong-Hong, D. F. Laefer, M. Bertolotto. "Octree-based region growing for point cloud segmentation". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 104 (2015), pp. 88–100 (cit. on pp. 17, 47).

[WA03] M. A. Wani, H. R. Arabnia. "Parallel Edge-Region-Based Segmentation Algorithm Targeted at Reconfigurable MultiRing Network". In: *J. Supercomput.* 25.1 (May 2003), pp. 43–62. ISSN: 0920-8542. DOI: 10.1023/A:1022804606389. URL: https://doi.org/10.1023/A:1022804606389 (cit. on p. 17).

[Wal07] I. Wald. "On Fast Construction of SAH based Bounding Volume Hierarchies". In: Oct. 2007, pp. 33–40. ISBN: 978-1-4244-1629-5. DOI: 10.1109/RT.2007.4342588 (cit. on p. 13).

[WBKP08] B. Walter, K. Bala, M. Kulkarni, K. Pingali. "Fast agglomerative clustering for rendering". In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, pp. 81–86. DOI: 10.1109/RT.2008.4634626 (cit. on p. 13).

[WBSS04] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli. "Image Quality Assessment: From Error Visibility to Structural Similarity". In: *Trans. Img. Proc.* 13.4 (Apr. 2004), pp. 600–612. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.819861. URL: https://doi.org/10.1109/TIP.2003.819861 (cit. on p. 55).

[WJA+17] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gunther, P. Navratil. "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 931–940. ISSN: 1077-2626. DOI: 10.1109/TVCG.2016.2599041. URL: https://doi.org/10.1109/TVCG.2016.2599041 (cit. on p. 33).

[WKJ+15] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, M. E. Papka. "CPU ray tracing large particle data with balanced P-k-d trees". In: *2015 IEEE Scientific Visualization Conference (SciVis)*. 2015, pp. 57–64. DOI: 10.1109/SciVis.2015.7429492 (cit. on pp. 14, 15).

[WS06] M. Wimmer, C. Scheiblauer. "Instant Points: Fast Rendering of Unprocessed Point Clouds". In: *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*. SPBG'06. Boston, Massachusetts: Eurographics Association, 2006, pp. 129–137. ISBN: 3905673320 (cit. on p. 22).

[WSN+14] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: http://dx.doi.org/10.7717/peerj.453 (cit. on p. 55).

[XTZ20] Y. Xie, J. TIAN, X. Zhu. "Linking Points With Labels in 3D: A Review of Point Cloud Semantic Segmentation". In: *IEEE Geoscience and Remote Sensing Magazine* (2020), 0–0. ISSN: 2373-7468. DOI: 10.1109/mgrs.2019.2937630. URL: http://dx.doi.org/10.1109/MGRS.2019.2937630 (cit. on p. 17).

All links were last followed on January 2, 2021.

A Evaluation

In this chapter, additional tables showing resulting data values of our conducted evaluation in Chapter 6 are presented. Additionally, a bar chart is shown, representing the calculated SSIM values of the graffiti dataset.

