FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interactive Shape Detection in Out-of-Core Point Clouds for Assisted User Interactions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Bernhard Rainer, BSc.
Matrikelnummer 0828592

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Mag. Michael Schwärzler

Wien, 12. Oktober 2017

Bernhard Rainer	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interactive Shape Detection in Out-of-Core Point Clouds for Assisted User Interactions

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Bernhard Rainer, BSc.
Registration Number 0828592

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Mag. Michael Schwärzler

Vienna, 12th October, 2017

Bernhard Rainer	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Bernhard Rainer, BSc.
Heigerleinstraße 53/8, 1170 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Oktober 2017

Bernhard Rainer

Acknowledgements

This thesis would not have been possible without the help of a handful of engaged people. My first and foremost gratitude goes to the VRVis Zentrum für Virtual Reality und Visualisierung for providing me with the opportunity to do an internship and in consequence conduct this thesis. Within the VRVis, I would like to thank the Semantic Modeling and Acquisition (SMAQ) group for integrating me into their team, both on a professional level and amicably. A special thank you goes out to my Aardvark and F# Gurus Harald Steinlechner, Georg Haaser, and Attila Szabo, whose help and expertise eased the execution of this thesis enormously. I would also like to thank Stefan Maierhofer, leader of the SMAQ group, and my project manager Michael Schwärzler for their ideas, continuous support, and proof-reading this thesis. A big thank you goes out to Michael Wimmer from the Technische Universität Wien for his supervision on not only this thesis but all preceding courses and practica.

Thanks to Lisa Kellner, who accompanied me through this thesis on a daily basis as she sat next to me and continuously provided me with feedback on my work.

Thanks to Phillip Erler, a fellow diploma student, for all the exchange of knowledge on both our theses.

Last but not least, I would like to thank my friends and family, who encouraged me to pursue an academic career and endured my ongoing talks about point clouds and shape detection for the bigger part of a year.

This work was enabled by the Competence Centre VRVis. VRVis is funded by BMVIT, BMWFW, Styria, SFG and Vienna Business Agency in the scope of COMET - Competence Centers for Excellent Technologies (854174) which is managed by FFG.

Abstract

This thesis presents a semi-automated method for shape detection in out-of-core point clouds. Rather than performing shape detection on the entire point cloud at once, a user-controlled interaction determines the region that is to be segmented next. By keeping the size of the region and the number of points small, the algorithm produces meaningful results within a fraction of a second. Thus, the user is presented immediately with feedback on the local geometry.

As modern point clouds can contain billions of points and the memory capacity of consumer PCs is usually insufficient to hold all points in memory at the same time, a level-of-detail data structure is used to store the point cloud on the hard disc, and data is loaded into memory only on use. This data structure partitions the point cloud into small regions, each containing around 5000 points, that are used for rendering and shape detection.

Interacting with point clouds is a particularly demanding task. A precise selection of a region of interest, using the two-dimensional lasso interaction, often needs multiple view changes and subsequent improvements. This thesis proposes improvements to the lasso interaction, by performing selection only on the set of points that are approximated by a detected shape. Thus, the selection of undesired points in the fore- and background is reduced. Point picking is improved as well by the use of a detected shape, such that only points that are approximated by this shape are pick-able.

The result of this thesis is an application that allows the user to view point clouds with millions of points. It also provides a novel interaction technique for quick local shape detection as well as shape-assisted interactions that utilize this local semantic information to improve the user's workflow.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In recent years, multiple acquisition devices and methods of point clouds from real objects have emerged, such as laser scanners, LIDAR, Microsoft Kinect, or photogrammetric reconstructions. The fields of applications for point clouds include, but are not limited to, documenting geomorphological erosion, monitoring urban and agricultural developments, mapping archeological sites, and generating assets for the entertainment industry. The acquisition techniques produce highly detailed point clouds that contain several millions of points. This enormous data resolution presents several challenges to both the system and the user.

The size of point-cloud datasets has increased at such a rapid rate that they are now simply too large to fit into system memory, let alone graphics card memory. Therefore, new solutions for out-of-core representations have emerged. In most of these solutions, the point cloud data is cached in one or more structured files on the hard drive and can therefore not be accessed directly. Based on a culling heuristic, chunks of point-cloud data are loaded into memory as needed. This continuous swapping of data yields the disk speed as a potential bottleneck when it comes to performance. However, it also introduces the benefit of only storing chunks of data in memory that are of immediate interest to the user.

Point-cloud datasets commonly lack structure and contain a lot of unneeded data. Thus, additional processing is required to enrich the data set with semantic information. To improve data quality, additional postprocessing steps must be performed by the user manually, such as removing imperfect regions, extracting regions of interest. However, achieving this task by using classic two-dimensional interaction metaphors can be tedious and cumbersome as the system cannot predict the desired boundaries of the third dimension of the interaction. Without the use of more semantic information, such as the

geometric shape of the region of interest, multiple view changes might be necessary to only select desired regions from the three-dimensional scene.

A way of introducing semantic information into unstructured point-cloud data is shape detection. The objective is to find regions in point clouds with similar characteristics, such as local curvature and neighborhoods, to help the user understand local and global structures. Current solutions, as presented by Schnabel et al. [SWK07a, SWK07b], can already produce a precise segmentation of point clouds. The complexity of these algorithms increases with the size of the point cloud, making the computation for billions of points infeasible in real time. However, when looking at raw numbers, the approach delivers promising results in a fraction of a second for point clouds of smaller size (<12,000 points).

This thesis proposes a user-controlled technique for shape detection in small local regions of point clouds. This approach helps to deliver semantic information on the local geometry in interactive time. This information is used to improve ordinary two-dimensional interaction metaphors by limiting the set of points available for interaction to those that are approximated by the selected shape (i.e. closely follow the curvature of the shape).

1.2 Problem Definition

Modern point clouds can contain millions of points with sizes often exceeding several gigabytes. Usually, consumer PCs do not have the memory capacity to hold the entire point cloud in system memory or video memory. Efficient out-of-core solutions for point clouds are discussed in numerous publications, Scheibelbauer [Sch14], Elseberg et al. [EBN13] or the Point Cloud Library [RC11], to only name a few. However, a custom solution is needed that stores point clouds enriched with semantic information.

In scans of urban environments, many structures can be represented in a more memory-efficient way. Points that follow a wall can often be compressed to few triangles. Pillars often share similarities with cylinders. Detecting such shapes is an immense task that scales with the number of points and fails to be executed in real time. When exploring a point cloud, immediate feedback of local geometry is useful, since it introduces additional information to the user. This task cannot be achieved without substantial postprocessing of the point cloud.

Common two-dimensional interaction metaphors (e.g. mouse) are useful tools when selecting or picking regions from a two-dimensional context. When porting these techniques to 3D, the third dimension (i.e. depth) must be guessed or controlled separately. A region of interest is usually a set of points that are spatially neighboring and create a structural element in the point cloud. Ideally, the selection of such a region is performed by defining a minimal enclosing volume that contains all points. Achieving this selection by using 2D-interaction metaphors only is challenging, as the techniques do not know the desired depth boundaries of the selection region. Therefore, interactions across multiple views are needed to achieve this selection. Methods that use 3D information, such as the

volumetric brush presented by Weyrich et al. [WPK+04], can ease selection tasks. By consulting the depth buffer each frame, the brush follows the curvature of the furthermost geometry. However, by reading pixels from the GPU, the rendering process is stalled. This technique reacts to occlusions such that the brush follows the geometry depicted in the depth buffer, rather then the desired structure. Thus, view changes are still required.

1.3 Contributions

The main contribution of this diploma thesis is the implementation of a semi-automated procedure to detect shapes in multiple levels of detail in point clouds. Instead of performing shape detection on the entire point cloud at once, our approach lets the user control the region in which shapes should be detected. By reducing those regions to a suitable size, a well-known shape-detection algorithm can return meaningful results in interactive time, such that the user is presented with immediate feedback on the local geometry of the selected region.

The size of detected shapes is limited to the region in which it was detected. A clustering algorithm finds shapes from different regions and levels of detail, based on a similarity heuristic, and creates a larger connected cluster of shapes. This cluster is used to present geometric information on a larger scale to the user, rather than each shape separately.

This thesis proposes several improvements to commonly known user interactions. *Point picking* and *region selection* are improved by consulting the local geometry of the point cloud to assist the user. By using a shape as support, the interaction dimensions are reduced to the parameter space of the shapes, allowing the user to exclude unwanted points from interactions easily. Additionally, a novel interaction technique is introduced that allows the user to increment the level of detail locally along a shape. This helps the user to explore the structure of the point cloud in more detail.

1.4 Structure of the Work

Chapter 2 covers the related work for this thesis, including point-cloud rendering and out-of-core representations, shape detection, and segmentation and advanced user interactions on point clouds. Chapter 3 describes the octree used for the out-of-core representation of the point cloud, as well as some metrics that further describe the content of an octree node. Chapter 4 describes the algorithms used to detect primitive shapes in a point cloud and proposes a technique to cluster similar shapes into one coherent shape cluster for user interactions. Chapter 5 discusses the application's features, including the user-controlled shape detection and assisted interactions that utilize the detected shapes as support shape. Chapter 6 focuses on implementation details in a functional context. Results of the application are presented in Chapter 7 along with performance benchmarks of the presented interactions. Chapter 8 concludes this thesis with a reflection on the application and an outlook on future work.

CHAPTER 2

Related Work

The scope of this thesis is on management of massive point clouds, shape detection in, and interactions with point clouds. This chapter discusses work that has inspired this thesis. Section 2.1 presents related work on storing and rendering out-of-core point clouds. Different rendering techniques for point-based graphics are discussed in Section 2.2. Section 2.3 provides different techniques for shape detection in point clouds and further processing. Related work on interactions is presented in Section 2.4.

2.1 Out-of-core Point-Clouds

As the size of modern point clouds often exceeds the available memory, specialized data structures and rendering techniques are needed that can handle such amounts of data.

The QSplat system [RL00] was one of the earliest systems that were capable of handling datasets with well over hundred million points. It uses a hierarchy of bounding spheres that makes it easy to perform visibility culling and level-of-detail control. Each node only contains information on the bounding sphere, not the points itself. Leaf nodes represent a single point sample. Hence, the bounding sphere is the point itself. A node is rendered if it is a leaf or the benefit of traversing to the children is too low; otherwise, the children are traversed. If a node is rendered, a spherical splat is drawn as the node's bounding sphere.

Gobbetti and Marton [GM04] propose a multi-resolution approach for rendering massive point clouds, called Layered Point Clouds (LPC). The point cloud is stored in a binary tree in chunks of approximately the same size. The multi-resolution model contains the same points as the point cloud, but grouped into chunks and organized in a level-of-detail representation. The root of the tree contains a subset of uniformly distributed samples of the point cloud. The remaining points are divided among the two subtrees that are further partitioned until the number of points is below a threshold value. Compared

to the QSplat system, LPC reduces the cost of traversal on the CPU-side significantly and benefits from the parallel architecture of the GPU. This approach hides out-of-core latency by speculatively fetching data.

Dachsbauer et al. [DVS03] introduced Sequential Point Trees (SPT), a data structure that allows adaptive rendering of point clouds completely on the graphics card. A hierarchical point tree is not suited for fast vertex array-based sequential processing by the GPU. Therefore, the point tree's nodes are rearranged into a list, sorted by depth. Rendering then only needs to draw the first p points, where p is controlled by a level-of-detail decision. While their technique allows for adaptive rendering purely on the GPU, it only allows for a single level of detail at a time and can only contain a limited number of points.

Wimmer and Scheiblauer [WS06] introduce nested octrees to create a multi-resolution model similar to LPCs. A nested octree consists of an outer octree that is used for visibility culling and a memory-optimized SPT as inner octree for efficient point rendering. Each node contains a subsample of points from the point cloud. The union of the points from level 0 to a level k creates the level-of-detail representation of the point cloud for level k.

Besides rendering and storing, modifications are a key task for out-of-core point cloud systems. Wand et al. [WBB+07] describe an out-of-core octree that also stores a multi-resolution model of the point cloud. The basic idea is to store downsampled point clouds in the inner nodes where each inner node either contains randomly chosen or averaged points from its children. Hence, this octree stores additional points per level of detail. Points are inserted bottom-up, and the multi-resolution model is updated on the way up. Removing points works similarly. The point is deleted from the leaf first, and the ancestor nodes are updated afterward.

An improvement to nested octrees is the Modifiable Nested Octree (MNO) by Scheiblauer and Wimmer [SW11]. The MNO enhances the nested octree by improving the performance of insertion and deletion operations. A grid replaces the SPTs from the inner nodes. When inserting a new point, an empty grid cell that contains the point is searched. When removing a point from the hierarchy, holes can occur. Therefore, points from a leaf node that intersects the grid cell of the removed point are pulled up. MNO is paired with an out-of-core point-cloud editing system. Since selecting points in an out-of-core setting is not trivial, the authors propose a structure, the so-called Selection Octree to store the selected points separately. This structure is a tool that allows to interactively change the visualization model without actually having to modify the original model permanently.

Until now, the discussed approaches focused on storage and rendering of point clouds. Wenzel et al. [WRFH14] propose an out-of-core octree system that is tailored towards quick data updates and memory management. Parallel to the octree containing the point cloud, a cycle stack is used as a node history of in-memory data. This history allows to keep track of currently used nodes, prevents premature de-allocation of shared nodes, and manages the available memory.

2.2 Point-Based Rendering

Point clouds are discrete samplings of continuous surfaces and usually only provide position and color information. The lack of connectivity information disallows for polygon-based rasterization as used for triangle meshes. Pfister et al. [PZVBG00] propose surface elements (surfels) to render complex geometric objects without connectivity information. A surfel is a point sampled from the surface of a geometric object and is rendered as an oriented disc.

Surface splatting by Zwicker et al. [ZPVBG01] is a technique for high-quality rendering of points with irregular spacing. An elliptical weighted filter is applied on a framebuffer that contains the points projected into screen space. Points closer to the center have a higher contribution to the pixel's color. Points with a depth difference above a particular threshold are excluded from the filtering so that only points that belong to the same surface contribute to the splat. This approach was developed in a software renderer. Bosch et al. [BHZK05] improved upon the original surface splatting algorithm by describing a GPU-driven approach to increase the performance. Their method consists of multiple render passes. First, a visibility pass determines which points are visible, then the attribute pass accumulates normal, and color attributes into a framebuffer, and finally, a normalization and shading pass normalizes the attributes and performs shading.

Most point-cloud datasets lack normal information, which is a requirement for drawing surface-aligned surfels for surface splatting. Instead, Scheiblauer [SP11] propose the usage of screen-aligned circles. To allow for higher-quality surface splatting, Preiner et al. [PJW12] describe a technique to calculate missing normals and radii in screen space during rendering.

More recently, Schütz and Wimmer [SW15b] propose a nearest-neighbor-like interpolation technique to improve the visual quality of point-cloud renderings without the need of multiple render passes. Instead of an aligned disk, a screen-aligned quad is drawn per point, and the depth of each fragment is displaced to form the surface of a sphere, cone or paraboloid. The resulting images show strong similarities to a Voronoi diagram. Due to its simplicity and performance, this technique is used in this thesis as well to render the point cloud.

2.3 Shape Detection and Processing

The objective of detecting structures in point clouds is a wide field of research. The term structure is defined very loosely. Structure can mean geometric structures, such as planes, cylinders, or spheres. Structure can also be interpreted as more complex components that represent distinct man-made or natural formations, such as cars or lanterns. This thesis draws inspiration from different shape-detection approaches and pairs them with a clustering algorithm to detect relationships between shapes.

The 2D Hough transform [VC62] is a technique used usually in the field of image processing. This method can detect straight lines such as building contours as well as

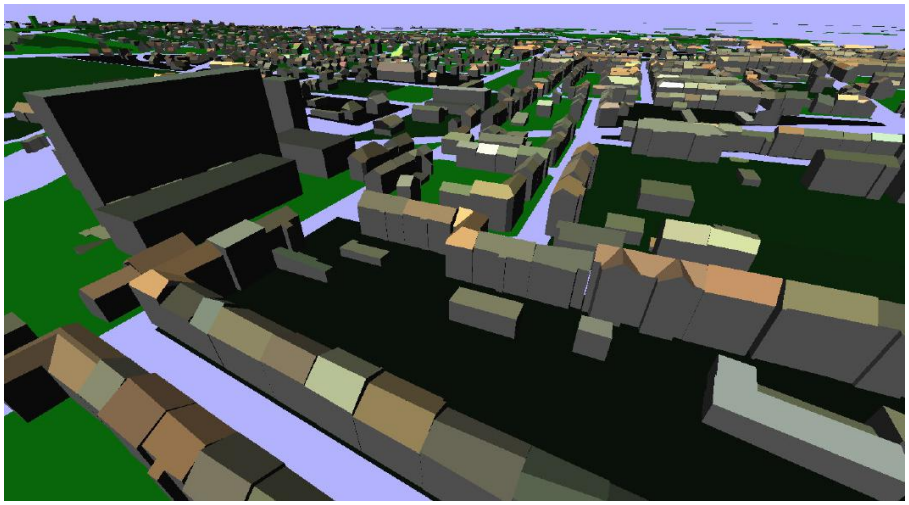
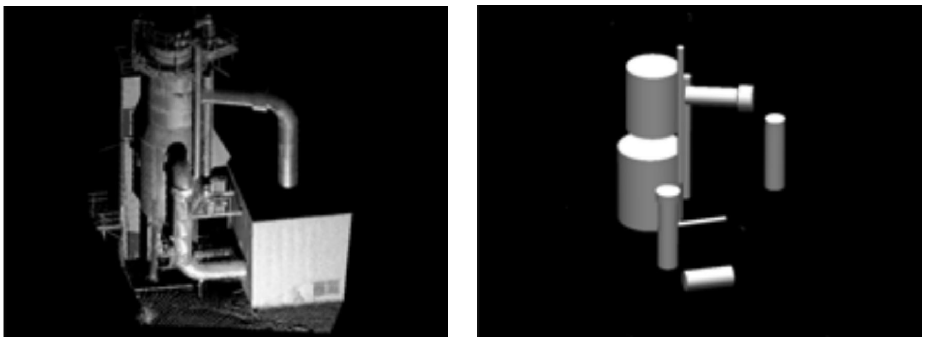


Figure 2.1: Scene created by using 3D Hough transform to detect planes from an airborne laser scanner. Image by Overby et al. [OBKI04].



(a) (b)

Figure 2.2: This figure shows the results of the 3D Hough transform for cylinder detection. (a) shows the input point cloud, (b) shows the detected cylinders. Image by Rabbani et al. [RVDH05].

curves. The Hough transform was extending to 3D by Maas et al. [MV99] and later by Oda et al. [OTDS04] and Overby et al. [OBKI04]. Rabbani et al. [RVDH05] utilize the 3d Hough transform to detect cylinders as well.

Figure 2.1 shows a city model consisting of planes, detected using 3D Hough transform, in a point cloud obtained from airborne laser scanning. Figure 2.2 showcases a point cloud obtained by a 3d scan and the detected cylinders.

Schnabel et al. [SWK07a] propose an alternate technique for shape detection. The authors propose the use of Random Sampling Consensus [FB81] to extract a minimal set of primitive shapes that approximate the global structure of the point cloud. The

Figure 2.3: Left: the original point cloud. Center: The points that belong to a detected shape in random colors. Right: The colors are determined by type (plane = red, cylinder = green, sphere = yellow, cone = purple, torus = grey). Image by Schnabel et al. [SWK07a].

algorithm randomly selects a set of points that roughly follow the curvature of a shape. If a defined number of points are approximated by this shape, the shape is considered to be valid. This approach is capable of detecting planes, cylinders, spheres, cones, and tori and has evolved into one of the most prominent shape detection algorithms and is used in this thesis as well. Later, Schnabel et al. [SWK07b] extended their existing solution, to be capable to process out-of-core datasets. An octree is used that partitions the point cloud into chunks of data that are suitable as input for their RANSAC shape detection.

Figure 2.3 shows the results of the RANSAC approach on a point cloud. Not only does this approach provide the geometry of the detected shapes, it also determines the membership of a point to a shape.

Tarsha-Kurdi et al. [TKLG+07] analyze the performance of the 3D Hough transform and RANSAC for detecting roof planes from airborne laser data. RANSAC proves to be more robust to noise and more efficient.

Besides the RANSAC approach and the Hough transform, graph-based methods are an alternative approach for shape detection and object recognition. Golvinskiy et al. [GKF09] utilize graph-based methods paired with machine learning to recognize shapes in urban environments in 3D point clouds. This method can detect objects, such as cars, newspaper boxes and traffic lights. Potential object locations are identified by clustering nearby points before the point cloud is segmented into foreground and background. For each cluster, a feature vector is built that is used in a trained classifier to obtain a final classification. By using a pre-trained classifier, any type of object can be detected. This method has the benefit of not being limited to the basic shapes types as with the RANSAC approach.

Due to manufacturing reasons, man-made objects are often a composition of primitive shapes that follow constraints, such as parallelism and orthogonality. While RANSAC and the Hough transform produce satisfactory results, these relations between shapes are often

overlooked. Therefore, algorithms that determine relations between shapes and refit the point cloud accordingly are needed to introduce information on a global scale to the point cloud. GlobFit by Li et al. [LWC+11] uses the RANSAC approach to detect primitive shapes along with their global mutual relations. The authors propose an automated approach that iteratively learns from the local relations and adjusts the shape-detection constraints accordingly. Starting from an initial set of RANSAC-detected primitives, the system searches for relations, such as orientation (e.g., parallelism, orthogonality), placement (e.g., coplanarity, coaxis), and equality among shapes and refits the point cloud to match the shapes, before serving as input for the next iteration.

O-Snap by Arikan et al. [ASF+13] utilizes Schnabel's algorithm to extract an initial model from a point cloud used in a reconstruction and modeling pipeline. Their approach introducse an additional polygonization step that creates enclosing polygons from detected shapes and approximated points using the local adjancency relations of the point cloud. They combine these automatic steps in an interactive workflow to snap polygon elements together, while simultaneously fitting the input point cloud to ensure the planarity of the polygons.

Oesau et al. [OLA16] propose an alternative approach to detect planar shapes in point clouds by using region growing and pairs it with a procedure to detect relationsships between shapes. A shape is represented as a set of points and an associated fitting plane. Points or shapes from the neighborhood are added consecutively to the plane, thus growing the region. After shape detection, relationships (regularities) between shapes are determined. First, parallel relationships are detected and the shapes are realigned to a cluster. In a next step, orthogonal relationships between two clusters are determined. Coplanarity relationships are detected by creating a cluster based on the distance between planes.

2.4 Interactions

Interactions are a crucial tool to improve data quality of the point cloud. Tasks like removing unwanted points, selecting regions of interest, or creating new geometry are examples of interactions that can be performed on point clouds to create more distinct visual representations of the objects in the point cloud. Many tasks require a selection of points of interest beforehand.

The lasso interaction is a common tool to select regions in two-dimensional screen space. Lucas et al. [LBCW05] brought the lasso interaction into a tree-dimensional environment by drawing the lasso on a tracked 2D canvas that shows a desired view of the scene. Elmqvist et al. [EDF08] propose to perform a selection in multi-dimensional data by selecting data sequentially in different 2D scatterplots using the lasso selection.

Yu et al. [YEII12] present two new methods of interaction using only two-dimensional input. The results are two techniques that turn a two-dimensional lasso into a three-dimensional volume that is fitted to the spatial structure of the point cloud. Similar

to sketch-based modeling [IMT07], TeddySelection inflates a user-drawn lasso using a heuristic that takes the local point density into account and fits it to the indented region. CloudLasso uses the Marching Cubes algorithm [LC87] to identify and select regions within the lasso where the density is beyond a threshold. Both techniques only use two degrees of freedom, thus can be used in a traditional mouse-based interaction, as well as in direct-touch environments.

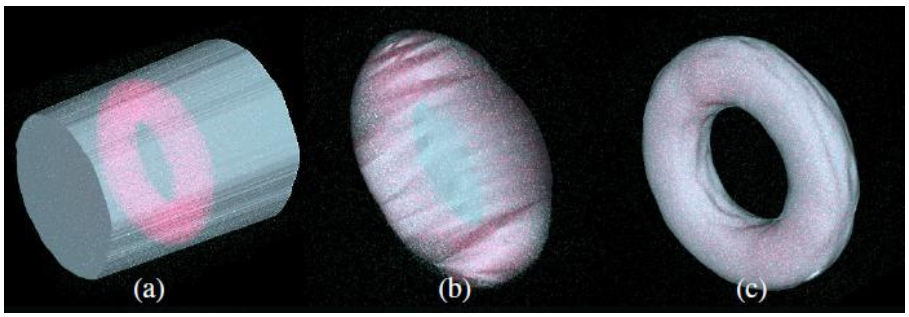


Figure 2.4: The region in grey describes the selection volume. Points that are selected are colored in pink. (a) shows a classic lasso selection, (b) shows a TeddySelection and (c) shows the CloudLasso. Image by Yu et al.[YEII12].

Figure 2.4 compares the results of a simple lasso selection, TeddySelection and the CloudLasso.

An example of a three-dimensional interaction technique is the Volumetric Brush by Weyrich et al. [WPK+04]. The brush follows the local curvature by retrieving the current depth value for the cursor's position from the z-Buffer. The reconstructed world-space position is then used to as the center of a volume, usually a sphere, to select points. Scheiblauer and Wimmer [SW11] utilize the volumetric brush for selections in point clouds.

Picking is a special case of selection where only a single element is selected. Raycasting is used frequently due to its simplicity for the user and performance over distance. However, raycasting on large datasets is limited by the CPU quickly as extensive geometric intersections must be calculated for each object.

Zhu et al. [ZDWX08] present a picking technique for 3-dimensional objects based on view space. The authors propose the use of a bounding-box tree. This hierarchical structure makes it easy to perform view culling and prematurely discard objects that do not intersect the pick ray. The bounding boxes are transformed into view space, where a first intersection is calculated. If a box intersects, intersections with the actual object are calculated. Zhang et al. [ZLL09] propose two GPU-driven primitive-picking algorithms. The first method renders the primitives along with a unique id into a render target texture and retrieves the pick information by downloading the texture. The second approach performs the pick ray intersections in screen space by transforming each

primitive using a geometry shader that emits the intersection information. The CPU retrieves this information using transform feedback.

Huang et al. [HWZF14] propose a GPU-driven implementation for point picking in large-scale point clouds. The idea is to perform picking in screen space and choose the point that is closest to the mouse position. Their approach utilizes the parallel architecture of the graphics card for screen-space transformations and distance measures for all points. Potree [Sch16] applies a similar technique without the use of compute shaders. Instead, a unique per-point id is rendered into a texture from which a small window around the cursor is downloaded to the CPU, and the final picking decision is performed.

CHAPTER 3

Out-of-core Octree

The term *out of core* is used to describe the management of datasets whose size exceeds the available system memory. In this thesis, an out-of-core data structure is needed to handle large-scale point clouds efficiently. Section 3.1 gives an overview in the data structure that is used in this thesis, Section 3.2 discusses the out-of-core organization of the octree. Section 3.3 gives insight on the post-processing of the point cloud once the octree is created. Section 3.4 describes the task of creating a smaller representation of the octree that can be rendered efficiently. Section 3.5 concludes this chapter with a brief discussion about the memory consumption of this solution.

3.1 Overview

An octree is a hierarchical data structure in which each node represents a spatial region defined by a three-dimensional bounding box. If the decision is made to split a node, eight children are created, each representing an octant of the parent's bounding box. Due to the spatial subdivision properties, an octree is a popular data structure for storing point clouds. The spatial subdivision is a characteristic that is shared amongst almost all octree implementations. A characteristic that varies in each application is the composition of the content of octree's nodes. Various approaches exist that store point clouds in different ways.

If an octree node is partitioned, the node's point set is distributed among the child nodes and the original node. Instant Points [WS06] keeps a small subset of points in the original node and distributes the remaining points according to the spatial position. This way, no points are duplicated, making it very memory efficient. Wand et al. [WBB+07] distribute the entire point set among the node's children. The original node keeps an averaged subset of points. Thus, a multi-resolution representation of the point cloud is created at the cost of a larger memory footprint.

This thesis uses an octree with a structure similar to that of Wand et al. If a node is partitioned, the point set is divided amongst its child nodes. A random subset of the original points is kept in the original node. Thus an efficient level-of-detail representation of the point cloud is created. The reason for not using the averaged subset is that the octree is tailored towards shape detection. This processing step is intended to be performed on original data. Moreover, for shape detection, each node is viewed as self-contained, such that no points from predecessor nodes are needed to represent the point cloud for this region and resolution entirely. This self-containment property is the reason why Wand et al.'s approach combined with the proposed subsampling technique is favored above an Instant-Points-like system.

Numerous decision rules exist that determine whether a node is split or not. A node is partitioned if the point count exceeds a threshold n. In this thesis, a decision based on the number of points in a node is favored as it keeps the number of points per node consistent. Having a consistent number of points allows the variation in loading time and data size to be kept low. With nodes of consistent size, the runtime of procedures that are directly affected by the number of points for different nodes is similar as well. Therefore, such methods can be used in a context where immediate feedback is useful, such as user interactions, since the execution time can be estimated. In this thesis, a split threshold of 5000 points in a node is chosen.

3.2 Out-of-core Functionalities

This thesis utilizes an out-of-core octree that stores each node's information and content separately. A *chunk* describes a coherent portion of data that is stored in the cache file. Each octree node is built so that the node's information and node's content is stored in separate chunks. If an octree node is loaded into memory, the content of the node (i.e., the point set) remains on the hard drive. Only on explicit access, the point-set data is loaded into memory. Child nodes are stored as chunks as well. This interleaved structure of chunks allows for the minimal memory consumption for nodes, whose content is not needed directly. For example, bounding-box intersections can be calculated without the need of loading the point set into memory. Figure 3.1 shows the interleaved chunk structure of the out-of-core octree. Loading data into memory can only be done as long as free memory is available. Unused chunks of data are removed from memory after not being accessed for a distinct amount of time.

3.3 Octree Postprocessing

Point clouds often only contain information on position and color and lack distinct geometric features such as normal vectors. Normal vectors are significant for shape detection as they introduce information on the local curvature to the point cloud. After the octree build process is completed, additional properties are computed to enrich the dataset with more specific information.

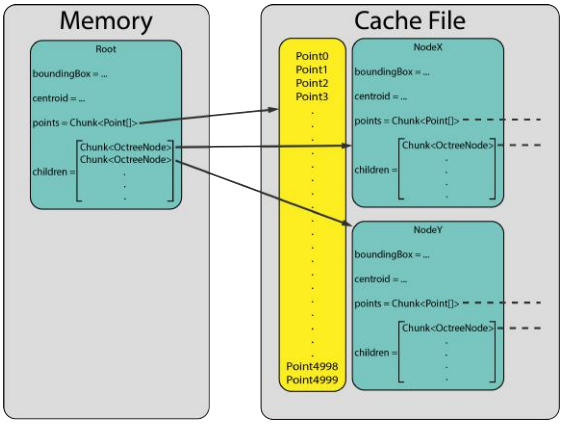


Figure 3.1: This figure showcases the out-of-core structure of an octree with only the root node in the system memory(left). The point set remains in the cache file, as well as the node's children. Only necessary information, such as the node's bounding box and centroid are loaded into memory together with the node itself.

3.3.1 rkd-Tree

An rkd-tree [Tob11] is an efficient data structure to perform fast n-nearest neighbor searches in static data sets. While the octree partitions the space into somewhat small regions, the rkd-tree further bisects the point set along the axes of the point space until only a single point is contained in each section. The rkd-tree improves the kd-tree by Friedman and Bentley [FBS75] by storing the radius of the sphere containing all points from the node's left and right subtrees. Thus, an early exclusion test with the sphere improves the performance of point queries. Instead of using a pointer-based binary search tree, the point array is rearranged so that the split point (i.e., median) for each bisection is located between the values of the one subtree and the values of the other subtree. Therefore, for each point in the array, only the index of the split dimension needs to be stored.

3.3.2 Normals

For lighting and shape detection, each point must possess a normal vector. The local neighborhood determines a point's normal. Using the node's rkd-tree, a k-nearest-neighbor search is performed to retrieve the k closest neighbors. Principal Component Analysis [Jol02] is used to fit a plane into the neighborhood. The plane's normal is defined by the eigenvector with the smallest eigenvalue. The plane's normal vector is used as the point's normal.

3.3.3 Centroid

The centroid of a node provides an indicator of the distribution of points in the octree node. The centroid is used as a target for the camera to focus on the presumably most

dense part of the point cloud.

3.3.4 Density

The density describes the average distance between a point and its nearest neighbor. The density increases with higher level-of-detail since more points are contained in a smaller region. To find the nearest neighbor, the node's rkd-tree is used again.

3.4 Octree Culling

As a point cloud contains more data than the GPU can render in reasonable time, only points from those nodes are drawn that contribute to the currently viewed scene. The result of the culling operation is a new octree that contains only nodes that are currently rendered. The culled octree uses the same cached point information as the original octree. Thus, memory consumption is kept minimal.

A simple yet powerful culling heuristic is view-frustum culling. Nodes that are outside of the view frustum are discarded. By using view-frustum culling, whole branches of the octree are removed. The remaining branches are still too large to be rendered completely. A level-of-detail decision function determines whether or not a node should be rendered. Depending on the node's volume and distance to the near plane, a decision is made if the node should be rendered or not.

The heuristic culls hierarchically, meaning that if a parent is excluded, its children are as well. Thus, entire branches can be removed from the octree efficiently by only checking the parent node. If the octree is culled purely for rendering purposes, it is sufficient to collect the visible nodes in a set and use it for rendering. However, since interaction and processing tasks are performed on the viewed data, the hierarchical structure is kept intact. The intact hierarchy allows these tasks to exploit the octree structure and exclude entire branches of the culled octree with little computational cost.

3.5 Memory Consumption and Performance

The number of nodes from one level of detail to the next increases by the factor of 8 (assuming that the octree is balanced and subdivided evenly). In reverse, when starting from the leaves, the number of nodes decreases by the factor of $\frac{1}{8}$. The upper bound of the relative size of the data stored in the entire octree is calculated from the geometric series:

$$s = 1 + \frac{1}{8} + \frac{1}{64} + \frac{1}{512} + ... = \sum_{i=0}^{\infty} \frac{1}{8^i} = \frac{8}{7}$$

To create the level-of-detail representation of the point cloud the size of the cache file and the number of stored points is increased by the factor of $\frac{8}{7}$.

All nodes from the culled octree are rendered, resulting in the multiple drawing of subsampled points from nodes with a lower level of detail. The previous calculations can be applied to estimate the overdraw when rendering the point cloud. The multiply drawn points account for ∼12% of the entire point budget. The overdraw could be reduced by not drawing nodes whose children are already drawn. However, the current implementation lacks this improvement.

CHAPTER 4

Shape Detection and Clustering

This chapter gives an overview of the shape detection and processing pipeline and presents a clustering algorithm to create a larger-scale representation of a geometric shape from small individual shapes. Shape detection is an automated approach to detect geometric primitives in point clouds. Results are usually computed offline, as the computation takes an extensive amount of time. This thesis utilizes an automated shape-detection algorithm by Schnabel et al. [SWK07a] that is capable of detecting different types of primitive shapes. It is designed to find shapes in point clouds that consist of several million points within minutes. However, when looking at the performance for smaller samples, results can be achieved at interactive time rates.

Shape clustering describes the procedure of creating a larger coherent shape from the detected primitive shapes using a selected shape as an initial start. The focus of clustering is to provide the user with information of larger-scale geometry around the region of a selected base shape.

4.1 Overview

Section 4.2 describes the algorithm for shape detection in point clouds in depth. Since some of the detected shapes are of infinite size, they need to be refitted to encapsulate the corresponding support points and create a minimal boundary. Section 4.3 proposes a postprocessing step to refit a primitive shape onto a point set, such that the size of the shape is kept minimal.

The goal of this thesis is to perform shape detection semi-automatically. Rather than performing shape detection on the complete point cloud, the procedure is executed only on a small subset of points, namely the content of a single octree node, at a time. Performing shape detection on single nodes of the point cloud results in shapes whose

size is limited to the extents of the octree node. However, we assume that shapes are presumably a part of a larger structure.

The starting point for the clustering process is a single user-selected shape, called base shape. From this base shape, the algorithm creates a coherent shape cluster that represents the local geometry on a larger scale by searching for neighboring shapes of similar form from the rendered parts of the octree. Section 4.5 proposes a set of heuristics to determine if two primitive shapes can be combined, thus creating a larger coherent shape. Section 4.6 describes the clustering procedure in detail. The resulting homogeneous shape cluster can later be used for user interactions and rendering.

4.2 Efficient RANSAC for Point-Cloud Shape Detection

The section gives a brief overview over the algorithm used to detect primitive shapes. Schnabel et al. [SWK07a] propose an automated way to detect simple primitive shapes in unstructured point clouds. The point cloud is decomposed into a set of shapes and a set of unused points. The algorithm supports detection of planes, spheres, cylinders, cones, and tori.

RANdom **SA**mpling **C**onsensus (RANSAC) was first discussed by Fischler and Bolles [FB81] as a paradigm for model fitting for image analysis and automated cartography. However, this approach can be generalized for points of an origin other than images. The shape detection works as follows: A minimal set is drawn randomly from the point data (e.g., three points whose normal vectors point in the same direction in case of a minimal set for a plane), and a primitive shape is constructed from it. This shape is called candidate shape. Section 4.2.1 describes the properties minimal sets to build different primitive shapes. The candidate shape is then tested against all remaining points to determine how many of them are well approximated by this shape. This evaluation is described in Section 4.2.2. After a given number of trials, the candidate shape that approximates the most points is chosen, and the next RANSAC iteration is executed with the remaining points. Section 4.2.3 gives an insight of the performance of the shape detection.

4.2.1 Minimal sets

A minimal set is the smallest set of points that describes a particular shape explicitly. The points are treated as possible points on the surface of a primitive shape. The primitive shape is considered a candidate shape if a set of rules applies. The RANSAC iteration randomly selects a set of points and creates a minimal set for a particular shape from it if one of the following rules can be applied:

- **Plane**: The minimal set of a plane consists of three points p_0, p_1, p_2 whose normals do not deviate from the plane's normal more than the angle α.

- **Sphere**: The minimal set to construct a sphere shape consists of two points p_0, p_1 with corresponding normal vectors n_0, n_1. The center c of the sphere is defined by the midpoint shortest line segment between the parametric lines $p_0 + tn_0$ and $p_1 + sn_1$. The radius is constructed by averaging the distance of p_0 and $p1$ to c.
- **Cylinder**: In order to create a cylinder, a minimal set of two points p_0, p_1 with corresponding normal vectors n_0, n_1 is used. The direction d of the axis is established by $d = n_0 \times n_1$. The origin c of the cylinder is created by projecting the parametric lines $p_0 + tn_0$ and $p_1 + sn_1$ onto the plane $d \cdot x = 0$ and taking their intersection as origin c. The radius is the shortest distance between p_0 and the axis $c + ud$
- **Cone**: For simplicity, the minimal set for a cone consists of three points p_0, p_1, p_2, rather than two. For each point-normal pair, a plane is created. The intersection of the three planes defines the apex c. To describe the direction of the axis, a plane is constructed from the points $\{c + \frac{p_0 - c}{||p_0 - c||}, c + \frac{p_1 - c}{||p_1 - c||}, c + \frac{p_2 - c}{||p_2 - c||}\}$. The normal of this plane is the direction d of the cone axis. The opening angle is given as $\omega = \frac{\sum_i^{max} (p_i - c) \cdot d}{3}$
- **Torus**: A minimal set of four points with normals is used, one more than theoretically necessary, However, this eases the computation. Two possible rotational axis are found by intersecting the four point-normal lines $p_i + \lambda n_i$[MLM01]. For each axis, a full torus is estimated, and the torus is chosen that causes the smaller error in respect to the four points. The minor radius is found by projecting the points onto a plane that rotates around the axis. A circle is constructed using three points, whose radius is the minor radius of the torus. The major radius is given as the distance from the circle center to the axis.

  If a minimal set does not qualify for a candidate shape, the set is discarded. If a minimal set fulfills the rules, the corresponding primitive shape is constructed and returned to as a candidate shape. From there on, the candidate shape score is evaluated.

4.2.2 Score function

The score function calculates the support of the candidate shape. The support is determined by the number of points that are approximated by this shape. All remaining points in the point cloud are tested against the candidate shape. Again, for each point to be a support point, the following two rules must apply:

- The distance between the point and the shape must be smaller than ϵ.
- The normal of the point must not deviate from the normal of the shape at the closest position more than a given angle α.

While the first rule ensures that only points are considered that are in close proximity to the shape, it is not sufficient to decide if this point is indeed approximated by this shape.

model	$\|\mathcal{P}\|$	ε	α	τ	$\|\Psi\|$	$\|\mathcal{R}\|$	sec
fandisk	12k	0.01	10	50	24	38	0.57
rocker arm	40k	0.003	20	50	73	1k	6.5
carter	546k	0.001	20	200	138	47k	29.1
rolling stage	606k	0.003	20	300	61	16k	15.1
oil pump	542k	0.0015	30	100	202	15k	30.9
master cyl.	418k	0.003	35	300	37	7k	12.1
house	379k	0.002	20	100	130	19k	10.7
church	1,802k	0.002	20	1000	160	690k	40.7
choir screen	1,922k	0.002	20	4,000	81	543k	20.8
				500	372	236k	61.5

Table 4.1: The original statistics from 2007 by Schnabel et al. [SWK07a] on processed models. ϵ is chosen as a constant fraction of the bounding box width. Results have been averaged over 5 runs and rounded.

The point's normal describes the orientation of the structure determined by the point's local neighborhood. Only if the normal vector fits the normal vector of the shape as well, the point is considered.

All points that fulfill the previous two conditions are projected into a bitmap in parameter domain of the shape. From this bitmap, the largest connected component is used as the final set of support points. The candidate shape is valid if the number of support points exceeds a threshold n.

4.2.3 Performance

Table 4.1 describes the statistical results for different models. $|P|$ is the number of points, ϵ the distance threshold, α the maximum normals deviation, τ is the minimum number of support points, $|\Psi|$ the number of shapes found, $|R|$ the number of RANSAC iterations. It can be seen that for a small number of points and weaker constraints, the algorithm returns plausible results within a fraction of a second, even though the results are ten years old. We utilize this feature the detect shapes in our application for small regions at a time to provide to the user with even faster feedback using modern-day hardware. Section 7.1 presents benchmarks of the shape-detection algorithm and verifies the capability to be executed within the desired time.

4.3 Refitting

The result of the shape-detection algorithm is a set of primitive shapes with their corresponding support points. Planes, cylinders, and cones are returned as infinite shapes in the original algorithm [SWK07a]. For rendering and interaction, it is necessary to create a finite representation for each shape. Calculating the intersections between primitive

shapes to obtain boundaries is not helpful because those intersections alone do not describe the exact boundary of the represented surface. Other reconstruction techniques [Jen08, SDK09] rely on the use of neighboring primitive shapes and thus, only work if all shapes are detected beforehand. Since our approach produces a continuous stream of new primitive shapes and accurate reconstruction is not a goal, the refitting process only uses the set of support points to refit the shape and create a finite representation. Spheres and Tori are finite by definition. Hence, they do not require refitting.

4.3.1 Refitting planes

Planes are represented by a point and a vector. All support points are projected onto the plane, thus reducing the fitting problem to two dimensions. The procedure starts by computing the convex hull of the projected points with the help of Andrew's monotone chain 2D convex hull algorithm [And79]. For this purpose, a bounding quad is used as the representation of the plane. The quad is obtained by using the minimum-bounding-rectangle algorithm by Freeman [FS75]. Arikan et al. [ASF+13] take this plane fitting process a step further. Rather than using a rectangular shape in the reconstruction work, an additional polygonization step is performed that extracts a more complex boundary. However, for this thesis the use of a rectangular quad is sufficient.

4.3.2 Refitting cylinders

A cylinder is defined by a center p, direction vector v and a radius r. The height of the cylinder is chosen as the maximum distance between two support points on the axis of the cylinder. This is achieved by projecting all points onto the axis $a = p + vt$ of the cylinder, and selecting the points p_{min}, where t is minimum and p_{max}, where t is maximum. The distance d between p_{min} and p_{max} is the height of the enclosing cylinder. The cylinder is refitted such that the new center is set to $p' = p_{min}$ and the d is encoded in the length of the new direction vector: $v' = \frac{v}{|v|}d$. The radius stays the same.

4.3.3 Refitting cones

A cone is defined by its apex c, axis direction v, and opening angle θ. Similar to the cylinder, all support points are projected onto the axis and the points p_{min}, p_{max}, with minimum and maximum t, are selected. Since the apex of a cone is fixed, the range cannot be encoded using c and v. The range is stored separately. Range checks are performed when rendering or interacting with cones.

4.4 Shape-Detection Parameter Selection

This section briefly discusses the issue of selecting optimal parameters for the shape detection. The ϵ parameter creates an ϵ-band that follows the curvature of the shape. All points within this ϵ-band are considered to be candidates. The authors propose to use the largest dimension of the point cloud's bounding box times 0.1 as ϵ. However, using such

a static parameter yields problems with extremely sparse regions and regions that are populated very densely. In this thesis, shape detection is performed semi-automatically on local regions of the point cloud at a time. The density of this region, calculated by averaging the distances of each point to its nearest neighbor, is chosen as ϵ. Thus, regions that are populated more densely create finer geometry.

The α parameter is used to determine the deviation between two directions. As the normals are the same at different levels of detail, this parameter is static. We use an α value of 0.95.

The minimum number of support points n per shape is set to 250. While performing tests for this thesis, a higher number increased the number of undetected shapes significantly. Weaker constraints resulted in the detection of falsely detected shapes.

4.5 Shape Matching

This section presents a set of matching functions to determine if two shapes originate from the same geometry within the RANSAC tolerance. Only shapes of the same type can be matching. Hence, it is not necessary to define functions to match for example a plane shape with a cone shape since the result will always be false. Shape matching is performed on two primitive shapes that are detected by Schnabel et al.'s algorithm. It is a key part of the clustering process in Section 4.6.

4.5.1 Elementary Matching Functions

As the primitive shapes are represented by only a handful of parameters, it is sufficient to determine a similarity measure (called *matching*) of these parameters. First, elementary matching functions for numbers, vectors, positions, and axes are defined. Since matching is an approximation of equality, each matching function's result is determined by comparing a computation result to a threshold λ. If a matching function returns `true` (i.e., the values don't deviate more than the given threshold), the input values are considered to be matching.

- **Matching floats** f_1, f_2:
$$\frac{f_1}{f_2} \geq \lambda, \text{ where } f_1 \leq f_2$$
- **Matching vectors** v_1, v_2:
$$\frac{v_1}{|v_1|} \cdot \frac{v_2}{|v_2|} \geq \lambda$$
- **Matching positions** p_1, p_2:
$$\sqrt{(p_1.x - p_2.x)^2 + (p_1.y - p2_y)^2 + (p_1.z - p_2.z)^2} \leq \lambda$$

- **Matching axes** a_1, a_2:
  An axis is defined by a start and end point. Let p_{01}, p_{02} be the start and end point of a_1 and p_{11}, p_{12} the start and end point of a_2. Furthermore, let v_1, v_2 be the direction vectors of a_1 and a_2. The rays of the axes are denoted as $r_1 = p_{00} + sv_1$ and $r_2 = p_{10} + tv_2$. The closest distances for start and end point for each axis to the complementary ray are calculated. From those four values, the largest value d is used for decision making. The matching decision is composed as follows:

$$\frac{v_1}{|v_2|} \cdot \frac{v_2}{|v_2|} \geq \lambda_1 \wedge d \leq \lambda_2$$

4.5.2 Primitive Shape Matching Functions

With the elementary matching functions defined in Section 4.5.1, it is easy to define matching functions for two primitive shapes based on the elementary matching functions:

- **Matching plane shapes**: A plane shape consists of a quad that encloses all support points. For distance computation, the plane is used rather than the quad, as the quad is limited to the shape's points. For each corner of each quad, the distance to the other plane is calculated. From those eight values, the largest distance d is chosen. Two plane shapes are matching if the planes' normal vectors are matching in regards to a threshold value λ_1 and d is smaller than or equal to a threshold value λ_2.

- **Matching cylinder shapes**: A cylinder consists of an axis and a radius. Two cylinder shapes are matching if radii and axes are matching.

- **Matching cone shapes**: Cones consist of an axis, an apex, and an opening angle. Two cone shapes are matching if the axes, apexes and opening angles are matching.

- **Matching sphere shapes**: Two sphere shapes are matching if the center positions and the radii are matching.

- **Matching torus shapes**: A torus consists of a center position, an axis and a major and minor radius. Two torus shapes are matching if the center position, axes, major radii, and minor radii are matching.

The matching result heavily depends on the chosen threshold values. Table 4.2 shows the λ values that are used for this implementation. Plane matching uses a custom heuristic that is not depicted in the table. For this heuristic, $\lambda_2 = 0.05$ is chosen. Note that matching floats is a relative measure, whereas matching positions and axes compares absolute distances. Matching vectors uses the angle between the two vectors to calculate a matching.

Matching	threshold values
Floats	$\lambda = 0.99$
Vectors	$\lambda = 0.95$
Positions	$\lambda = 0.05$
Axis	$\lambda_1 = 0.95$, $\lambda_2 = 0.05$

Table 4.2: The different threshold values for parameter matching.

4.6 Shape Clustering

Shape detection is performed on individual octree nodes from different levels of detail. Thus, the size of the detected shapes is limited to the extent of the octree node. While on a low level of detail, a wall may be contained by a single octree node, on a higher level, a node will only contain parts of the wall, and multiple nodes contain primitive shapes that originate from the same wall.

Given a base shape selected by the user, the clustering algorithm aims to find matching primitive shapes and build a larger coherent cluster of primitive shapes over multiple levels of detail. Oesau et al. [OLA16] propose a graph-based clustering for shapes to detect coplanar shapes. Our shape-clustering algorithm works similarly, but is not limited to planes. The octree is searched for primitive shapes that were detected previously and that match the base shape. From this set of shapes, a complete graph is created, and a region-growing algorithm reduces the number of shapes to those that create a coherent shape cluster.

4.6.1 Building a set of matching primitive shapes

The cluster is constructed from primitive shapes that are currently visible. Therefore, rather than searching in the original octree, the octree culled at the render horizon is queried. Matching shapes are found by traversing the octree and returning those shapes that match the base shape. The base shape, selected by the user, has a particular level of detail. For the shape cluster to mimic a structure of a similar level of detail, only shapes from nodes are used whose level of detail does not deviate more than a user-specific threshold l. Let l_0 be the level of detail of the base shape, only shapes are considered whose level of detail lie in the range of $[l_0 - l; l_0 + l]$. Hence, a threshold $l = 0$ only allows for shapes of the same level, $l = 2$ allows for shape from two levels above and below the base shape's level of detail to be considered. Using a threshold value of 0 creates small clusters, as larger overlapping shapes cannot form bridges between smaller shapes. While testing, the threshold $l = 2$ creates large enough clusters include meaningful parts of a structure, while still having little to no unwanted shapes in the cluster. Shapes with the largest extent from the lowest level of detail hardly get clustered as they usually differ significantly from the smaller shapes.

This set of shapes already creates a cluster where each primitive shape can be seen as a part of a larger shape. However, the distances between the shapes are not yet taken into

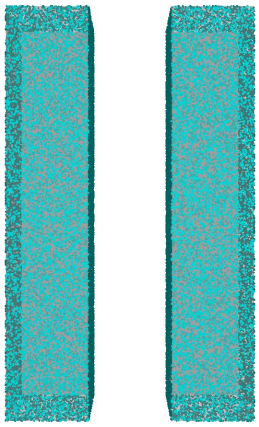


Figure 4.1: A generic point cloud of two cuboids. The detected planes are rendered in grey.

account. Thus, gaps between the shapes may still exist. Figure 4.1 shows the case of two cuboids, whose front face share a plane. By only using the matching functions to create a cluster, both front faces are packed into the cluster, even though there is a visible gap between them. For sphere and torus shapes, this step is sufficient to create a coherent cluster, since both shapes are finite. For infinite shapes, an additional step is performed using a graph-based region growing algorithm.

4.6.2 Graph-based Region Growing

A cluster of shapes can be seen as an ϵ-connected component from a larger graph. A complete graph is created using all matching shapes, as well as the base shape, as vertices. In a complete graph, an edge exists between any pair of vertices. The weight of an edge is determined by a distance function for each primitive shape:

- **Plane Shape**: As a plane shape is bounded by a quad, the distance between two plane shapes is computed as the shortest distance between the two bounding quads.
- **Cylinder Shape**: The shortest distance between two cylinder shapes is determined by the shortest distance between all pairs of start/end points of both cylinders.
- **Cone Shape**: The shortest distance between two cone shapes is determined by the shortest distance between all pairs of start/end points from both cones.

Overlapping shapes can occur when shapes from multiple levels of detail are clustered. However, this causes no problems for the clustering process, as the distance between two

overlapping shapes is set to 0. A cluster is created by growing a region in a graph, adding only vertices that connect to the current region via an edge whose weight is smaller than ϵ. This creates a cluster of shapes, ensuring that the distance to the closest neighboring shape is at maximum ϵ.

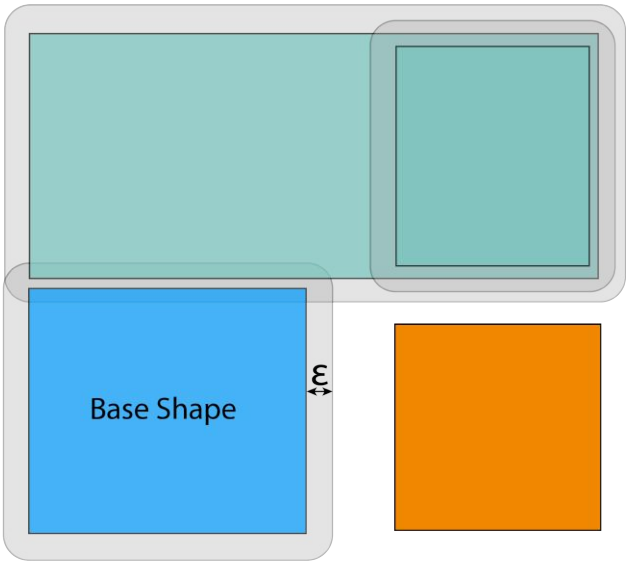
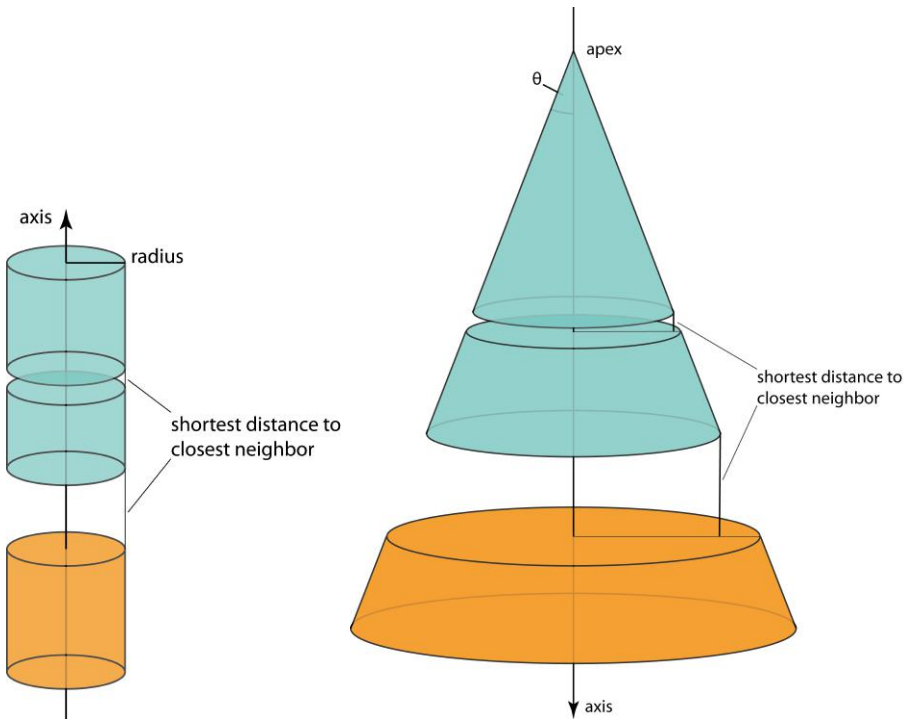


Figure 4.2: A cluster of plane shapes created by computing the ϵ-connected component. The base shape is colored in blue. Planes that belong to the cluster are colored in turquoise. Shapes that do not belong to the cluster are colored in orange. The cluster's ϵ distance is drawn in grey. Each shape that intersects this area belongs to the cluster.

Figure 4.2 shows an exemplary illustration of region growing for plane shapes. The distance between two plane shapes is measured as the closest distance between the two bounding quads.

Figure 4.3 showcases region growing for both cylinder shapes and cone shapes. The matching heuristic already confirms that the shapes lie on the same axis and share a similar radius. Therefore, instead of a three-dimensional world distance, a one-dimensional distance between two points is sufficient to build a shape cluster.

The region-growing component of the clustering heavily depends on the ϵ distance threshold. A proper distance threshold that mirrors the region's topology well is the density of an octree node as described in Section 4.4. For this task, we chose the density of the node of the base shape, more specific: $\epsilon = 2 \cdot density$.



(a) (b)

Figure 4.3: This figure shows a cylinder cluster and a cone cluster build from matching shapes. Shapes that belong to the cluster are colored in turquoise.

CHAPTER 5

System Design

This chapter discusses the design of the application developed for this thesis. The aim is to develop a stand-alone desktop application to view and interact with out-of-core point clouds. The application offers various methods for the user to interact with the point cloud. The application has three main elements that work together, all controlled by the user.

Shape detection, as described in Chapter 4, is performed on a sub-region of the point cloud one at a time. Section 5.2 describes an interactive heuristic that selects an octree node as a region of interest, using the user's cursor position and camera view as input. Using the user's input effectively changes the task of detecting shapes from an automated approach to a user-controlled interaction.

Most interactions presented in this thesis share similar techniques and naming conventions. Therefore, some terms that are used throughout this chapter are defined in Section 5.1.

Section 5.3 describes a picking algorithm to select a primitive shape from the screen that can later be used as support shape for assisted user interactions.

Section 5.4 provides detailed information on the additional user interactions that use a support shape as assistance. *point picking, region selection*, and *local level-of-detail increment* are interactions that benefit from the use of a support shape.

5.1 Term Definitions

The basis of all interactions is the user's cursor on the screen. The *pick ray* is a ray that origins from the cursor's position in world space and goes in the direction of the camera to the cursor position. Each interaction iteratively filters the octree's data such that coarse filtering is carried out before finer adjustments are performed on the dataset before choosing a final candidate. Data that survives the coarse filtering is referred to as *candidate*, e.g., candidate nodes or candidate points.

5.2 User-guided Shape Detection

The shape-detection algorithm is already described in Chapter 4. As the shape-detection computation usually takes several minutes, the approach in this thesis performs this computation on small chunks of roughly the same size of point-cloud data at a time. The octree already provides the point cloud as pieces of spatially neighboring data, such that each node describes an enclosed subset of the point cloud at a specific level of detail. Using chunks of the same size allows the user to expect results in about the same time for each node. This response time should ideally be a fraction of a second, at best less than 250 milliseconds.

The *user-guided shape detection* is updated in the background continuously, relying only on the user's current mouse position and view. Thus, only nodes are considered that intersect the pick ray. To be selected as candidate node from the octree, a node must fulfill the following constraints:

1. The node must currently be rendered and visible to the user.
2. The node must intersect the pick ray.
3. The node must contain at least n points, the same amount used as minimal support point count for shape detection.
4. The node must not have been processed before. Hence, it already contains detected shapes.

These constraints have the following motivation: Since the user controls the shape-detection process, it makes sense that the user only interacts with what is presented on screen. Therefore, the node must currently be rendered and visible. To reduce the amount of redundant computation, only nodes that contain enough points to fit at least one shape are considered to be candidates. Lastly, the shape-detection algorithm works under the probabilistic assumption that, once it terminates, all shapes in this region are detected. Therefore, nodes that already contain detected shapes do not qualify as candidates as well.

The culling operation on the octree, described in Section 3.4, returns an octree that only contains nodes that are rendered. Thus, all nodes in this tree are already visible to the user and fulfill constraint 1. Only a single raycast with the pick ray needs to be performed on the culled octree to obtain the set of candidate nodes that fulfill constraints 1 and 2.

From this set, nodes are eliminated that do not fulfill constraints 3 and 4. The heuristic favors nodes with higher level-of-detail, such that the user receives geometric information for the most detailed parts of the currently explored region first. The projected distance to the nearplane is used to select between nodes with the same level of detail. The node closest to the camera is chosen.

The selection of a suitable candidate node depends heavily on the camera position. When zooming out, the camera moves away from the scene, thus reducing the render horizon and therefore reducing the maximum level-of-detail. If the node with the highest level of detail is processed already the heuristic chooses a node from a lower level of detail. Thus, a multi-scale representation of the local geometry is constructed over time, creating a level of detail for primitive shapes as well.

5.3 Shape Picking

This thesis presents several interactions that are supported by using detected shapes to ease interactions with point clouds. The use of a support shape raises the need for an initial interaction to pick a primitive shape. A raycast is performed on the culled octree using the pick ray to create an initial filtering of octree nodes. From this set of nodes, only those shapes are collected that intersect the pick ray. The picking heuristic prefers primitive shapes of higher level of detail that are closer to the camera. All primitive shapes are collected from the candidate nodes and sorted using a custom key. Primitive shapes that intersect the pick ray come with information on the nearest intersection point and the level of detail. For each intersection point, the *depth* in the range of $[0, 1]$, where 0 is at the near plane, and 1 is at the far plane, is calculated. For shapes that intersect the pick ray more than once, the intersection point closest to the camera is chosen. The sort key is composed as follows:

$$key = level + 1 - depth.$$

The primitive shape with the highest key is chosen as support shape. The composition of the key ensures that a shape with the highest level of detail that is closest to the camera is picked.

As a primitive shape only covers a small region, limited by the extents of the octree node, and user interactions are usually performed on larger areas, a single primitive shape is not sufficient to provide semantic information to the user. Therefore, once a shape is picked, a cluster is created from this shape using primitive shapes that are similar to this shape. This clustering is described in Section 4.6 in depth. All shapes that match the base shape are collected from nodes that are visible and have the proper level of detail. From this set of shape a coherent cluster is created, such that each shape has a neighbor within ϵ distance, as described in Chapter 4. A cluster can be seen as a single, multi-level shape and is referred to as *support shape*.

5.4 Shape-Assisted Interactions

Creating new interactions is a key topic in this thesis. In 3D applications, classic two-dimensional interaction metaphors, such as *lasso selection* and *point picking*, are limited by the lack of information on the desired depth. In a 3D setting, these interactions must

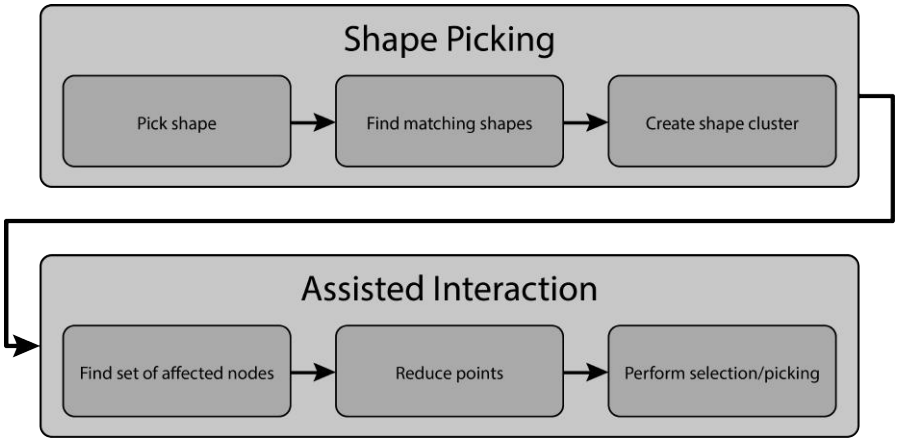


Figure 5.1: The workflow for shape-assisted interactions is divided into two major steps. First, the user picks a primitive shape, from which a larger cluster is created and used as support shape for the following interaction. Upon starting an assisted interaction, the system finds all nodes that are affected by the interaction, reduces the set of points to those that are approximated by this shape and, finally, the interaction (e.g., selection, picking) is performed.

either guess the desired depth boundaries or ignore them. Furthermore, these interactions lack the possibility to mask out unwanted points from a selection. Unwanted points are selected on a regular basis. Hence, many view changes are necessary to select only points that are of interest. By using a primitive shape as support, the user can easily interact with the point cloud in a way that only points are considered that belong the support shape.

Figure 5.1 shows the basic workflow for shape-assisted user interaction. The workflow comprises two major steps. The first interaction is to pick a primitive shape and build a larger cluster from it. This cluster acts as a support shape for the following user interaction. For this interaction, when initialized, all affected nodes are collected, and their point sets are reduced to only those points that are approximated by this shape. A point is filtered if it fulfills the same ϵ-distance and normal-deviation criteria with regard to the support shape as used for the shape detection in Chapter 4. After the filtering step, the actual interaction is performed on the remaining points.

Sections 5.4.1 and 5.4.2 describe the pros and cons of current state-of-the-art two-dimensional interactions and propose improvements using the workflow mentioned earlier. Furthermore, Section 5.4.3 proposes a technique to locally increment the visible level of detail along structures of interest to amplify details.

5.4.1 Point Picking

Point picking describes an interaction where the user is interested in selecting a single point from the scene at a time. The pick radius r denotes the maximum distance of a

point to the pick ray for the point to be considered a candidate point. Multiple picking techniques influence the pick radius r differently. The pick radius r can either be constant in world space, constant in screen space or is depended on the depth value. This section gives two examples on how different pick radii influence the consistency of the picking interaction. After that, shape-assisted point picking is discussed.

Point picking in world space

The first explored technique is to use a fixed pick radius in world space. The picked point is the point closest to the pick ray in world space. Since the user only interacts with points that are projected onto the near plane, the projection of the pick radius is smaller for points that lie in the background. Therefore, the distance in pixels from the mouse position to the picked point in the background is smaller than the distance to a picked point in the foreground. While this encourages the picking of points in the front, the non-uniform pixel distance introduces inconsistencies as the cursor reacts stronger to points in the foreground.

Using a simple raycast is not sufficient to find all octree nodes that are affected by this interaction. If the pick ray does not intersect the node's bounding box, it is still possible that the cylinder created by the pick ray and the radius intersects the octree node. Hence, a cylindrical cast must be performed to collect all candidate nodes.

Point picking in screen space

A more consistent way of picking a point is to only use the screen-space information for each point. The mouse position p in screen space combined with the pick radius r create the pick circle c. This circle corresponds to the projection of a cone in world space with its apex at the camera position, the pick ray as direction and the opening angle defined by the pick radius. All points that intersect this cone are treated as candidate points. To calculate this intersection, all points are projected to the screen space. The cone intersects a point if c contains the projected point. Then the point with the projection closest to the mouse position is picked. This technique works consistently for different depth values. However, since all points are treated equally, the method does not distinguish between foreground and background points, thus introducing possible depth ambiguities.

The projection of points can be executed on the GPU by rendering the projected points, paired with an identifier, to a texture. From this texture, a window around the mouse cursor is downloaded, and the closest point is determined. Reading pixels from a texture forces the CPU and GPU to sync and stalls the graphics pipeline.

Much like picking in world space, the complete set of candidate nodes cannot be retrieved by a simple raycast, as this means that some nodes are node considered that are would intersect the pick cone. Instead, a cone cast is performed to retrieve all affected octree nodes. This conecast is realized by projecting the corners of the node's bounding box onto the near plane and calculating the convex hull polygon. The intersection is determined by the intersection of the polygon with the pick circle in screen space.

Shape-Assisted Point Picking

Classic point picking using one of the two described techniques comes with the disadvantage that some constellations of points can influence the picking interaction negatively. The user is forced to change the view to pick an otherwise occluded point from the structure of interest. In some cases, a point in the background is favored over the desired point on a structure in the foreground. *Shape-assisted point picking* utilizes primitive shapes to perform the picking routine only on points that are part of a structure. The user selects a cluster of shapes, thus reducing the number of possible candidate points to only those that belong to this support shape.

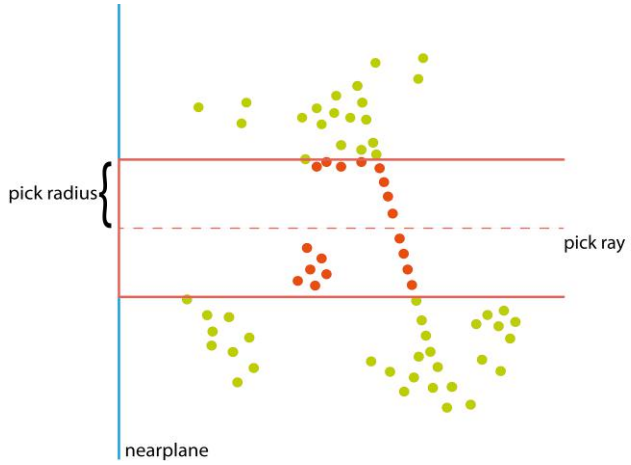
Rather than using a cone or raycast to collect all candidate nodes, only nodes whose bounding boxes intersect the support shape are considered. Furthermore, only points are considered that are approximated by the support shape. This reduction leaves only a handful of nodes and points that follow the curvature of the support shape on which the interaction is performed.

Due to shapes possibly having front and back sides, such as cylinders and spheres, points on the back of a shape are projected near the mouse position as well. By using the projected distances, points that lie on the back side of the shape might get favored over points that are on the front side of the shape (facing the user). Therefore, point picking is performed in world space using a pick sphere. The sphere's position is the intersection point of the pick ray with the support shape. The sphere's radius is calculated by unprojecting the pick radius to the intersection point. Only points are considered that lie in the pick sphere, constructed by the intersection point and the pick radius. The point closest to the intersection point is then selected.

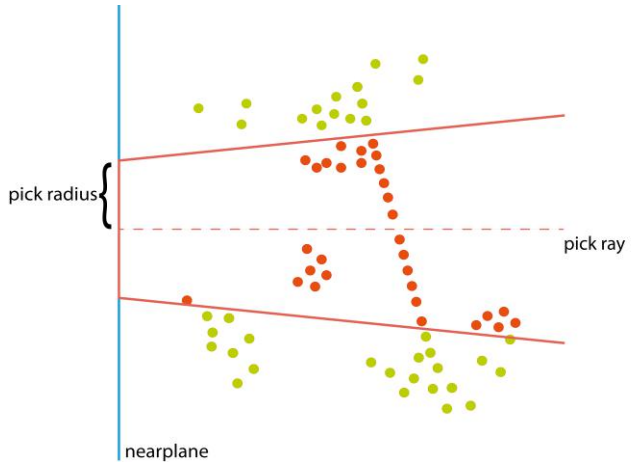
This technique not only improves interaction; computation time is reduced as well. Usually, a shape cluster intersects fewer nodes than a raycast result as the cluster's extension is limited to a moderate region in the point cloud. The number of points per node is reduced as well, and distance measures are computed only for candidate points.

Figure 5.2 shows the different picking methods, described in Section 5.4.1. Figure 5.2a showcases a simple raycast with a radius. The combination of a ray and a radius yields a cylinder, which contains all candidate points on world space. The pick distance is consistent in world space. Figure 5.2b uses a conecast instead. The opening angle is defined by the pick radius in screen space. The pick distance in world space increases the higher the depth value. All points inside the volume are treated equally, introducing consistency in screen space. Figure 5.2c showcases the use of a support shape to filter candidate points. All points that belong to the support shape are filtered before being used as input for a spherecast.

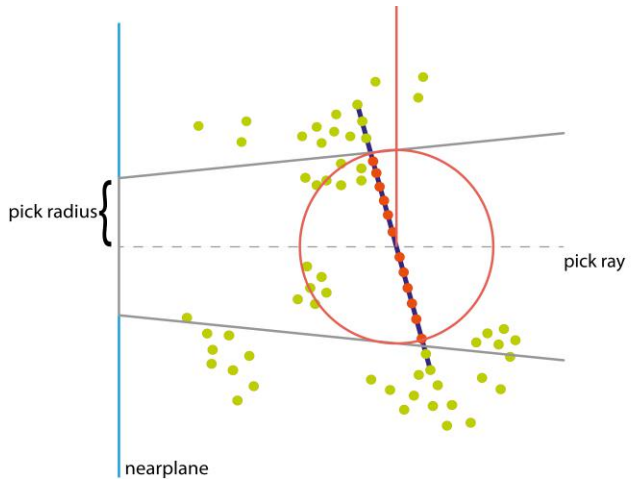
An example of shape-assisted point picking can be seen in Figure 5.3. The cross hair, even though other points' projections are closer to the cursor, sticks to the structure. Picking points on edges is improved in particular since the picking result follows the edge rather than jumping to a point in the background.



(a)



(b)



(c)

Figure 5.2: Two-dimensional illustration of various picking methods. Candidate points are colored in green; other points are colored in red. The areas in red describe the different volumes in which candidate points are located. (a) showcases a picking process using a simple raycast. The ray combined with a radius constructs a cylinder in world space that contains all candidate points, (b) uses a cone instead. (c) utilizes a selected shape (dark blue) to filter candidate points that are approximated by the shape. A sphere cast is then performed on the filtered points using the unprojected pick radius to obtain the final set of candidate points.

Figure 5.3: Shape-assisted point picking is performed on a shape cluster that represents the wall in the foreground. The cross hair indicates the picked point. Note that the cross hair does not jump to a point in the background even though they would be closer to the cursor in screen space.

5.4.2 Region Selection

Region selection aims at selecting a set of points that share certain criteria. The design for the *shape-assisted region selection* is guided by one seemingly simple example task: *Select points that belong to this wall only.* A wall can intersect with other building elements such as roof, balconies or the ground. In regions close to intersections, it is tedious and cumbersome to only select points on the desired structure. Using two-dimensional interaction metaphors, selecting multiple spatially neighboring points along the same curvature is particularly challenging due to the system not knowing the desired depth boundaries for the selection region. In this section, the benefits of using support shapes for two- and three-dimensional interaction metaphors to select regions of points, are discussed.

Lasso Selection

The *lasso selection* is a common two-dimensional interaction metaphor used for multiple geometry-based applications. While it is a useful technique to select regions in 2D, drawbacks appear when porting the interaction to 3D. The user draws a polygon onto the

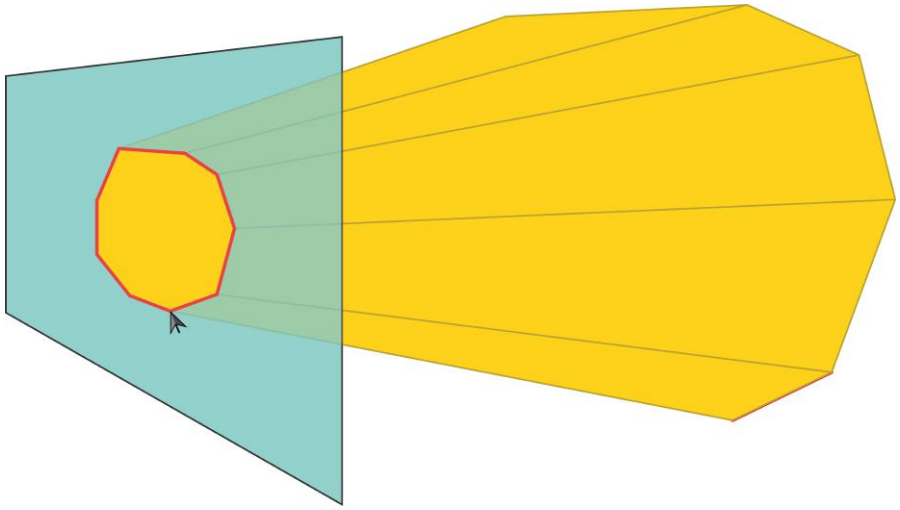


Figure 5.4: The user draws a polygon (red) on the screen (light blue). The constructed three-dimensional area (yellow) contains all points whose projection lie inside the lasso polygon.

screen. All points whose projection lie inside this polygon are selected. Much like point picking, points are projected onto the near plane, the intersection between the point and the lasso polygon in screen space determines whether or not a point is selected. The lasso polygon in combination with the camera view creates a three-dimensional volume, whose area contains all points whose projection lie inside the lasso polygon. Figure 5.4 showcases the volume created by a lasso polygon drawn onto the screen.

Note that this interaction is computed asynchronously and the selection is performed on the entire octree. Therefore, it is essential that octree nodes whose level-of-detail are too high and are therefore not rendered are still included in this interaction as well.

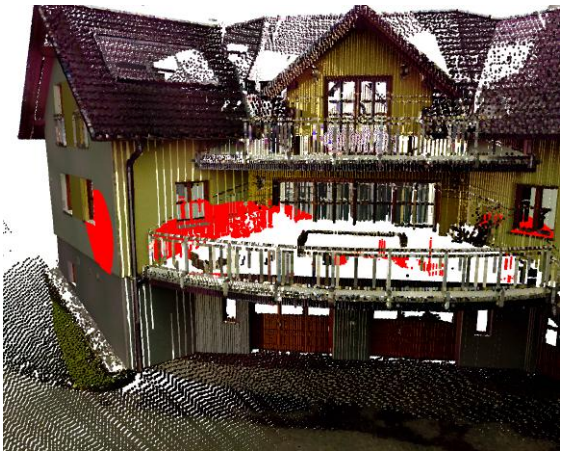
Figure 5.5 shows a classic lasso selection without the use of a support shape performed on a point cloud. The user draws a polygon on the screen. The selected points are highlighted in red. When changing the view, selected points appear that were occluded while drawing the lasso. The user must control the selection distance by hand to minimize this effect. However, to solve the task of only selecting points on the wall, further lasso selections must be applied to remove points from the selection that were selected unintentionally.

Shape-Assisted Lasso Selection

This interaction aims to provide smaller sets of points on which a lasso selection is performed. To construct this smaller sets, the octree is consulted for nodes that intersect the support shape. The number of candidate points per node is reduced by filtering points that are approximated by the support shape. On this reduced set, a normal lasso selection is performed. The result of this interaction is a selection that mimics a lasso selection, with the benefit of not selecting ‘through‘ the point cloud. The depth

(a)

(b)



(c)

Figure 5.5: (a) - (c) show a lasso selection performed on a point cloud. In (a) the user draws a polygon onto the screen. In (b) the selected points are visualized in red. Figure (c) showcases the selection from a different angle. All points that are projected to the area of the polygon are selected. The unintentional selection of points that are obscured by objects in the foreground is a byproduct of the lasso selection.

ambiguities of the lasso selection are circumvented by introducing continuous depth boundaries defined by the local curvature of the shape cluster.

Figure 5.6 shows the workflow for selecting points on a shape. The shape is selected beforehand by the user. A lasso is drawn on the screen that selects all points that lie within the lasso and belong to the selected support shape. In Figure 5.6c, it can be seen that contrary to Figure 5.5c, no points are selected that do not belong to the support shape.

Volumetric Brush

The *volumetric brush* [WPK+04, SW11] is designed in such a way that a volume is projected onto the foremost geometry. Points that intersect this volume are considered to be selected. To retrieve the projected position of the volume (e.g., a sphere) the depth buffer is consulted, and the depth value for the current mouse position is retrieved. The world position is the unprojection of the mouse position's xy-coordinates and the depth value.

Since this technique follows the foremost geometry only, sudden depth changes occur if different geometry occludes the area of interest. Thus, view changes are still required to achieve the example task. In regions close to intersections with other structures, such as below the roof, the user must control the size of the volume to not select points on neighboring structures.

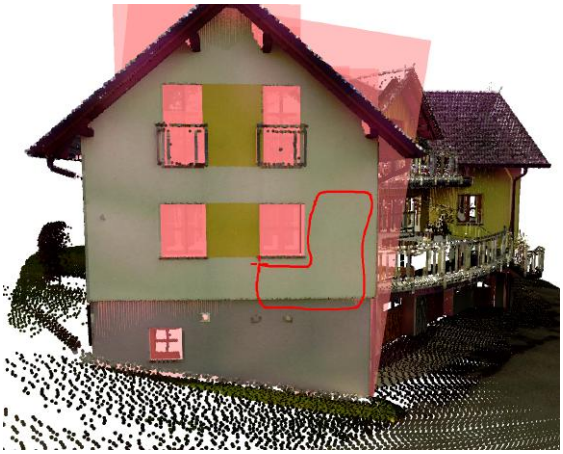
Shape-Assisted Volumetric Brush

The volumetric brush can easily be adapted to be used in combination with support shapes. Instead of consulting the depth buffer to reconstruct the cursor's world position, the pick ray is intersected with the selected support shape, thus resulting in a three-dimensional world position. The octree is consulted for nodes that intersect the support shape, and the sets of points are reduced to only those that are approximated by the support shape. Figure 5.7 shows a *shape-assisted volumetric brush* interaction performed on a wall. The wall creates as a cluster of planes. Only points are selected that belong to this cluster.

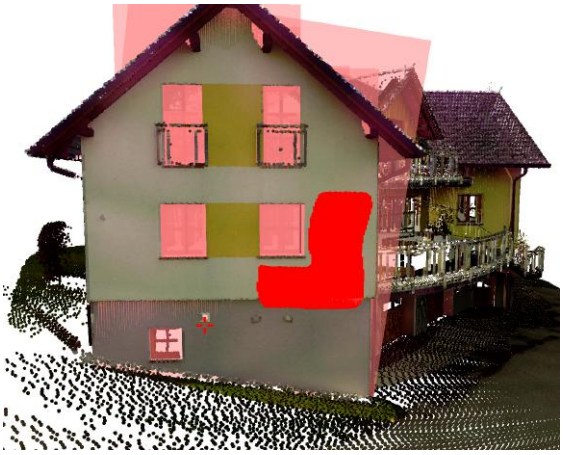
5.4.3 Shape-Assisted Local Level-of-Detail Increment

To further investigate the local structures of a point cloud, the currently rendered maximum level-of-detail might not suffice. The highest level of detail is chosen such that the GPU is not overloaded and the balance between detail and performance is retained. Temporarily adding a handful of additional nodes is sufficient to provide more detailed information, and does not pose an enormous impact on performance.

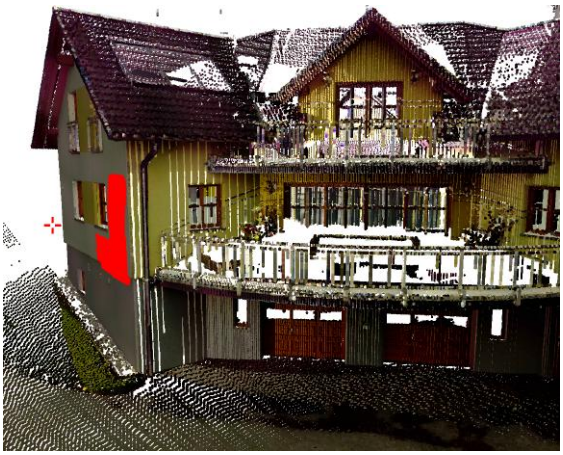
Section 3.4 describes the culling heuristic to create an octree that only contains nodes that are rendered for the current frame. This heuristic uses view-frustum culling paired with a level-of-detail culling decision. However, for the *Shape-assisted local level-of-detail*



(a)



(b)



(c)

Figure 5.6: (a) - (c) show a shape-assisted lasso selection performed on a point cloud. The front-facing wall is selected as support shape by the user. The shape cluster is visualized in light red. In (a) the user draws a polygon on the screen after selecting a shape as support shape. (b) shows the selected points visualized in red from the same point of view. Upon view change, it can be seen that this interaction only selects points that belong the shape cluster.

(a)

(b)

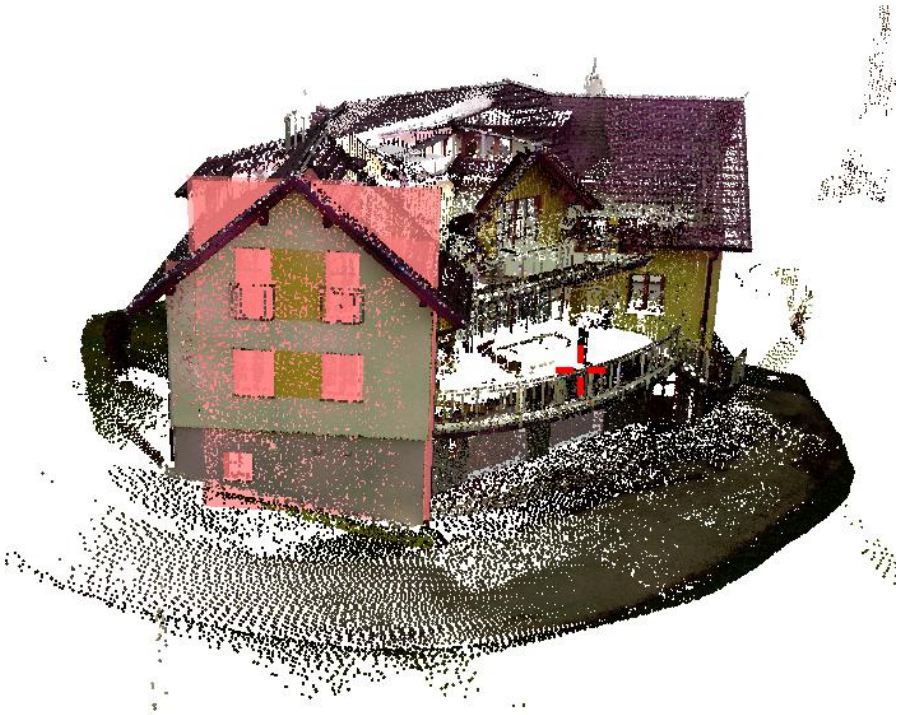
Figure 5.7: This figure shows a shape-assisted volumetric brush selection performed shape cluster (transparent red) detected in a point cloud. In (a) the trajectory of the brush is shown as subsequently rendered spheres(grey). (b) shows the selected points for this brush interaction. Even tough some points of the roof structure are intersecting the brush, they are not selected.

increment the culled octree is not sufficient, and more detailed point information needs to be fetched from the original octree.

Each node of the culled octree is compared with its counterpart in the original octree. All children of the counterpart node that were culled by the level-of-detail decision, but not by view-frustum culling, are collected in an initial set. These nodes share the property that each of them could be rendered on the screen, but its level of detail is too high for the current scene. The same property applies to the successors of each node as well. A *level-of-increment* parameter $i > 0$ controls the number of additional levels of detail that should be displayed for this interaction. For $i = 1$, e.g. the children, the initial set needs no further processing. Thus, it can be used as candidate set for the next step directly. For a level-of-increment parameter $i > 1$, all $(i-1)^{\text{th}}$ successors of the nodes from the initial set are taken as candidate set instead.

By adding smaller nodes with a higher level of detail, the overall detail in the scene is increased. However, by adding nodes without additional point filtering, noise and unwanted structures are amplified as well. Therefore, this interaction utilizes a shape cluster, selected by the user, to amplify detail only on structures of interest. The candidate set is filtered further, such that only those nodes remain that intersect the support shape. For each node from this final set, only those points that are approximated by the support shape are added to the scene.

Figure 5.8 showcases the difference in a scene with and without amplified details along a wall. The selected support shape is rendered in transparent red, while the additional points are rendered without lighting to appear brighter. In the intersecting octree nodes, only those points are rendered that fulfill the shape's score function.



(a)

(b)

Figure 5.8: This figure shows the benefits of shape-assisted level-of-detail increment for a point cloud. (a) shows the currently rendered structure, as well as as shape cluster that is currently selected (red). (b) shows the scene with amplified details along the structure. The additional points are rendered without lighting to appear brighter.

CHAPTER 6

Implementation

This chapter gives an overview of implementation details for this thesis. The implementation of the application is guided by the *functional-first* programming paradigm. *Functional-first* takes inspiration from functional programming, as well as other paradigms, such as object-oriented programming. However, the majority of the application is written in a purely functional way, with connections to non-functional components, such as .NET libraries for multi threading or Schnabel et al.'s shape detection implementation [SW]. The backbone of the application is the Aardvark platform [fVRuVFGa]. It is a functional-first incremental rendering engine in active development at the VRVis Zentrum für Virtual Reality und Visualisierung [fVRuVFGb].

This chapter starts with an introduction to functional programming and the programming language `F#` [F#05] before introducing the key features of the Aardvark platform in Section 6.2. The primary data structure used for organizing the point cloud data is an octree with out-of-core capability. A set of functions that simplify querying with the octree is proposed in Section 6.3.

Multiple coroutines are performed in the background and produce a stream of updates on the point-cloud data. The multi-threaded architecture of the application is discussed in Section 6.5. Section 6.6 concludes this chapter by briefly discussing implementation details of the point-cloud rendering system.

6.1 Functional Programming

Functional programming is a paradigm that views every expression as a mathematical function. The result of each expression is either an elemental data type or a functional type. A core difference to other programming paradigms is the type-level representation of functions (same as data). Thus, functions can be used as input for other functions

and have a distinct type defined by its parameters and result. A simple example of a function in F#:

```
let square (i : float) : float = i * i
```

This function is of type `float -> float`. It takes a `float` as input and returns a `float`. This function type can be used as input parameter as well. A definition for a function that takes a function as parameter looks as follows:

```
let compute (i: float)(f : float -> float) : float = f i
```

Its type is `float -> (float -> float) -> float` and is used as such:

```
let result = compute 10.0 square
```

`result` is an expression that executes the `compute` function with arguments 10.0 of type `float` and `square` of type `float -> float`. Even though this is just a simple example, it showcases the strength of functional programming. Using functions as input allows the user to create complex and dynamic computations with ease.

Each expression in a functional context is mathematically defined, parameter space and result space are fully known and must be fully defined on type level. No values that the program does not expect ever occur, thus eliminating undefined behavior. Functional programming tries to limit mutation, e.g., change of an external variable's value (program state), as much as possible. A program without mutation is called ṕure. Each call to a pure expression with the same parameters must produce the same results. External mutation can change the behavior of expressions that depend on the mutated field, thus changing the result without changing the input. Therefore, avoiding mutation means avoiding undefined behavior. Finally, strict purity and full type-level specification make it possible to reason and construct proofs about programs. This gives the programmer tools to manage very high complexity and reduces the need for debugging significantly.

F# is a functional programming language developed and maintained by Microsoft [Micb]. The language is fully integrated into the .NET framework [Micc] and is built upon the Common Language Infrastructure [ISO12], thus allowing it to use resources written in other languages, such as C# [Mica]. Even though F# is a functional language, it also supports object-oriented programming, such that classes with member functions can be used as well.

6.2 Aardvark

The Aardvark platform [fVRuVFGa] is an implementation of research results on the field of incremental rendering [WSMT13, HSMT15] and semantic shader composition [HSM+14a, HSM+14b]. The key feature of the Aardvark platform is incremental rendering. In a conventional engine, updates are performed periodically. Each scene object is re-evaluated,

even though the simulation or user input may not yield any changes. Incremental rendering counters this overhead by reacting to changes in a way that only components that depend on changed values are re-evaluated. This section describes some of Aardvark's key features that are used throughout this thesis. Section 6.2.1 shows the language-specific constructs on how to build adaptive blocks that react to changes. A *scene graph* represents an object hierarchy in the scene. Section 6.2.2 describes the composition of a scene graph using only a handful of lines of code.

6.2.1 Adaptive - IMod - transact

As mentioned in Section 6.1, mutations are often the cause of undefined behavior and bugs. To introduce state-changeable variables into a functional environment, Aardvark provides the `IMod<'a>` type. This generic type is a wrapper around a particular value, whose value might change over time. The programmer does not care about the concrete value, and only specifies the computation applied to the value instead. Aardvark contains an extensive implementation for a three-dimensional transformation, called `Trafo3d`. The type `IMod<Trafo3d>` listens to changes of the transformation. The following example shows the composition of a model transformation from a position, scale, and rotation, all of which can change over time.

```
let position    : IMod<V3d> = ...
let scale       : IMod<V3d> = ...
let rotation    : IMod<V3d> = ...

let trafo : IMod<Trafo3d> =
    adaptive {

        let! sc  = scale
        let! rot = rotation
        let! pos = position


        let S = Trafo3d.scale sc
        let R = Trafo3d.Rotation rot
        let T = Trafo3d.Translation pos

        return S * R * T
    }
```

The `adaptive` computation expression builder allows the system to keep track of the state of all IMods that are accessed using the `let!` operator. This *binding* operator makes a computation react to their changes. The result of the adaptive block, again, is an IMod<Trafo3d>. If one of the accessed IMods changes its value, the code below,

including the `let!` statement, is reevaluated. The computation is kept minimal using caching.

To actively change a value, a subtype of `IMod<'a>`, `ModRef<'a>` is needed. The type `ModRef<'a>` contains the functionality to allow changes to the value, thus causing re-evaluation. All state transactions are collected and executed sequentially, thus reducing the number of re-evaluations and circumventing race conditions. Each change must be wrapped in a `transact` function, representing transactional logic. The following example uses the Aardvark-specific mouse callback function to trigger a reevaluation based on mouse movement. The Move-callback is called each time the mouse moves and provides the user with the old position and the new position. The difference on the x-axis controls the y-value of the rotation.

```
let mouse : IMouse = ...

let rotation : ModRef<V3d> = Mod.init V3d.OOO

mouse.Move.Values.Add(fun (oldPos : PixelPosition, newPos :
   PixelPosition) ->
    let delta =
            newPos.NormalizedPosition.X - oldPos.
               NormalizedPosition.X
    let angle = delta * 2.0 * Math.Pi

    let newRotation =
            rotation.GetValue() + V3d(0.0, angle, 0.0)

    transact(fun () -> Mod.change rotation newRotation)
    )
```

The `ModRef rotation` can then be used bound within an `adaptive` block.
Transactions are usually used to handle user input. However, asynchronous computations that produce results in parallel use transactions as well to mitigate race conditions on shared data and notify the rest of the program on completion.

6.2.2 Scene-graph composition

A scene graph (`ISg`) contains all information that is needed to render an object. A common paradigm in functional programming is to separate functionality from data, such that the functions that utilize this data are stored in a different namespace than the data. In combination with the pipe operator (|>) clean and easy-to-read code can be produced. The following example showcases the composition of an `ISg`.

```
// Transformations
let trafo :  IMod<Trafo3d> = ...
let view  :  IMod<Trafo3d> = ...
let proj  :  IMod<Trafo3d> = ...

// IndexedGeometry contains triangles to render
let geometry : IndexedGeometry = ...

// The renderpass for this scenegraph
let renderPass : RenderPass = ...
// The shader (surface) to render the geometry with
let shader : IMod<ISurface> = ...
// Create scenegraph
let sg = geometry   |> Sg.ofIndexedGeometry
                    |> Sg.trafo trafo
                    |> Sg.viewTrafo view
                    |> Sg.projTrafo proj
                    |> Sg.pass renderPass
                    |> Sg.surface shader
```

The field `sg` represents a scene graph with transformations, geometry, shader and render pass without the need of complicated constructors or setter functions. Furthermore, an own implementation of `ISg` can easily extend the functionality without reimplemented procedures for this particular type. This extensibility is implemented using a dynamic object attribute mechanism.

6.2.3 Out-of-core capabilites

The Aardvark platform contains functionality to handle out-of-core data. The type `Database` provides means to store data. A chunk of data stored in this database is called `thunk<'a>` where `'a` is the type of the data to be stored. The following code shows how to create a `thunk<'a>`.

```
let db : Database = ...
let data : 'a     = ...

let stored : thunk<'a> = db.ref data
```

When a `thunk` is loaded from the database, its content is not loaded into memory yet. Only when accessing the data directly, it is loaded into memory. This technique is referred to as *lazy evaluation*. The data is held in memory as long as the thunk lives. Therefore it is crucial to keep track of unused thunks and dispose of them regularly, thus freeing memory.

6.3 Functional Out-of-core Octree

Chapter 3 already provides information on the capabilities of the octree. This section describes implementation details of the octree's structure using out-of-core mechanics and custom parametric functions for convenient octree queries.

6.3.1 Structure

The octree's out-of-core capability is managed by an interleaved structure of `thunks`. Each `OctreeNode` contains an array of points. However, the array is not stored directly; it is stored as a `thunk<Point[]>`. Its children are stored in an array of thunks (`thunk<OctreeNode>[]`). The points are stored as a single `thunk`, the reason for storing the node's children in separate thunks is not to load all children into memory when only a subset of children is accessed. Other information such as the centroid, density, rkd tree, and the detected primitive shapes are stored as `thunks` as well.

6.3.2 Elementary functions

The interaction methods described in Section 5.4 rely heavily on the efficient collection of octree nodes that fulfill certain criteria. Thus, a set of routines is implemented that can easily be parameterized and reused. `F#` collections contain several functions that use a lambda function as input and perform this function on elements in the collection. Two examples are `map` and `filter`. Both take two arguments, an array and a function that takes an element as input. All operations on collections can be composed from these basic functions. This is called the *mapReduce pattern*. One example of such a composition is the `choose` function. Even though the octree is not generic, these functions are implemented similarly, to provide a uniform interface to query the octree efficiently.

Filter

The method of filtering returns a subset of elements of the original collection for all of which a decision function returns `true`. Filtering an octree works similarly. The decision must be made whether or not to use the current node and whether to traverse the children as well, as children may be desired, where the parent is not. The function's definition looks as follows:

```
let filter (decisionFun : OctreeNode -> GridCell -> int[] ->
   bool*bool) (tree: Octree) : (Octree)= ...
```

The decision function takes as input the octree's node, grid cell, and unique path. A node's path in the octree is constructed from an array of indices that point to the next predecessor in the tree. It returns a tuple of `bool`, deciding whether to use this node and whether to traverse the children. The result of this operation is a new tree that only contains the nodes that are filtered. For the case that a parent node is not filtered,

but some of its children are, an empty placeholder node is used instead to preserve the octree's structure.

Map

Usually, the `map` function is used for generic data structures to create a projection for each element and returns a data structure of the same type with the projected elements. Since the octree is not generic, the `map` function returns the projection for each node of the octree as an array of `'T` where `'T` is the type of the projection. The functions's definition looks as follows:

```
let map (projection : OctreeNode -> GridCell -> int[] -> 'T) (
   tree: Octree) : ('T[])= ...
```

Contrary to the filter function, the `map` function does not return a new octree since the type of the projection must not necessarily be hierarchical. Therefore an array of `'T` is returned instead.

Choose

A similar task to filtering is the choose function. However, instead of returning the filtered values directly, the decision function also maps the values to the desired type `'T`. The decision function uses the same input as the decision function for filtering. Its return type is a tuple that consists of an `Option<'T>` and a `bool`. The option type is used to introduce null-mechanics into the functional context. An option can either be `Some value` or `None`. If the `choose` function's returned type is `Some value`, its value is filtered, the `bool` value decides whether or not to traverse the children. The function's definition looks as follows:

```
let choose (decisionFun : OctreeNode -> GridCell -> int[] ->
   Option<'T>*bool) (tree: Octree) :('T[]) = ...
```

Figure 6.1 shows the described functions and their effects on an exemplary octree. The `filter` function returns a new octree, the `map` function returns an array of projected types, the `choose` function is a combination of `filter` and `map` such that only those projections are returned that are of interest.

Replacing nodes

As mutations introduce side effects that can lead to bugs, instead of changing information within an octree node, a new node is created containing the new data. The old node might still be in use in a different thread. Thus, race conditions are possible when mutating values. When information changes in an octree node, a new `thunk` is created that stores the new content onto the disc; the old `thunk` is discarded when it is not used anymore. The newly created node has a distinct position that is determined by the

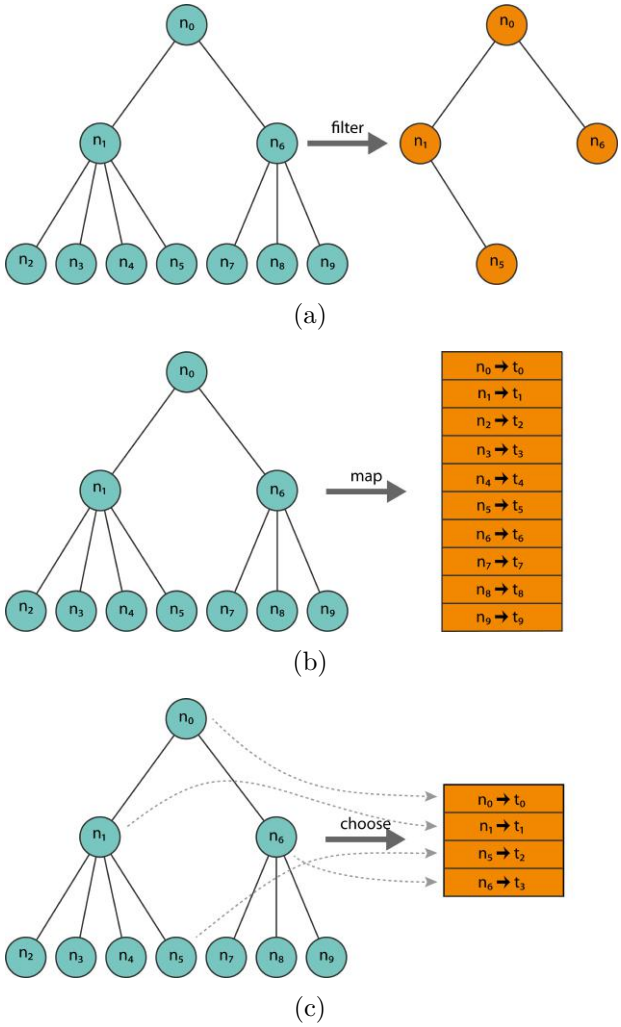


Figure 6.1: This figure shows an exemplary octree, on which the filter, map and choose function is applied. In (a) the filter function creates a new octree that only contains the nodes that are filtered. (b) shows the result of the map function. Each node in the octree is projected to a new type t_i and the result is returned as an array. (c) shows the choose function. Only nodes are returned that fulfill the decision function and are projected to a new type t_i.

path in the octree. The octree is traversed recursively to find the location of the node. When resolving the recursion, the content of all ancestor nodes changes as well, since one of the children is a new node. Therefore, for each ancestor, a new node is created too, containing the new content. Since the root changes as well, as the replaced node is a successor of it, a new octree is constructed each time a node is replaced. This octree, however, contains mostly the data of the old octree, except the replaced node.

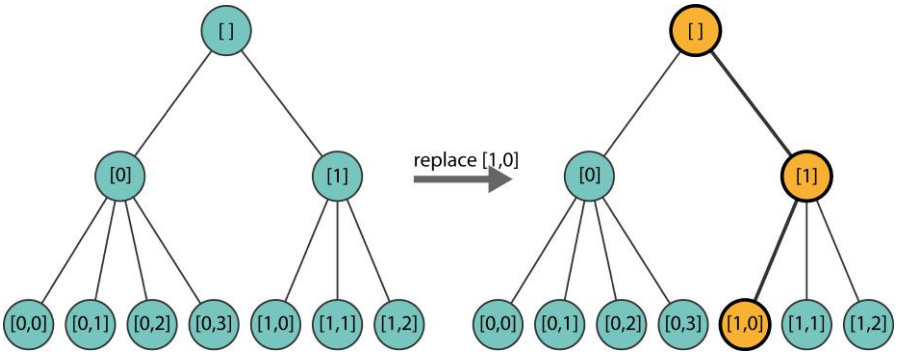


Figure 6.2: Replacing an octree node subsequently changes the node's ancestors as well. The left tree shows the original octree, the right tree is the new octree after the node [1,0] was replaced. The replaced node is highlighted with a red border, all nodes that changed due to the replacement are colored in orange.

Figure 6.2 shows an example on the replacement of a node. The nodes are labeled with their paths in the octree to identify them uniquely. The node with label `[1,2]` is replaced. Thus, all ancestors in the octree are changed as well since the node got a new child. Nodes that changed are colored in orange.

6.3.3 Raycast

Spatial subdivision accelerates ray casting. If the ray does not intersect the bounds of a larger parition of the space, it will not interesect the objects that are contained in the region as well. Thus, large parts of data can be excluded from the ray cast prematurely. The hierarchical structure of the octree ensures that only a minimum set of nodes is tested for intersection. If the ray does not intersect the parent's bounding box, it does not intersect the children's bounding boxes as well. The raycast on the octree can be performed with logarithmic cost. The raycast is implemented using the octree's `choose` function with a decision function that performs the intersection with the bounding box and returns a `RaycastHit` structure containing all necessary information on the raycast hit.

6.3.4 Culling

Culling utilizes the octree's `filter` function to create a new octree that only contains nodes that are currently rendered. The culling function performs view-frustum culling as well as a level-of-detail culling heuristic based on the node's size and distance to the near plane. Figure 6.3 shows an exemplary culling performed on a tree. The level-of-detail decreases for nodes that are further away from the camera and only those nodes are used that intersect the view frustum.

Let V be the volume of the node's bounding box, win_x, win_y the dimensions of the window and d_{min} be the smallest distance from the bounding box to the near plane in

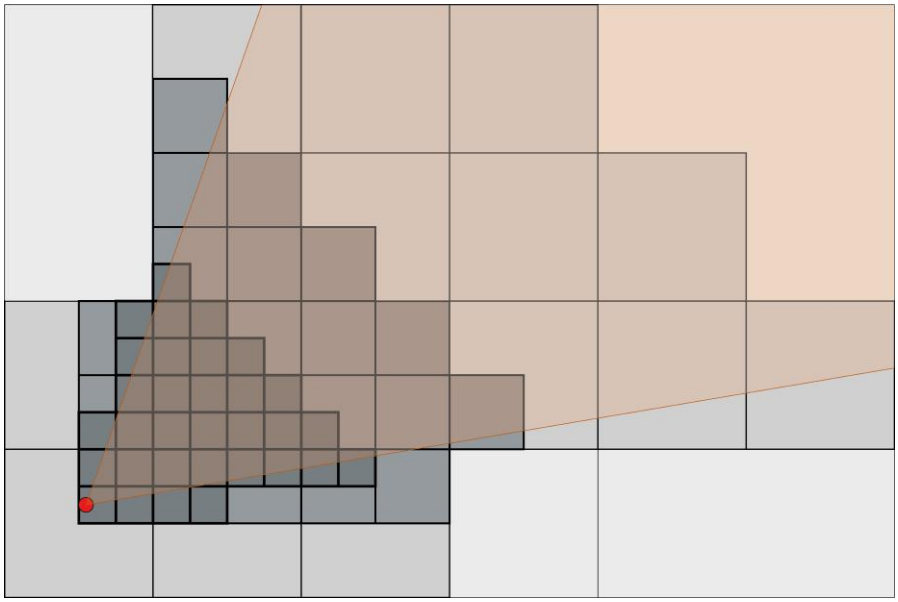


Figure 6.3: An exemplary culling is performed on a tree. Octree nodes that are darker with a thicker edge have a higher level-of-detail. Only close to the camera (red), the nodes with the highest level-of-detail are rendered. Nodes that are outside the view frustum (orange) are discarded.

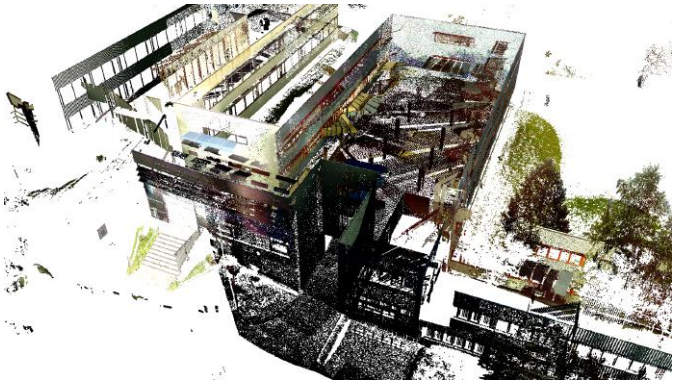
world space, clamped by the near and far plane. Furthermore, let t be a user-controlled distance threshold. For this application, $t = 2$ is used.

$$t_{win} = \frac{t}{max(win_x, win_y)}$$
$$g = \frac{\sqrt[3]{V}}{d_{min}}$$

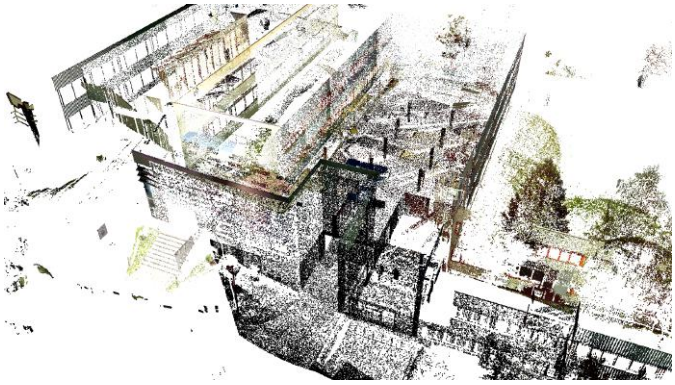
The decision whether or not to render the node is determined by:

$$g > t_{win}$$

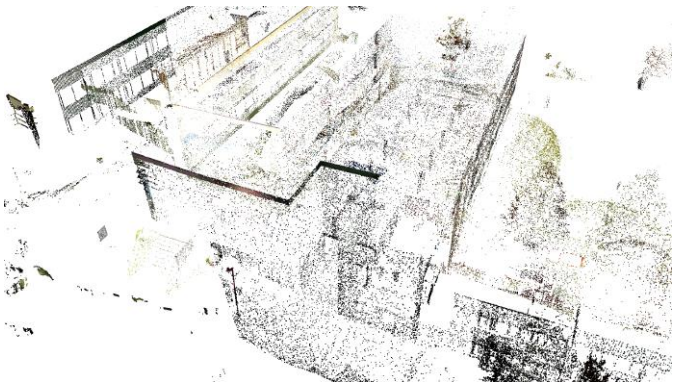
The user-controlled threshold t is divided by the largest window dimension that allows the level-of-detail to adapt to the window size. d_{min} is calculated by projecting all corners of the node's bounding box to view space and negating the z value. The smallest value is then clamped to the view frustum, so the computation stays in visible view space. A smaller t_{win} allows for higher level-of-detail to be displayed. The granularity g of an octree node is controlled by the minimal distance and the size of the node. Nodes close to the camera have higher granularity. Smaller nodes have smaller granularity. Figure 6.4 shows the usage of different thresholds t on a point cloud.



(a)



(b)



(c)

Figure 6.4: This figure shows the Technologiezentrum rendered with a t value of 2, 5, and 10 in (a) - (c). A smaller t results in a more dense visualization of the point cloud for the current view.

6.3.5 Diff

Let $t1, t2$ be two octrees where $t2$ is a sub tree of $t1$ and v a non-negative integer. The function `diff` $t1t2$ v returns all nodes from $t1$ whose parents exist in $t1$ and $t2$, but the node itself only exists in $t1$. v limits the maximum level-of-detail difference. Hence $v = 1$ only allows for the direct children only, $v = 2$ allows for the children and their children as well.

The `diff` function takes nodes that are in the first octree and not in the second octree. It comes in two varieties. The first implementation returns only those nodes with the maximum level-of-detail difference. The second implementation returns all intermediate nodes as well. The construction of the candidates set for the shape-assisted local level-of-detail increment in Section 5.4.3 is implemented using the `diff` function, where $t1$ is the entire octree, $t2$ is the culled octree that only contains nodes that are rendered and v is the level-of-detail increment parameter.

6.4 Sequential Computation Applicator

The octree receives changes on a regular basis from multiple sources in this application. The shape detection coroutine continuously inserts detected primitive shapes into the octree, resulting in a new octree every time. User interactions change the point set by selecting regions of interest. To synchronize the octree access, locking mechanics can be used. However, such mechanics can get confusing easily, especially when the application grows.

All changes to the octree depend on the state of the octree that, in the meantime, may be altered by an operation from a different thread. Furthermore, since a new octree is created every time, the result of an operation may be overwritten by a second operation that starts while the first operation has not yet finished. All subsequent operations depend on the result of the previous operation. Thus, a structure is needed that handles such dependent operations.

The *sequential computation applicator* is a structure that provides a synchronized way of processing sequential operations. The applicator's functionality is synchronized such that multiple threads can dispatch operations on the octree. The *sequential computation applicator* is a wrapper around a `ModRef<'T>` that invokes changes on this value regularly. In this case, the type of the applicator wraps around a `ModRef<Octree>`.

All dispatched operations are processed sequentially, and the `ModRef`'s value is changed after an operation is completed, thus notifying the Aardvark engine. Since Aardvark evaluates adaptives lazily, operations simply get aggregated as long as no results are requested (i.e., rendering is never stalled). The applicator provides the interfaces to dispatch an operation, as well as to dispatch an operation with high priority to enqueue an operation on the front.

```
member public this.DispatchPrioritized(computation : 'T ->
    'T) (timeout : int) (timeoutCallback  : unit -> unit) :
    unit = ...

member public this.Dispatch(computation : 'T -> 'T) (
    timeout : int) (timeoutCallback  : unit -> unit) : unit
    = ...
```

The type `'T` is the generic type of the *sequential computation applicator*, in this case, it is `Octree`. A computation is dispatched that takes a `'T`, and projects it to a new `'T` (i.e. all of the previously mentioned computations). The input parameter on execution is the current value of the octree. Additionally, an operation can be shut down after a defined timeout in milliseconds. If this is the case the `timeoutCallback` is invoked to allow clean up and error handling.

6.5 Multi-Threaded Environment

Multithreading can be achieved on multiple granularities, depending on the task's needs. The application uses three primary threads that run in parallel. The Aardvark rendering engine, combined with the `IMod` evaluation system and user interactions build the main thread of the application. One pitfall of the `IMod` system is that it only reacts to changes in the system, however, to invoke procedures after a particular time without direct changes is not possible since that would involve mutation.

The *user-guided shape detection* in Section 5.2 is triggered when the camera has not changed for a particular amount of time. As the Aardvark `IMod` system only reacts to changes, such no-changes must be invoked by a separate thread that continuously checks for changes. If a shape detection should be performed, this thread dispatches the computation to the applicator thread.

The *sequential computation applicator* performs changes on the octree. All tasks that are dispatched from multiple threads are executed in a sequential order in the background. The thread transacts the changes to the main thread once a computation has finished.

As Figure 6.5 shows, multiple threads are dispatching computations to the applicator thread. However, only this thread transacts changes to the main thread. The shape detection invoker thread works independently of the main thread.

The second parallelization technique used in this thesis is task-based multithreading. The goal is to identify tasks that are executable in parallel for multiple instances of the same type, such as per-point or per-node computations. As long as shared resources are accessed as read-only, tasks can be executed in parallel without race conditions and the need for locking. The input is an array of elements and a mapping function. The mapping function is executed for each element of the input array using the .Nets `System.Threading.Tasks.Parallel.For` statement. The return type is a new

array containing the results of the computation for each element at the original position. The function's signature looks as follows:

```
module Parallel =
  let map (computation : 'T1->'T2)(array : 'T1[]) : 'T2[]= ...
```

The technique is realized as an parallel implementation of F#s `map` function for arrays.

`Parallel.map` is used when point or node conversions are needed, such as projecting points to screen space or calculating the distances of points to a shape.

6.6 Point-Cloud Rendering

The culling heuristic described in Section 3.4 reduces the octree to set of nodes that can be processed by the GPU and can be drawn in real time. The point data may still be located on the hard drive and must be loaded into memory.

The Aardvark platform already provides a level-of-detail point cloud rendering system that can handle out-of-core datasets. The rendering system uses a cache to store nodes that were rendered previously. Once a frame is redrawn, all rendered nodes are collected, and for new nodes, the data is loaded into the memory. New nodes are added to the cache, nodes that are not used for this frame, are kept in the cache until the memory consumption requires removal of unused data. For each node new to the cache, a queue

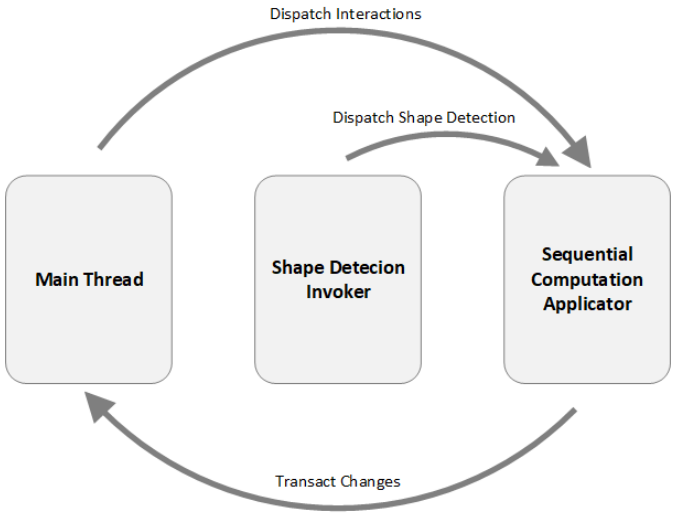
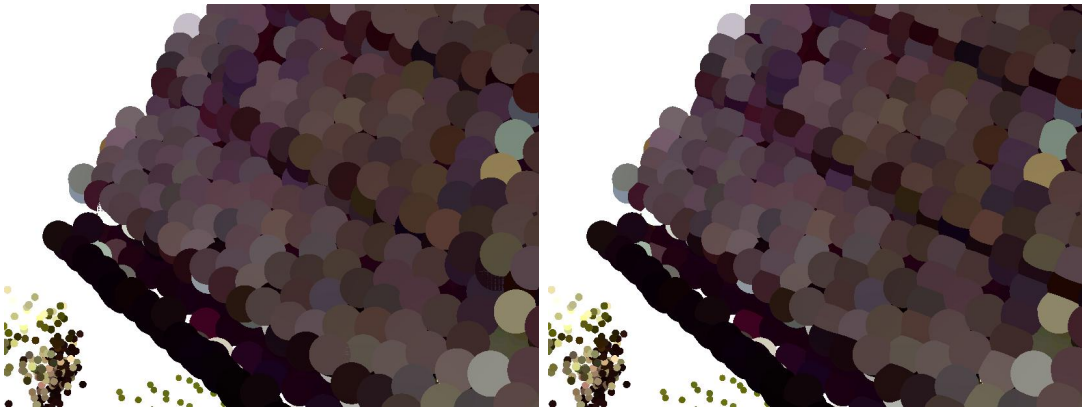


Figure 6.5: The main thread controls the rendering, mod evaluation and interactions. Point selections from interactions are handed to the applicator thread. The shape detection is invoked by a separate thread and dispatched on the applicator thread as well.



(a) (b)

Figure 6.6: This figure shows a direct comparison of using (a) circular splats and (b) sphere impostors with depth displacement. In (b) the intersections between points are visible due to the spherical shape, whereas points that are a little below other points are occluded almost entirely.

entry is added for a worker thread to load the point data into memory, stored in a working set. This working set reacts to changes like an `IMod` so that every time a new entry is created, re-evaluation is triggered and a frame is drawn. Thus, the point-cloud data that is loaded into memory appears gradually on the screen.

Mathematically, a point has no extent. Therefore it cannot be depicted directly. A common way to draw points is to interpret them as splats of certain size and shape. In this thesis, the points are represented by a sphere imposter. For each point, a camera-aligned quad is created in a geometry shader, whose size is the diameter of the sphere, pixels that are outside the radius of the sphere are discarded.

Popping effects occur when the view changes and overlapping impostors change their order. Similar to the approach by Schütz and Wimmer [SW15a], each splat is given a depth displacement that mimics a sphere's curvature. The splat is extruded towards the camera with a depth value dependent on the distance to the splat's center. This displacement causes the impostors to intersect naturally, thus reducing depth-related popping effects. Figure 6.6 shows a direct comparison of using spheres and circular splats.

CHAPTER 7

Results

All results were obtained on a PC with an AMD FX-9590 CPU, 16 GB RAM, and an AMD Radeon HD 7970 GPU. The datasets Technologiezentrum and JB_Haus are a courtesy of rmData [rV]. The techniques in this thesis are tested on three different datasets described in Table 7.1. The synthetic point cloud is constructed from a set of primitives that are discretized to point sets and includes approximations for all types of primitives.

Section 7.1 presents results for shape detection and shows benchmarks that verify the capability of the user-guided shape detection to be performed in interactive time on the test computer. Section 7.2 discusses flaws of the shape-detection algorithm that occurred while implementing this thesis. Section 7.3 shows comparisons of the classic RANSAC shape detection and our user-guided implementation. Section 7.4 presents performance measures for picking and selection and compares the performance of the assisted version with the traditional techniques. Section 7.5 concludes this chapter and provides a set of screenshots that show the workflow of the user-guided interactions on different data sets.

	#Points	**#Nodes**	**max. Depth**
JB_Haus.pts	620.722	440	15
Technologiezentrum_Teil1.pts	11.762.924	8863	15
Synthetic_Primitives.pts	472.000	315	5

Table 7.1: This table shows the different datasets used for testing. The first column describes the number of points, the second column describes the number of nodes in the octree, the third column describes the maximal depth (e.g., level of detail) in the octree.

7.1 Interactive Shape-Detection Performance

The goal of *user-guided shape detection* is to provide meaningful results within interactive time, such that detected shapes can be used to support interactions immediately. The capabilities of the implementation by Schnabel et al. [SW] are benchmarked on three different datasets in Table 7.1 using the interactive approach, as well as on the entire point cloud at a whole. The tests are performed without user interaction. Instead, all octree nodes are collected and fed into the shape-detection pipeline sequentially to retrieve results for each node of the point cloud independently. Therefore, shape detection is performed on the complete point cloud for each level of detail. Each octree node contains at most 5000 points. The results are averaged over five runs. Table 7.2 shows the per-node performance of the user-guided shape detection. Each node from the datasets is benchmarked five times and the results are averaged. It can be seen that detecting only planes is significantly faster than detecting all types of primitive shapes. Detecting all types of primitive shapes still, produces results within the desired time period.

Figure 7.1a shows the total number of shapes per level of detail. All three datasets share an increase in the number of shapes in the third quarter before a rapid decline in the highest level of detail. Figure 7.1b shows the number of shapes divided by the number of nodes per level of detail. Both figures emphasize that the majority of shapes is not found in the highest level of detail. The reason for this is that many leaf nodes only contain a handful of points or only contain a single structure. Figure 7.1c shows the averaged calculation time per the level of detail. Shape detection for smaller, denser nodes takes less time than for larger nodes with lower level of detail because in larger nodes, usually more shapes are found.

	#Shapes per Node			**Time per Node (ms)**			**#Shapes**
	min	**max**	**avg**	**min**	**max**	**avg**	**total**
JB_Haus*	0	6	1.31	0.026	578.664	22.396	576.40
JB_Haus	0	6	1.40	0.024	741.384	93.264	616.00
Technologiezentrum*	0	10	1.18	0.023	440.614	18.781	10458.34
Technologiezentrum	0	10	1.19	0.024	991.554	83.180	10546.97
Synthetic*	0	7	1.89	0.026	440.064	29.779	595.35
Synthetic	0	7	1.24	0.024	753.345	136.474	390.60

Table 7.2: This table presents per-node performance measures of the interactive shape detection. The number of shapes and durations listed are minimum, maximum, and average. The last column shows the number of shapes found in the entire dataset on all levels of detail (without clustering). Each dataset is benchmarked using plane detection only (items with *), as well as detecting all types of primitive shapes. The results are averaged over five runs.

The results of the original shape detection on the whole point cloud at once are shown in Table 7.3. The interactive approach processes the point cloud in chunks, resulting in the detected primitives being chunks of larger primitives as well. This explains the

	#Shapes			Time (s)		
	min	**max**	**avg**	**min**	**max**	**avg**
JB_Haus*	59	68	64	1.72	1.84	1.79
JB_Haus	61	72	66	5.45	5.76	5.60
Technologiezentrum*	664	699	689	265.00	296.70	285.84
Technologiezentrum	750	793	773	477.44	556.88	514.94
Synthetic*	7	18	13	3.66	4.42	4.01
Synthetic	11	12	12	3.68	13.68	8.39

Table 7.3: This table presents performance measures of the original shape detection for the different datasets. The shape detection is executed on the entire point cloud at once. Each dataset is benchmarked using plane detection only (items with *), as well as detecting all types of primitive shapes. The results are averaged over five runs.

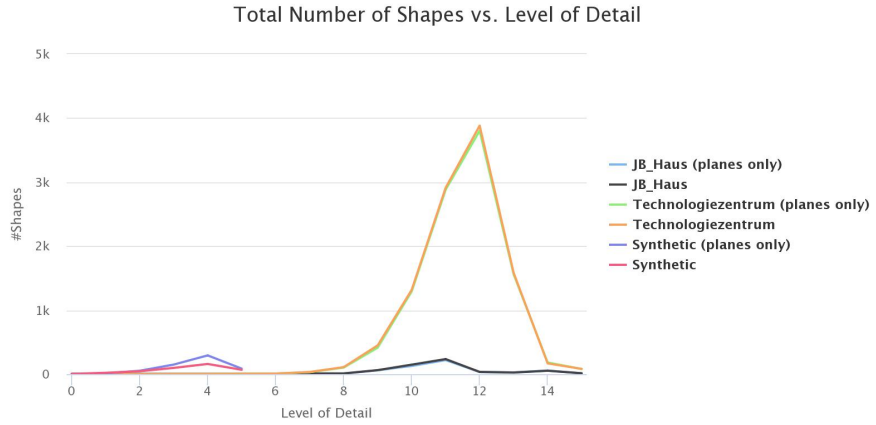
higher number of detected primitives using the interactive approach. While on small datasets, such as the JB_Haus, the original shape-detection approach produces results within seconds, larger datasets require significantly more computation time, in some of our tests up to ∼9 minutes.

7.2 Shape-Detection Problems and Undesired Behavior

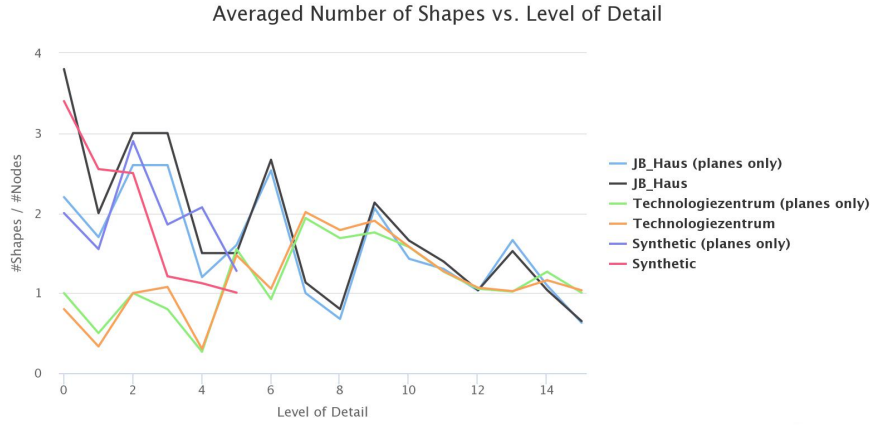
A problem with the shape-detection implementation by Schnabel et al. [SW] are reoccurring non-terminations for some octree nodes, causing the shape-detection coroutine to stall. To circumvent this problem, all shape detection tasks that are dispatched to the `sequential computation applicator` are assigned a timeout value of one second, after which the computation is interrupted.

Another recurring problem with the shape-detection algorithm is the plausibility of detected shapes. The RANSAC options guarantee that shapes are found that fit the local geometry within a certain margin, α for the normal angle and ϵ for the distance to the shape. So within theses two parameters, the shape is considered to be valid. However, certain constellations of points allow the shape detection to produce results that are accurate in regards to the RANSAC parameters but are implausible to the eye. A prominent example is a torus that is fitted onto a section that describes a cylinder. The major radius of the torus is of such dimensions that the local cylinder fits within the curvature of the torus. Thus, a torus is detected, rather than the simpler cylinder. Figure 7.2 shows this behavior within an example scene that consists of multiple primitives. Figure 7.2b shows the size of the detected torus.

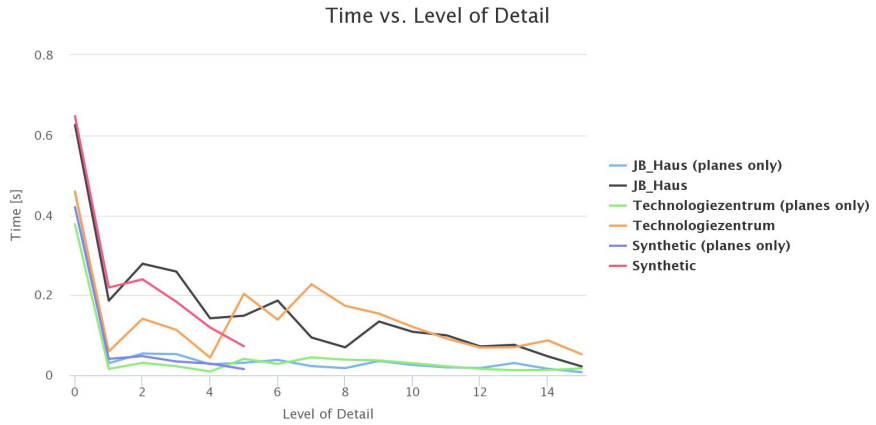
Such implausible shapes can exist due to the density-controlled ϵ parameter that weakens the margin for octree nodes of larger volume. Even within a node, the density can vary strongly for different regions. A way to counter this behavior is to create a ranking of the different types of shapes, such that certain types of shapes are detected preferably before other types. A second approach is to check the validity of a detected shape afterward



(a)



(b)



(c)

Figure 7.1: This set of figures shows different performance measures for the interactive shape detection. All values are averaged over all nodes that share the same level of detail. (a) shows a plot of the total number of shapes vs. the level of detail of the node, (b) shows a plot of the average number of shapes per node vs. the level of detail of the node, and (c) describes the average computation time for the shape detection for each specific level of detail.

by comparing the dimensions of the detected shape to the extents of the region it was detected in.

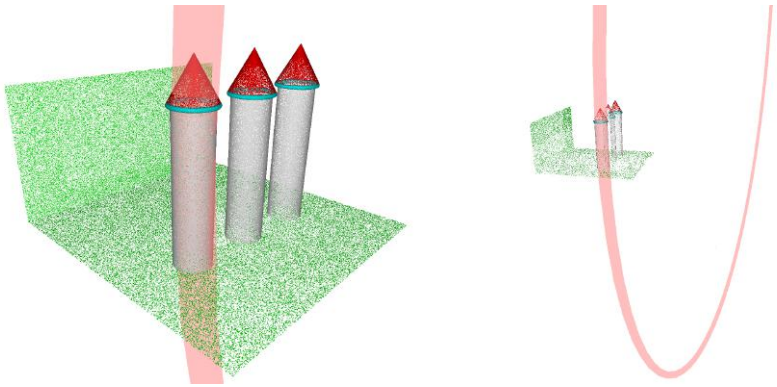
7.3 Shape-Detection Results

Figure 7.3 and 7.4 compares the results of the RANSAC shape detection on the entire dataset with our interactive approach. Our interactive method finds more shapes that can be grouped into one larger consecutive shape. However, the overall structure of the building is approximated similarly in both techniques as can be seen in (b) and (c) respectively.

The interactive shape detection was unable to produce results on the synthetic point cloud. On coarse levels of detail, implausible shapes are detected that occluded the entire scene. Hence, a visual comparison of the classic shape detection with our approach did not yield any satisfying results. The results of the classic shape detection for the synthetic scene can be seen in Figure 7.5

7.4 Interaction Performance

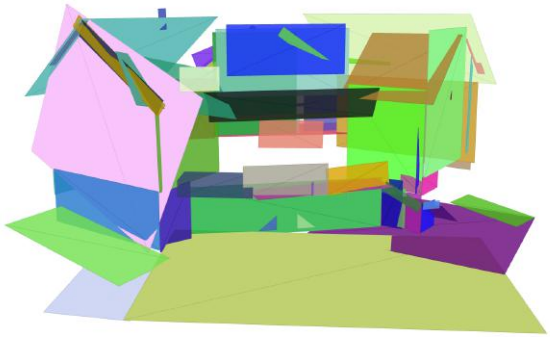
The use of support shapes for assisted interaction improves the computation time. *Shape-assisted point picking* is tested against traditional point picking. Point picking is performed only on points that are visible to the user. Hence, the size of the point cloud does not affect the picking performance. However, the level-of-detail culling does. The results in Table 7.4 are obtained by randomly picking points with and without support shape on all three datasets. Over all datasets, shape-assisted point picking performs better



(a) (b)

Figure 7.2: This figure shows a cylinder from the synthetic point cloud whose points are classified as a torus instead of a cylinder. Even though the points fit the torus, determined by the RANSAC options, the result is not plausible, since the user would expect a cylinder for this constellation of points.

(a)

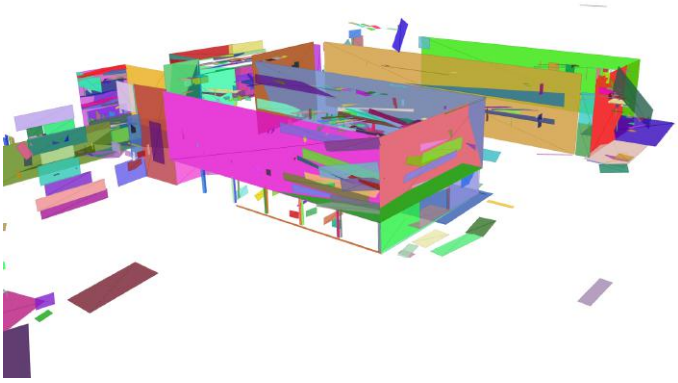


(b)

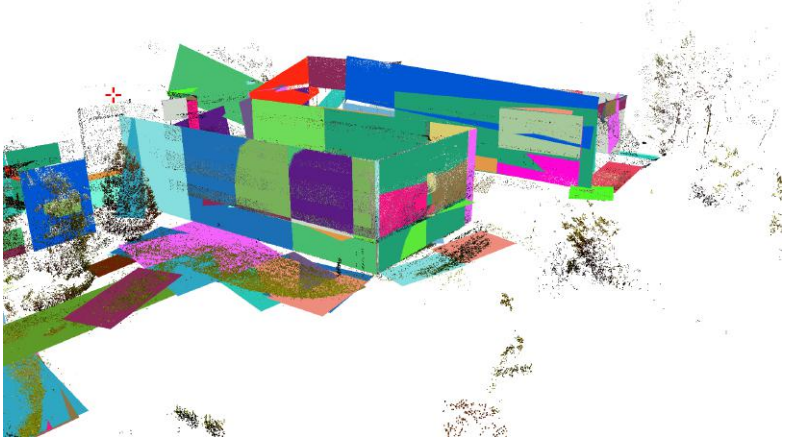
(c)

Figure 7.3: Figure (a) shows a rendering of the JB_haus dataset, (b) shows the resulting shapes of the RANSAC shape detection, (c) shows the detected shapes using the interactive shape-detection method.

(a)

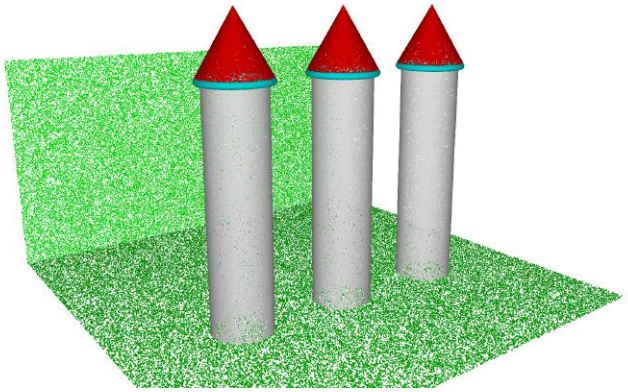


(b)

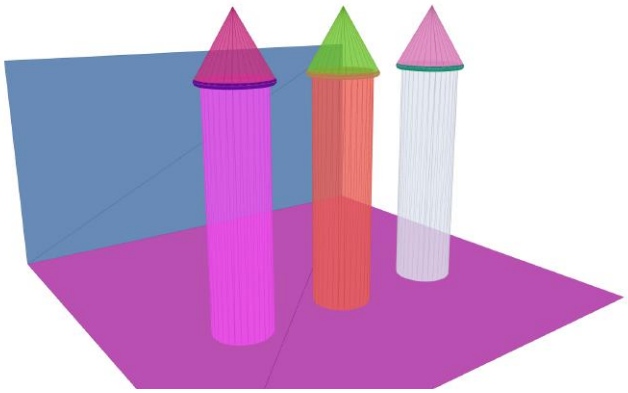


(c)

Figure 7.4: Figure (a) shows a rendering of the Technologiezentrum dataset, (b) shows the resulting shapes of the RANSAC shape detection, (c) shows the detected shapes using the interactive shape-detection method.



(a)



(b)

Figure 7.5: (a) shows the synthetic point cloud, consisting of two planes, three cylinders, three tori, and three cones. (b) shows the detected shapes rendered as triangle meshes. For each shape in the point cloud, the RANSAC shape detection has found a suitable primitive shape.

	Time (ms)		
	min	**max**	**avg**
Without support	13.402100	148.346100	37.964309
With support	0.348200	39.929400	3.281752

Table 7.4: This table showcases the benchmark results for point picking with and without the use of a support shape. The assisted technique performs on average more than ten times faster than traditional point picking.

	Time (ms)		
	min	**max**	**avg**
Without support	193.628900	2494.494800	748.725405
With support	191.082600	923.332500	437.933686

Table 7.5: This table showcases the benchmark results for shape-assisted lasso selection compared to traditional lasso selection. On average, shape-assisted lasso selection performs faster. However, performance strongly depends on the size of the point cloud and selection.

than traditional point picking. For shape-assisted point picking, the actual per-point processing time is higher due to the additional point-shape relation check. However, the assisted interaction performs significantly better due to the reason that traditional point picking must process a larger set of octree nodes (i.e., all nodes that intersect the pick ray), whereas shape-assisted point picking only needs to check nodes that intersect the support shape.

The lasso selection is performed on the entire point cloud. Hence, the size of the point cloud directly influences the response time of the interaction. Furthermore, the size and form of a lasso also affect the computation time. For a rectangular selection, it takes significantly less time to compute the intersection with a point than for a more complex polygon. The shape-assisted lasso selection is benchmarked by directly comparing the computation time to a classical lasso selection that shares the same lasso polygon. The performance measures in Table 7.5 are obtained from selection on all datasets.

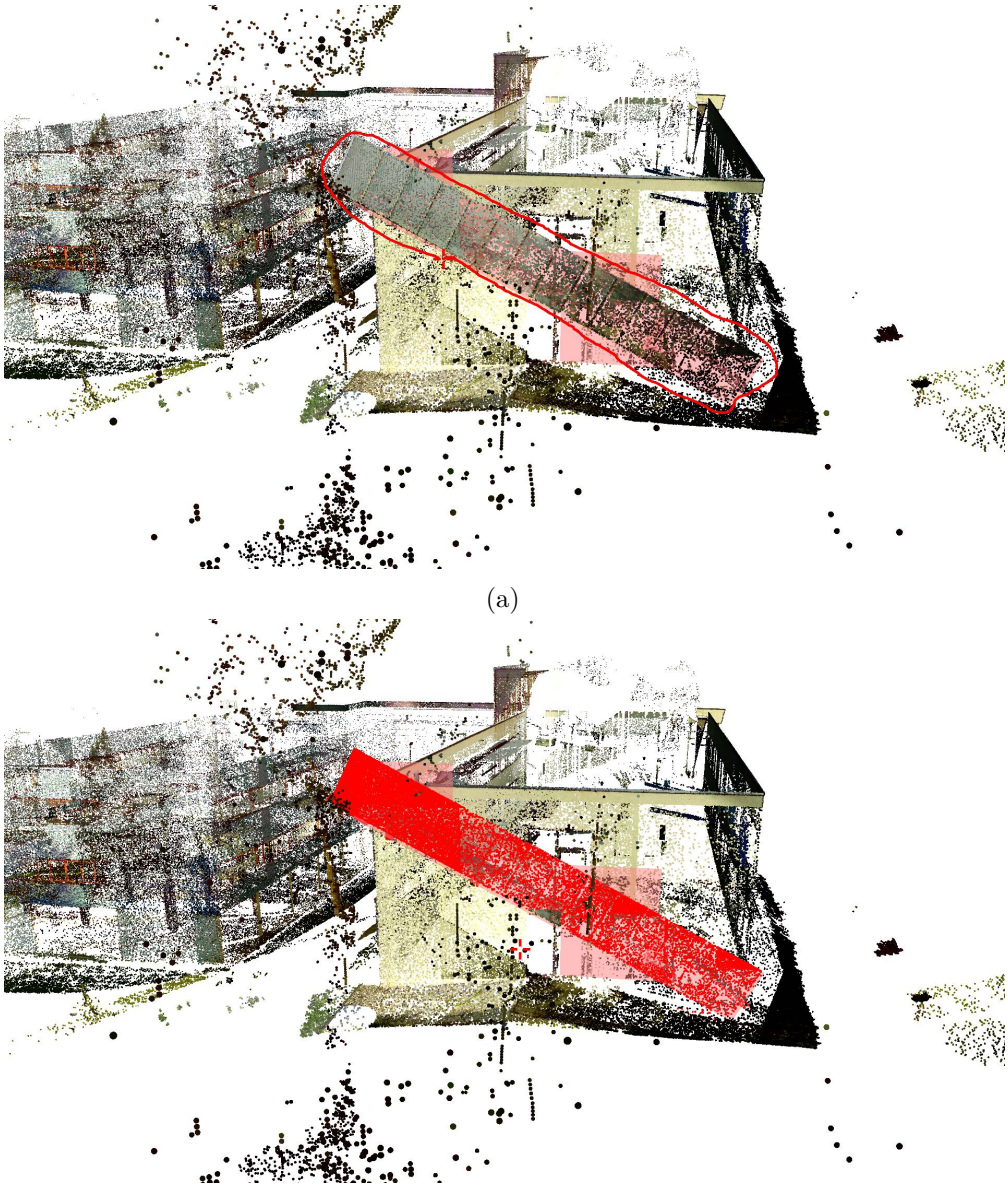
The testing methodology for the lasso selection was to perform the shape-assisted and the traditional lasso selection for the same lasso polygon. Hence, the testing provided a direct comparison of the interactions for the same lasso and the same point cloud. We calculated the ratio between the processing times of both techniques. We obtained an averaged ratio between the shape-assisted and traditional lasso selection of $\sim$1.81, meaning that the traditional lasso selection takes on average approximately 1.8 times as long to compute as the shape-assisted lasso selection.

7.5 Interaction Results

This section presents a set of figures that showcase the different interactions from Section 5.4. Examples of all interactions using the JB_Haus dataset are presented already throughout this thesis. This section shows the results for the interactions on both the Technologiezentrum and the synthetic dataset. The synthetic dataset showcases the use of non-planar support shapes for the assisted interactions.

Figures 7.6, 7.7, and 7.8 show examples of a shape-assisted lasso selection, volumetric-brush selection and level-of-detail increment using a support shape on the Technologiezentrum dataset.

Figures 7.9 and 7.10 show a shape-assisted lasso selection and volumetric-brush selection. The support shape is a cylinder. While the volumetric brush still only follows the surface of the cylinder, the lasso selects partly 'through' the point set, more precisely, the lasso selection is performed on points on the back side of the cylinder as well.

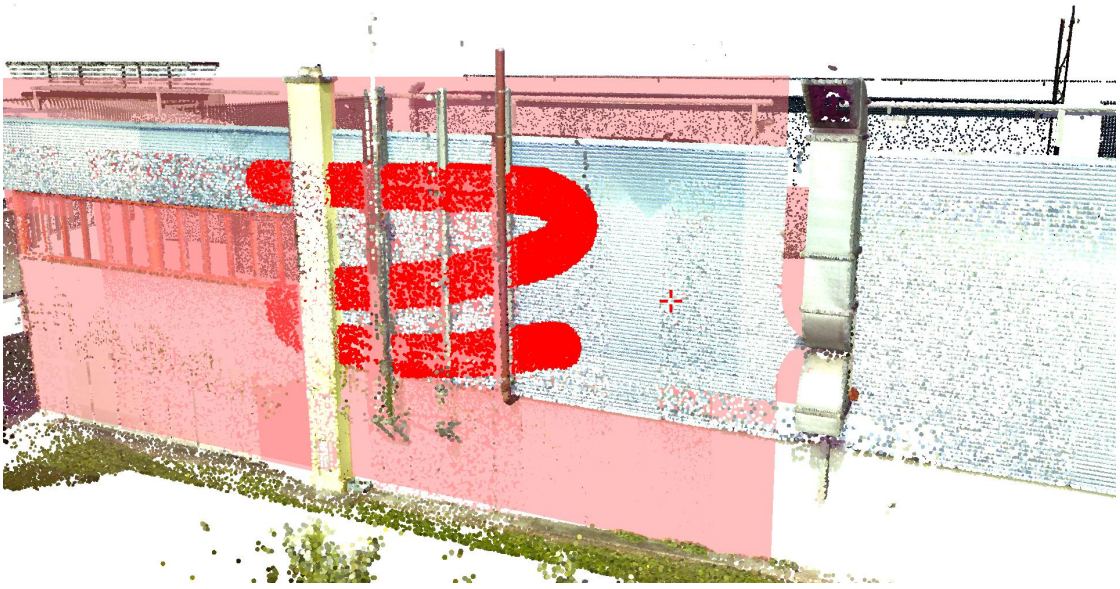


(a)

(b)

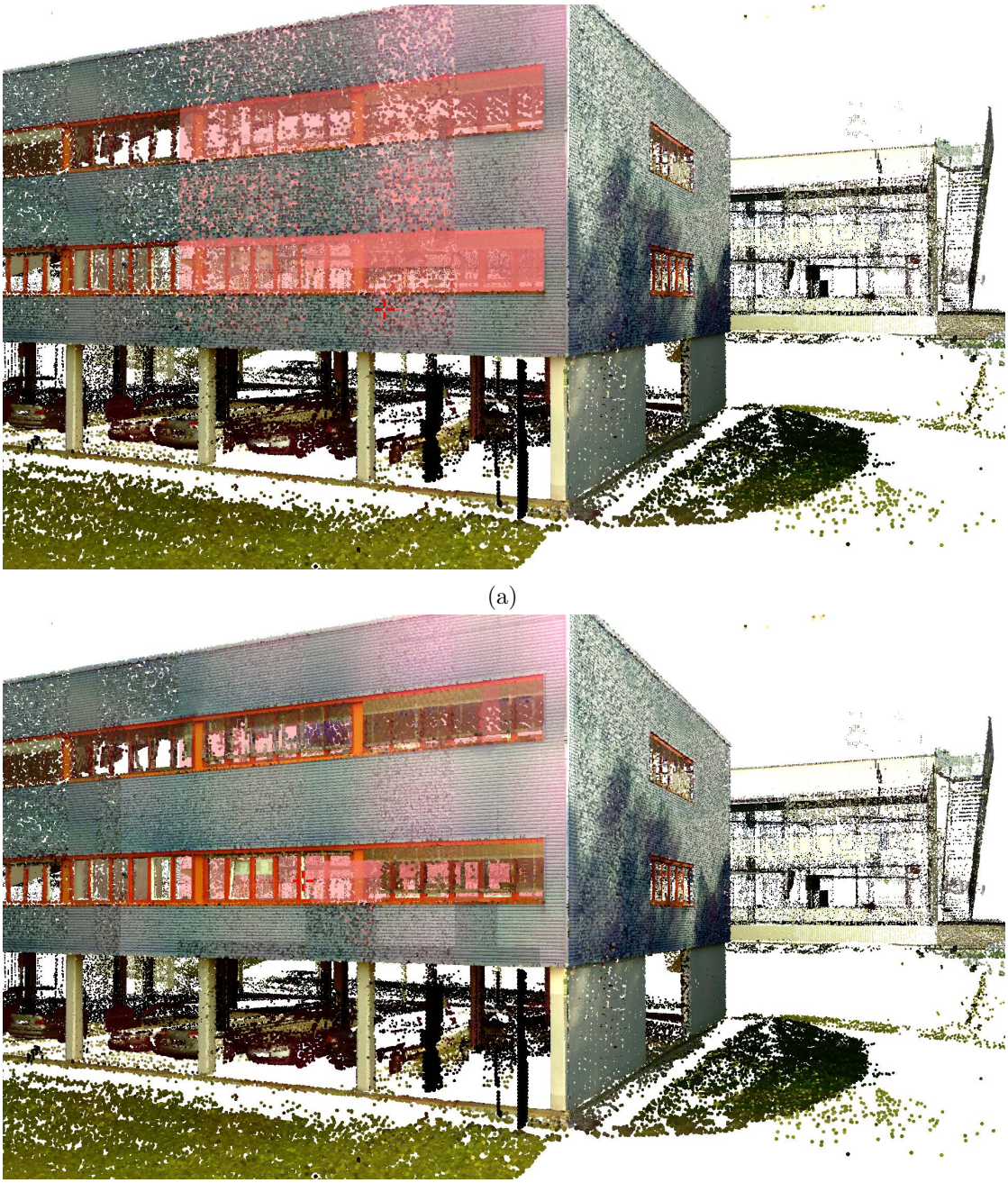
Figure 7.6: A lasso selection is performed on the selected support shape in (a). Only points are selected that lie on the support shape as shown in (b). Points in front and back of the support shape are not selected.

(a)



(b)

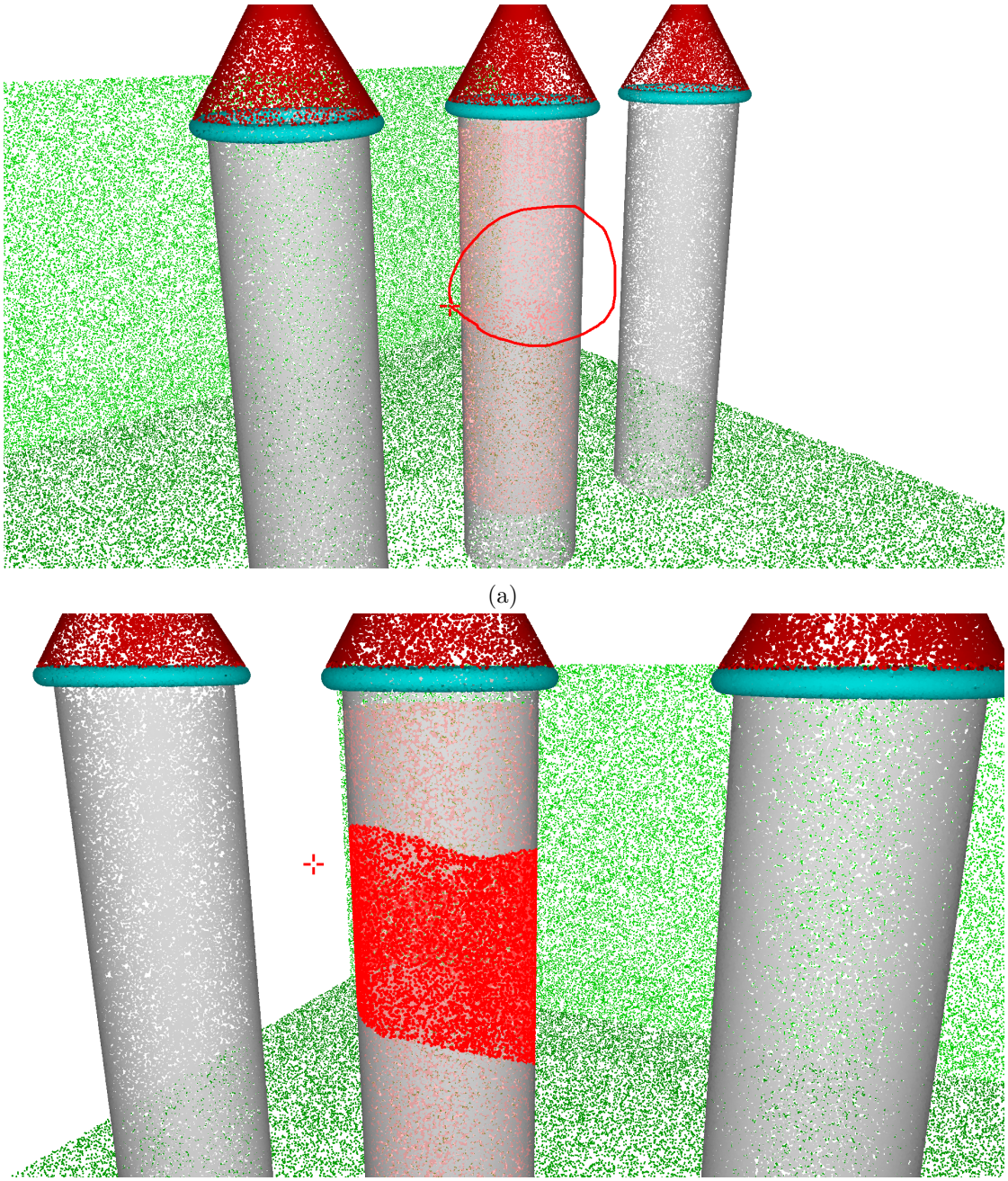
Figure 7.7: A volumetric brush selection is performed on the selected support shape in (a). Points are only selected if they belong to the support shape and intersect the brush. The result of the selection can be seen in (b).



(a)

(b)

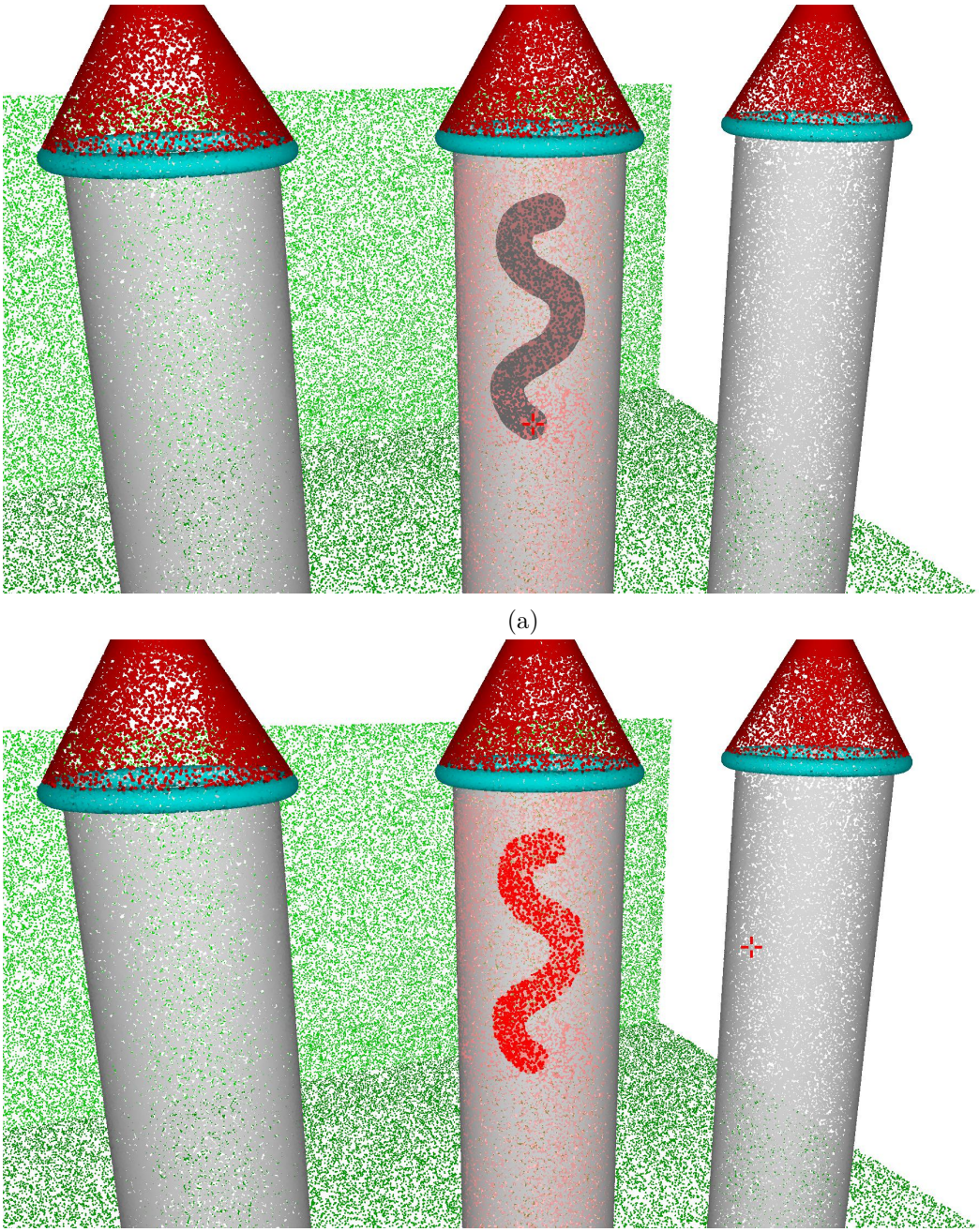
Figure 7.8: This figure shows the use of the level-of-detail increment interaction. The level of detail is incremented along the selected support shape (drawn in red). (a) shows the original rendering model of the point cloud, (b) shows the point cloud with additional points.



(a)

(b)

Figure 7.9: This figure shows an improved lasso selection performed using a cylinder shape as support. (a) shows the lasso that is drawn on the screen, (b) shows the selection result from a different angle. Points in the back of the cylinder are selected, as they are approximated by the cylinder as well.



(a)

(b)

Figure 7.10: This figure shows the volumetric brush selection on a selected cylinder shape. The brush sticks to the side of the cylinder that is facing the camera.

CHAPTER 8

Conclusion and Future Work

This chapter completes this thesis by summarizing the contributions of the work and proposing future work.

Multi-Resolution Point Cloud Representation

This thesis presents an out-of-core octree data structure that stores the point-cloud data in a cache file on the hard disc, thus allowing for datasets whose size exceeds the available system memory. The octree stores random subsets of the original point cloud at different resolutions in its intermediate nodes. The point cloud is stored in a multi-resolution representation similar to the octree by Wand et al. [WBB+07], with the difference that their approach uses averaged subsets to build the multiple resolutions rather than random subsets. The reason why this thesis favors random subsamples over averaged subsamples is that averaging points creates new points that are not present in the original data which might have a negative impact on processing results. The Instant Point system [WS06] stores random subsamples in the intermediate nodes as well but does not keep copies of the points in the higher levels of detail. Furthermore, the point set within an octree node is stored as an rkd-tree [Tob11] to accelerate point queries. The use of random subsets like Instant Points, multiple resolutions like Wand et al., and an rkd-tree presents a unique combination that can used for fast processing on multiple levels of detail as well as rendering.

User-Guided Shape Detection and Clustering

This thesis shows an alternative use of the shape-detection algorithm by Schnabel et al. [SWK07a] that lets the user control the regions that are segmented. By pointing to a region with the mouse, the system selects the most suitable octree node to be segmented. The octree's split decision ensures a consistent number of points per shape, so that the shape detection delivers results in interactive time. The shape detection is performed on

the highest visible level of detail. Hence, this approach creates a multi-scale representation of the shapes by performing shape detection on nodes with different level of detail. To account for different point distributions in octree nodes, a dynamic ϵ threshold, based on the node's point density, is proposed for the shape detection.

As the size of a single shape is limited to the extent of the octree node it was detected in, this thesis proposes a shape clustering algorithm that determines if two shapes are matching and creates a larger coherent shape from multiple matching primitive shapes of different levels of detail that is then used to assist user interactions.

Improvement of Interaction Techniques

Several improvements to well-known two-dimensional interaction techniques are discussed in this thesis. We propose a way of pre-filtering points for interactions such that only points are considered that are approximated by a support shape, picked by the user. Classic point picking is improved so that only points are picked that belong to the support shape. Only using points that belong to a shape is especially useful when trying to pick points that are otherwise occluded or lie on the edge of a structure. Multiple ways exist to select regions in a point cloud. We utilize a support shape to improve the lasso selection. Classic lasso selection selects all points, whose projection lies inside the lasso on the near plane. The user selects 'through' the point cloud, whereas when using a support shape, the user only selects points that lie on this support shape. The volumetric brush allows the user to select points that are in the foreground. Instead of consulting the depth buffer to retrieve the brush's position, the position of the cursor on the support shape is used. Thus, the brush does not follow the structures that are in the foreground but the curvature of the support shape and points can be selected that were occluded otherwise.

Novel Interaction to Magnify Local Details

The final task of this thesis was to design a novel interaction technique called *shape-assisted local level-of-detail increment*. The level-of-detail rendering of the point cloud displays only a subset of points that can be rendered by the graphics card in real time. Thus, details may get culled and are not rendered. The interaction technique collects points from nodes that are not rendered due to their level-of-detail. These additional points are approximated by the support shape. Along the support shape, extra points are rendered that would otherwise be culled, therefore, allowing the user to get a more detailed look at structures.

8.1 Future Work

As the focus of this thesis was the design of the user-guided shape detection and the assisted interactions, future work will focus on performance and robustness improvements on various fronts. The usage of a selection data structure, such as a selection octree [SW11], can improve the performance when selecting or editing the point cloud. Also,

different ways can be explored to improve computation speed by taking advantage of the parallel architecture of the graphics card.

As the selection processing is performed asynchronously, feedback is not displayed immediately, but with a delay. To overcome this delay between interaction and receiving the result, a visual selection can be performed on the GPU [Rai16] to bridge this gap. This technique utilizes volumetric shadows to create a visual selection using the GPU's stencil buffer.

Shape detection is performed using an external library by Schnabel et al. [SW]. Multiple problems occur, such as non-termination or non-plausible shape matching. More work must be carried out to limit or eliminate these problems. The robustness of the shape detection, especially when detecting non-planar primitive shapes, suffers due to weak constraints. Alternatively, the shape detection can be implemented in such a way, that particular types of shapes are prioritized to reduce the amount of non-plausible shapes.

In the current application, edges between nodes with different levels of detail are visible. Regions with sparse points contain visible gaps. Adaptive point sizes [Sch14] could be used to reduce the visual artifacts that come with this type of level-of-detail rendering. The individual point size for each octree node is controlled by the number of points and the weighted level of detail. Points in octree nodes with fewer points but larger extent have a larger size.

The current point-cloud rendering system draws points multiple times. The overdraw of redundant points accounts for $\sim$12% of the entire point budget. Future work on this domain will aim at the reduction of this overdraw. An intermediate node needs to be drawn only if the level-of-detail decision culls one of its children. If this is the case, the parent node contains non-duplicate point information that is not displayed by one of its children.

List of Figures

List of Tables

Bibliography

[And79] Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[ASF+13] Murat Arikan, Michael Schwärzler, Simon Flöry, Michael Wimmer, and Stefan Maierhofer. O-snap: Optimization-based snapping for modeling architecture. *ACM Transactions on Graphics*, 32:6:1–6:15, January 2013.

[BHZK05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings*, pages 17–141. IEEE, 2005.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 657–662. ACM, 2003.

[EBN13] Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. One billion points in the cloud–an octree for efficient processing of 3d laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing*, 76:76–88, 2013.

[EDF08] Niklas Elmqvist, Pierre Dragicevic, and Jean-Daniel Fekete. Rolling the dice: Multidimensional visual exploration using scatterplot matrix navigation. *IEEE transactions on Visualization and Computer Graphics*, 14(6):1539–1148, 2008.

[F#05] F# Software Foundation. F#. http://fsharp.org/, 2005. Accessed: 2017-07-11.

[FB81] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[FBS75] Jerome H Friedman, Forest Baskett, and Leonard J Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on computers*, 100(10):1000–1006, 1975.

[FS75] Herbert Freeman and Ruth Shapira. Determining the minimum-area encasing rectangle for an arbitrary closed curve. *Communications of the ACM*, 18(7):409–413, 1975.

[fVRuVFGa] VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH. Aardvark-vrvis. https://www.vrvis.at/research/projects/aardvark. Accessed: 2017-07-11.

[fVRuVFGb] VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH. Vrvis. https://www.vrvis.at/. Accessed: 2017-07-11.

[GKF09] Aleksey Golovinskiy, Vladimir G Kim, and Thomas Funkhouser. Shape-based recognition of 3d point clouds in urban environments. In *2009 IEEE 12th International Conference on Computer Vision*, pages 2154–2161. IEEE, 2009.

[GM04] Enrico Gobbetti and Fabio Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6):815–826, 2004.

[HSM+14a] Georg Haaser, Harald Steinlechner, Michael May, Michael Schwärzler, Stefan Maierhofer, and Robert Tobler. Cosmo: Intent-based composition of shader modules. In *Computer Graphics Theory and Applications (GRAPP), 2014 International Conference on*, pages 1–11. IEEE, 2014.

[HSM+14b] Georg Haaser, Harald Steinlechner, Michael May, Michael Schwärzler, Stefan Maierhofer, and Robert Tobler. Semantic composition of language-integrated shaders. In *International Joint Conference on Computer Vision, Imaging and Computer Graphics*, pages 45–61. Springer, 2014.

[HSMT15] Georg Haaser, Harald Steinlechner, Stefan Maierhofer, and Robert F Tobler. An incremental rendering vm. In *Proceedings of the 7th Conference on High-Performance Graphics*, pages 51–60. ACM, 2015.

[HWZF14] Ming Huang, Yanmin Wang, Yong Zhang, and Xinle Fu. Pickup of large scale point cloud based on gpu. *BioTechnology: An Indian Journal*, 10(23), 2014.

[IMT07] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: a sketching interface for 3d freeform design. In *Acm siggraph 2007 courses*, page 21. ACM, 2007.

[ISO12] ISO. International Organization for Standardization. ISO/IEC 23271:2012. https://www.iso.org/standard/58046.html, 2012. Accessed: 2017-07-11.

[Jen08] Philipp Jenke. Surface reconstruction from fitted shape primitives. 2008.

[Jol02] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[LBCW05] John Lucas, Doug Bowman, Jian Chen, and Chad Wingrave. Design and evaluation of 3d multiple object selection techniques. Technical report, Virginia Polytechnic Institute and State University, 2005.

[LC87] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.

[LWC+11] Yangyan Li, Xiaokun Wu, Yiorgos Chrysathou, Andrei Sharf, Daniel Cohen-Or, and Niloy J Mitra. Globfit: Consistently fitting primitives by discovering global relations. In *ACM Transactions on Graphics (TOG)*, volume 30, page 52. ACM, 2011.

[Mica] Microsoft. C# Language Specification. https://msdn.microsoft.com/en-us/library/ms228593(v=vs.110).aspx. Accessed: 2017-07-11.

[Micb] Microsoft. Microsoft - official home page. https://www.microsoft.com/de-at/. Accessed: 2017-07-11.

[Micc] Microsoft. .NET Powerful Open Source Cross Platform Development. https://www.microsoft.com/net. Accessed: 2017-07-11.

[MLM01] David Marshall, Gabor Lukacs, and Ralph Martin. Robust segmentation of primitives from range data in the presence of geometric degeneracy. *IEEE Transactions on pattern analysis and machine intelligence*, 23(3):304–314, 2001.

[MV99] Hans-Gerd Maas and George Vosselman. Two algorithms for extracting building models from raw laser altimetry data. *ISPRS Journal of photogrammetry and remote sensing*, 54(2):153–163, 1999.

[OBKI04] Jens Overby, Lars Bodum, Erik Kjems, and PM Iisoe. Automatic 3d building reconstruction from airborne laser scanning and cadastral data using hough transform. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 34:296–301, 2004.

[OLA16] Sven Oesau, Florent Lafarge, and Pierre Alliez. Planar shape detection and regularization in tandem. In *Computer Graphics Forum*, volume 35, pages 203–215. Wiley Online Library, 2016.

[OTDS04] K Oda, T Takano, T Doihara, and R Shibaski. Automatic building extraction and 3-d city modeling from lidar data based on hough transform. *International Archives of Photogrammetry and Remote Sensing*, 35(B3):277–281, 2004.

[PJW12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In *EGPGV*, pages 139–148, 2012.

[PZVBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342. ACM Press/Addison-Wesley Publishing Co., 2000.

[Rai16] Bernhard Rainer. Selection Visualization. https://github.com/JimmyLaessig/SelectionVisualization, 2016. Accessed: 2017-08-02.

[RC11] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4. IEEE, 2011.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

[rV] rmData Vermessung. rmData Vermessung Österrreich - Software für Vermessung, Planeerstellung und Geoinformation. http://www.rmdata.at/. Accessed: 2017-07-11.

[RVDH05] Tahir Rabbani and Frank Van Den Heuvel. Efficient hough transform for automatic detection of cylinders in point clouds. *Isprs Wg Iii/3, Iii/4*, 3:60–65, 2005.

[Sch14] Claus Scheiblauer. *Interactions with Gigantic Point Clouds*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2014.

[Sch16] Markus Schütz. Potree: Rendering large point clouds in web browsers. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, September 2016.

[SDK09] Ruwen Schnabel, Patrick Degener, and Reinhard Klein. Completion and reconstruction with primitive shapes. *Computer Graphics Forum (Proc. of Eurographics)*, 28(2):503–512, March 2009.

[SP11] Claus Scheiblauer and Michael Pregesbauer. Consolidated visualization of enormous 3d scan point clouds with scanopy. In *Proceedings of the 16th International Conference on Cultural Heritage and New Technologies*, pages 242–247, 2011.

[SW] Ruwen Schnabel and Roland Wahl. Software 1.1. http://cg.cs.uni-bonn.de/aigaion2root/attachments/Software%20v1.1.zip. Accessed: 2017-07-24.

[SW11] Claus Scheiblauer and Michael Wimmer. Out-of-core selection and editing of huge point clouds. *Computers & Graphics*, 35(2):342–351, 2011.

[SW15a] Markus Schütz and Michael Wimmer. High-quality point based rendering using fast single pass interpolation. In *Proceedings of Digital Heritage 2015 Short Papers*, pages 369–372, September 2015.

[SW15b] Markus Schütz and Michael Wimmer. High-quality point-based rendering using fast single-pass interpolation. In *Digital Heritage, 2015*, volume 1, pages 369–372. IEEE, 2015.

[SWK07a] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. Efficient ransac for point-cloud shape detection. *Computer Graphics Forum*, 26(2):214–226, June 2007.

[SWK07b] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. Ransac based out-of-core point-cloud shape detection for city-modeling. *Schriftenreihe des DVW, Terrestrisches Laser-Scanning (TLS 2007)*, 2007.

[TKLG+07] Fayez Tarsha-Kurdi, Tania Landes, Pierre Grussenmeyer, et al. Hough-transform and extended ransac algorithms for automatic detection of 3d building roof planes from lidar data. In *Proceedings of the ISPRS Workshop on Laser Scanning*, volume 36, pages 407–412, 2007.

[Tob11] Robert F Tobler. The rkd-tree: An improved kd-tree for fast n-closest point queries in large point sets. In *Proceedings of Computer Graphics International*, volume 2011, 2011.

[VC62] Hough Paul VC. Method and means for recognizing complex patterns, December 18 1962. US Patent 3,069,654.

[WBB+07] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. Interactive editing of large point clouds. In *SPBG*, pages 37–45, 2007.

[WPK+04] Tim Weyrich, Mark Pauly, Richard Keiser, Simon Heinzle, Sascha Scandella, and Markus H Gross. Post-processing of scanned 3d surface data. *SPBG*, 4:85–94, 2004.

[WRFH14] K Wenzel, M Rothermel, D Fritsch, and N Haala. An out-of-core octree for massive point cloud processing. In *PROCEEDINGS, IQMULUS 1ST WORKSHOP ON PROCESSING LARGE GEOSPATIAL DATA*, page 53, 2014.

[WS06] Michael Wimmer and Claus Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. In *SPBG*, pages 129–136, 2006.

[WSMT13] Michael Wörister, Harald Steinlechner, Stefan Maierhofer, and Robert F Tobler. Lazy incremental computation for efficient scene graph rendering. In *Proceedings of the 5th high-performance graphics conference*, pages 53–62. ACM, 2013.

[YEII12] Lingyun Yu, Konstantinos Efstathiou, Petra Isenberg, and Tobias Isenberg. Efficient structure-aware selection techniques for 3d point cloud visualizations with 2dof input. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2245–2254, 2012.

[ZDWX08] Ming-liang ZHU, Bing DONG, Yi WANG, and Bu-ying XIE. Algorithm for picking in 3d scenes based on viewport space [j]. *Journal of Engineering Graphics*, 2:84–97, 2008.

[ZLL09] Jia-hua ZHANG, Cheng LIANG, and Gui-qing LI. 3d primitive picking on gpu [j]. *Journal of Engineering Graphics*, 1:10, 2009.

[ZPVBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378. ACM, 2001.