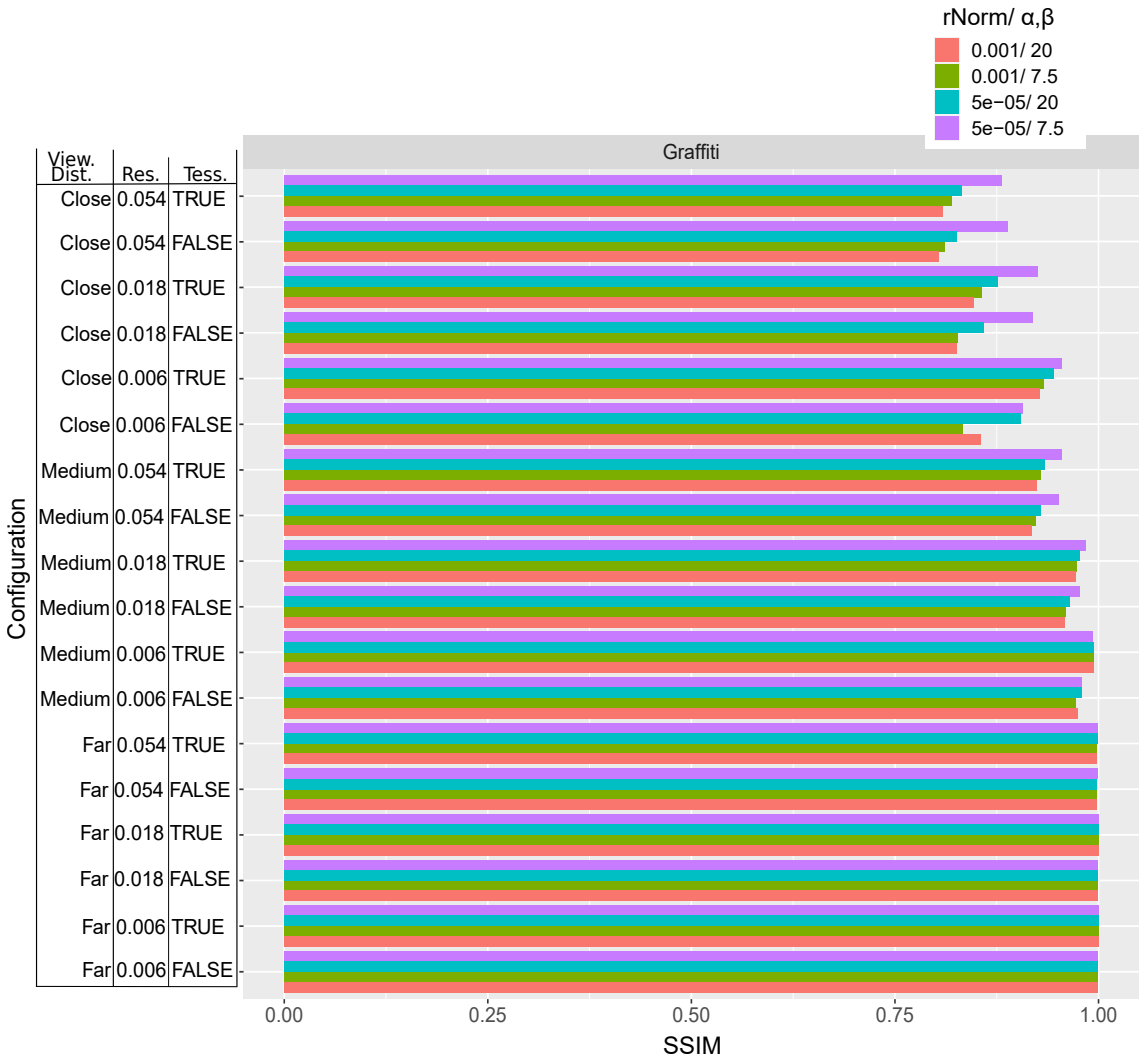

Figure A.1: Bar chart representing the calculated similarity index measure (SSIM) between different preconfigurations of our approach. The parameter values of the configurations are presented in the left table. Furthermore, the different segmentation parameter values are encoded in four colors.

Table A.1: Render times of all configurations in milliseconds for the graffiti dataset. The first four columns encode different preprocessing parameters. Following, the render times of four different point cloud segmentation configurations are displayed.

Tess.	del.Transp	Res	ViewDist	rNorm:0.001 α/β : 7.5°	rNorm:0.001 α/β:20°	rNorm:0.0005 α/β:$7.5°	rNorm:0.0005 α/β : 20°
True	False	0.006	Far	39.373 446	38.901 733	44.250 455	40.490 366
True	False	0.006	Medium	63.673 208	62.768 960	93.606 069	69.820 020
True	False	0.006	Close	110.378 990	109.835 386	149.556 148	121.815 772
True	False	0.018	Far	38.274 970	38.761 159	43.611 940	39.383 030
True	False	0.018	Medium	57.858 792	56.305 604	86.132 644	63.333 584
True	False	0.018	Close	105.298 544	103.847 654	146.187 336	115.494 594
True	False	0.054	Far	37.801 743	38.472 119	41.688 584	38.839 614
True	False	0.054	Medium	56.008 614	53.514 564	84.938 148	61.542 455
True	False	0.054	Close	100.962 971	99.203 069	138.508 080	111.502 238
True	True	0.006	Far	40.078 871	40.641 218	45.598 535	41.148 792
True	True	0.006	Medium	73.547 267	70.885 277	108.406 475	78.316 317
True	True	0.006	Close	133.655 703	128.309 386	189.165 861	140.692 039
True	True	0.018	Far	38.721 396	38.060 515	44.229 040	39.655 089
True	True	0.018	Medium	63.840 218	61.153 188	95.140 950	69.661 376
True	True	0.018	Close	117.074 089	116.202 466	168.959 812	127.433 089
True	True	0.054	Far	38.010 832	38.620 158	42.121 158	39.137 961
True	True	0.054	Medium	60.055 208	58.712 475	91.490 010	65.815 317
True	True	0.054	Close	106.821 258	103.770 119	153.644 228	117.597 059
False	-	0.006	Far	37.736 248	38.181 099	42.616 881	38.741 594
False	-	0.006	Medium	55.014 792	53.410 584	83.479 851	60.086 535
False	-	0.006	Close	100.028 515	93.048 614	150.010 753	108.826 307
False	-	0.018	Far	38.159 307	37.308 267	42.014 643	38.644 911
False	-	0.018	Medium	54.049 337	52.828 921	77.661 228	59.058 703
False	-	0.018	Close	96.891 821	93.446 218	145.271 347	105.471 446
False	-	0.054	Far	37.523 356	37.167 436	41.105 644	38.601 248
False	-	0.054	Medium	53.249 594	50.450 762	79.133 019	58.569 921
False	-	0.054	Close	91.874 832	91.308 475	136.661 712	102.994 059

Table A.2: Render times of all configurations in milliseconds for the bike dataset. The first four columns encode different preprocessing parameters. Following, the render times of four different point cloud segmentation configurations are displayed.

Tess.	del.Transp	Res	ViewDist	rNorm:0.0005 α/β : 7.5°	rNorm:0.0005 α/β:20°	rNorm:0.000025 α/β:$7.5°	rNorm:0.000025 α/β : 20°
True	False	0.004	Far	45.619 228	43.983 475	48.608 099	45.657 168
True	False	0.004	Medium	90.829 802	98.473 931	101.397 376	104.378 594
True	False	0.004	Close	272.177 771	331.001 683	283.421 446	365.768 613
True	False	0.012	Far	43.873 644	42.785 574	45.694 812	43.712 812
True	False	0.012	Medium	84.598 762	85.108 941	95.040 574	94.443 970
True	False	0.012	Close	248.934 980	278.629 682	257.918 762	319.768 307
True	False	0.036	Far	42.709 069	42.271 505	45.191 792	42.452 673
True	False	0.036	Medium	77.385 951	73.562 802	88.697 248	83.886 237
True	False	0.036	Close	222.934 168	225.420 495	233.735 367	274.585 109
True	True	0.004	Far	42.988 822	42.344 356	45.992 713	42.112 010
True	True	0.004	Medium	70.909 555	66.401 119	80.647 089	71.027 466
True	True	0.004	Close	161.725 753	153.951 050	174.162 139	162.523 970
True	True	0.012	Far	42.542 733	42.367 485	43.946 941	41.507 733
True	True	0.012	Medium	69.372 337	65.402 495	79.524 079	69.009 356
True	True	0.012	Close	160.898 337	153.244 663	175.336 812	160.046 030
True	True	0.036	Far	42.339 742	40.094 614	43.610 505	41.143 049
True	True	0.036	Medium	67.963 238	61.982 871	76.660 040	67.442 554
True	True	0.036	Close	162.257 436	155.392 911	170.779 435	159.877 148
False	-	0.004	Far	44.724 425	42.331 970	44.828 010	43.265 653
False	-	0.004	Medium	82.096 386	80.606 089	84.527 842	88.432 861
False	-	0.004	Close	237.833 564	284.479 059	230.668 782	306.789 823
False	-	0.012	Far	44.611 980	42.944 257	45.239 564	42.969 743
False	-	0.012	Medium	81.217 753	79.871 337	83.345 871	85.582 178
False	-	0.012	Close	233.102 118	265.209 752	225.906 505	293.672 742
False	-	0.036	Far	44.000 149	41.949 564	43.957 545	42.154 030
False	-	0.036	Medium	76.399 565	72.393 198	80.335 396	78.862 782
False	-	0.036	Close	214.404 654	225.618 078	211.030 486	266.294 049

Table A.3: Memory usage of all configurations in megabyte for the graffiti dataset. The first four columns encode different preprocessing parameters. Following, the render times of four different point cloud segmentation configurations are displayed.

Tess.	del. Transp	Res	rNorm:0.0005 α/β : 7.5°	rNorm:0.0005 α/β:20°	rNorm:0.000025 α/β:\$7.5°	rNorm:0.000025 α/β : 20°
True	False	0.006	514	475	843	545
True	False	0.018	240	208	564	276
True	False	0.054	217	180	534	242
True	True	0.006	452	416	759	481
True	True	0.018	238	202	557	276
True	True	0.054	211	178	534	241
False	-	0.006	218	185	540	255
False	-	0.018	212	179	534	247
False	-	0.054	207	175	530	245

Table A.4: Memory usage of all configurations in megabytes for the bike dataset. The first four columns encode different preprocessing parameters. Following, the render times of four different point cloud segmentation configurations are displayed.

Tess.	del. Transp	Res	rNorm:0.0005 α/β : 7.5°	rNorm:0.0005 α/β:20°	rNorm:0.000025 α/β:\$7.5°	rNorm:0.000025 α/β : 20°
True	False	0.004	463	370	533	410
True	False	0.012	403	290	453	340
True	False	0.036	393	288	447	330
True	True	0.004	443	337	500	372
True	True	0.012	403	289	448	334
True	True	0.036	396	285	447	325
False	-	0.004	401	288	452	327
False	-	0.012	400	287	450	325
False	-	0.036	393	279	448	320

Table A.5: Results of the image quality evaluation for the graffiti dataset. SSIM is used to compare images of our approach with a sphere-based reference method. The first four columns encode different preprocessing parameters. Following, the SSIM values of four different point cloud segmentation configurations are displayed.

Tess.	Res	ViewDist	rNorm:0.001 α/β : 7.5°	rNorm:0.001 α/β:20°	rNorm:0.0005 α/β:\$7.5°	rNorm:0.0005 α/β : 20°
True	0.006	Far	0.999 910	0.999 904	0.999 926	0.999 931
True	0.006	Medium	0.993 605	0.994 231	0.993 316	0.994 427
True	0.006	Close	0.932 511	0.927 358	0.955 404	0.945 465
True	0.018	Far	0.999 717	0.999 701	0.999 824	0.999 780
True	0.018	Medium	0.972 921	0.971 593	0.984 779	0.977 241
True	0.018	Close	0.856 893	0.846 314	0.925 273	0.876 153
True	0.054	Far	0.998 404	0.998 268	0.998 964	0.998 566
True	0.054	Medium	0.928 586	0.924 073	0.954 905	0.934 338
True	0.054	Close	0.819 576	0.808 490	0.880 791	0.831 513
False	0.006	Far	0.999 418	0.999 307	0.999 581	0.999 447
False	0.006	Medium	0.972 008	0.974 780	0.978 927	0.979 057
False	0.006	Close	0.833 243	0.854 753	0.906 471	0.904 552
False	0.018	Far	0.999 333	0.999 147	0.999 615	0.999 325
False	0.018	Medium	0.959 616	0.958 102	0.976 503	0.965 183
False	0.018	Close	0.826 990	0.826 242	0.918 684	0.858 515
False	0.054	Far	0.998 074	0.997 811	0.998 848	0.998 174
False	0.054	Medium	0.922 862	0.918 131	0.951 210	0.928 484
False	0.054	Close	0.811 414	0.803 236	0.888 181	0.825 668

Table A.6: Results of the image quality evaluation for the bike dataset. SSIM is used to compare images of our approach with a sphere-based reference method. The first four columns encode different preprocessing parameters. Following, the SSIM values of four different point cloud segmentation configurations are displayed.

Tess.	Res	ViewDist	rNorm:0.0005 α/β : 7.5°	rNorm:0.0005 α/β:20°	rNorm:0.000025 α/β:\$7.5°	rNorm:0.000025 α/β : 20°
True	0.004	Far	0.999 882	0.999 739	0.999 936	0.999 903
True	0.004	Medium	0.991 547	0.986 947	0.993 244	0.991 080
True	0.004	Close	0.956 951	0.888 188	0.965 275	0.925 018
True	0.012	Far	0.999 657	0.999 151	0.999 725	0.999 530
True	0.012	Medium	0.983 630	0.970 796	0.985 665	0.978 462
True	0.012	Close	0.927 463	0.816 895	0.937 227	0.863 396
True	0.036	Far	0.998 672	0.996 889	0.998 736	0.997 632
True	0.036	Medium	0.966 854	0.942 613	0.970 023	0.952 615
True	0.036	Close	0.888 650	0.733 917	0.894 496	0.786 995
False	0.004	Far	0.999 761	0.999 276	0.999 725	0.999 590
False	0.004	Medium	0.987 146	0.974 426	0.985 566	0.981 169
False	0.004	Close	0.952 840	0.855 458	0.948 577	0.900 924
False	0.012	Far	0.999 570	0.998 756	0.999 612	0.998 976
False	0.012	Medium	0.981 145	0.963 524	0.982 996	0.970 963
False	0.012	Close	0.924 310	0.800 792	0.929 539	0.846 567
False	0.036	Far	0.998 426	0.996 584	0.998 563	0.996 960
False	0.036	Medium	0.964 132	0.940 145	0.967 486	0.948 131
False	0.036	Close	0.880 816	0.719 881	0.880 986	0.769 806

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature