FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interactions with Gigantic Point Clouds

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Claus Scheiblauer
Matrikelnummer 9125272

an der
Fakultät für Informatik der Technischen Universität Wien

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

Diese Dissertation haben begutachtet:

(Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer)

(Professor Dr. Reinhard Klein)

Wien, 25.06.2014

(Claus Scheiblauer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interactions with Gigantic Point Clouds

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Claus Scheiblauer
Registration Number 9125272

to the Faculty of Informatics
at the Vienna University of Technology

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

The dissertation has been reviewed by:

(Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer)	(Professor Dr. Reinhard Klein)

Wien, 25.06.2014

(Claus Scheiblauer)

Erklärung zur Verfassung der Arbeit

Claus Scheiblauer
Kranzbichlerstraße 41/63, 3100 St.Pölten

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.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I would like to thank all the people who have contributed to this dissertation in some way, either by writing code for Scanopy, testing workflows, giving bug reports, or providing insights. These are, in approximately chronological order, Michael Wimmer, Michael Gervautz, Wolfgang Knecht, Stefan Marek, Gabriele Hebart, Fritz-Michael Gschwantner, Norbert Zimmermann, Irmengard Mayer, Verena Fugger, Yaroslav Barsukov, Reinhold Preiner, Julian Mayer, Michael Pregesbauer, Markus Tragust, Murat Arikan, and Kurt Leimer.

Furthermore, I would like to thank the Austrian Science Fund (FWF) START Project "The Domitilla-Catacomb in Rome. Archaeology, Architecture and Art History of a Late Roman Cemetery", made possible by commission and with the help of the Pontificia Commissione di Archeologia Sacra/Roma, for providing access to the point model of the Domitilla catacomb.

I would like to thank Norbert Zimmermann and the Austrian Academy of Sciences (ÖAW) for providing access to the point model of Centcelles. I would like to thank Michael Pregesbauer and the Office of the Government of Lower Austria, Department for Hydrology and Geoinformation (Amt der Niederösterreichischen Landesregierung, Abteilung für Hydrologie und Geoinformation) for provinding access to the point model of the Amphitheater 1. Finally, I would like to thank the Dombauhütte zu St.Stephan for providing access to the point model of the Stephansdom.

The work for this dissertation was funded by the Austrian Research Promotion Agency (FFG) Bridge Project SCANOPY, by the FFG FIT-IT Project TERAPOINTS, and by the EU FP7 Project Harvest 4D.

Abstract

During the last decade the increased use of laser range-scanners for sampling the environment has led to gigantic point cloud data sets. Due to the size of such data sets, tasks like viewing, editing, or presenting the data have become a challenge per se, as the point data is too large to fit completely into the main memory of a customary computer system. In order to accomplish these tasks and enable the interaction with gigantic point clouds on consumer grade computer systems, this thesis presents novel methods and data structures for efficiently dealing with point cloud data sets consisting of more than 10^9 point samples.

To be able to access point samples fast that are stored on disk or in memory, they have to be spatially ordered, and for this a data structure is proposed which organizes the points samples in a level-of-detail hierarchy. Point samples stored in this hierarchy cannot only be rendered fast, but can also be edited, for example existing points can be deleted from the hierarchy or new points can be inserted. Furthermore, the data structure is memory efficient, as it only uses the point samples from the original data set. Therefore, the memory consumption of the point samples on disk, when stored in this data structure, is comparable to the original data set. A second data structure is proposed for selecting points. This data structure describes a volume inside which point samples are considered to be selected, and this has the advantage that the information about a selection does not have to be stored at the point samples.

In addition to these two previously mentioned data structures, which represent novel contributions for point data visualization and manipulation, methods for supporting the presentation of point data sets are proposed. With these methods the user experience can be enhanced when navigating through the data. One possibility to do this is by using regional meshes that employ an out-of-core texturing method to show details in the mesoscopic scale on the surface of sampled objects, and which are displayed together with point clouds. Another possibility to increase the user experience is to use graphs in 3D space, which helps users to orient themselves inside point cloud models of large sites, where otherwise it would be difficult to find the places of interest. Furthermore, the quality of the displayed point cloud models can be increased by using a point size heuristics that can mimic a closed surface in areas that would otherwise appear undersampled, by utilizing the density of the rendered points in the different areas of the point cloud model.

Finally, the use of point cloud models as a tool for archaeological work is proposed. Since it becomes increasingly common to document archaeologically interesting monuments with laser scanners, the number application areas of the resulting point clouds is raising as well. These include, but are not limited to, new views of the monument that are impossible when studying the monument on-site, creating cuts and floor plans, or perform virtual anastylosis.

All these previously mentioned methods and data structures are implemented in a single software application that has been developed during the course of this thesis and can be used to interactively explore gigantic point clouds.

Kurzfassung

In den letzten Jahrzehnten hat der verstärkte Einsatz von Laserabtastgeräten zur Aufnahme der Umwelt zu gigantischen Punktwolkendatensätzen geführt. Wegen der Größe dieser Datensätze wurden Aufgaben wie das Betrachten, Bearbeiten, oder Präsentieren der Punktedaten zu einem eigenständigen Problem, da die Datensätze zu umfangreich sind um komplett in den Hauptspeicher eines handelsüblichen Computers zu passen. Um diese Aufgaben dennoch ausführen zu können, und die Interaktion mit gigantischen Punktwolken auf einem handelsüblichen Computer zu ermöglichen, werden in dieser Dissertation neuartige Methoden und Datenstrukturen präsentiert, um effizient mit Punktwolkendatensätzen umgehen zu können die aus mehr als 10^9 Einzelpunkten bestehen.

Um schnell auf Einzelpunkte zugreifen zu können die auf einer Festplatte oder im Speicher abgelegt sind, müssen sie räumlich sortiert werden, und dafür wird eine Datenstruktur vorgeschlagen, die die Einzelpunkte in eine Hierarchie von Detailgraden einordnet. Einzelpunkte die in diese Hierarchie eingeordnet wurden können nicht nur schnell dargestellt werden, sondern sie können auch bearbeitet werden, zum Beispiel können bestehende Punkte aus der Hierarchie gelöscht werden oder neue Punkte können eingefügt werden. Darüberhinaus ist die Datenstruktur speichereffizient, da nur die Einzelpunkte aus dem originalen Datensatz verwendet werden. Deshalb ist der Speicherverbrauch der Einzelpunkte auf der Festplatte, wenn sie in diese Datenstruktur eingeordnet sind, vergleichbar zum Speicherverbrauch des originalen Datensatzes. Eine zweite Datenstruktur wird vorgeschlagen um Punkte auszuwählen. Diese Datenstruktur beschreibt ein Volumen, innerhalb dessen die Einzelpunkte als ausgewählt betrachtet werden. Das bietet den Vorteil, daß die Information über die Auswahl nicht bei den Einzelpunkten selbst gespeichert werden muß.

Zusätzlich zu den beiden oben erwähnten Datenstrukturen, welche neue Beiträge zur Punktdatendarstellung und Punktdatenmanipulation darstellen, werden Methoden vorgeschlagen, welche die Präsentation von Punktdatensätzen unterstützen. Mit diesen Methoden kann das Nutzungserlebnis verbessert werden, wenn man durch die Daten navigiert. Eine Möglichkeit das zu erreichen ist es, lokal Polygonnetze zu verwenden, die mit einer Methode texturiert werden, welche auf externem Speicherzugriff basiert. Damit können mesoskopisch große Oberflächendetails der aufgenommenen Objekte darstellt werden, und die lokalen Polygonnetze können gemeinsam mit Punktwolken gezeigt werden. Eine andere Möglichkeit das Nutzungserlebnis zu verbessern besteht darin, Graphen im dreidimensionalen Raum zu verwenden, welche den Nutzer bei der Orientierung in Punktwolken von räumlich großen Örtlichkeiten unterstützen, in denen es ansonsten schwierig wäre die Sehenswürdigkeiten zu finden. Darüberhinaus gibt es die Möglichkeit die Qualität der dargestellten Punktmodelle zu verbessern, indem eine Punktgrößenheuristik

verwendet wird, die in Bereichen, die ansonsten mit einer zu geringen Abtastrate aufgenommen wurden, geschlossen Oberflächen nachbilden kann. Diese Oberflächennachbildung geschieht unter Berücksichtigung der unterschiedlichen Punktedichten in den verschiedenen Bereichen der dargestellten Punktmodelle.

Schließlich wird auch die Benutzung von Punktwolken als Werkzeug für die archäologische Arbeit vorgeschlagen. Seit archäologisch interessante Monumente immer häufiger mit Laserabtastgeräten dokumentiert werden, steigt auch die Anzahl der Anwendungsbereiche für die daraus resultierenden Punktwolken. Diese Anwendungsbereiche beinhalten, unter anderem, neue Ansichten der Monumente die nicht möglich wären wenn man sich in oder bei den Monumenten vor Ort befinden würde, weiters das Erstellen von Schnitten und Grundrißplänen, oder das Durchführen einer virtuellen Anastylose.

Alle oben erwähnten Methoden und Datenstrukturen sind in eine Softwareapplikation eingebaut worden, die während der Dissertation entwickelt wurde, und mit Hilfe derer es möglich ist, gigantische Punktwolken zu erkunden.

Contents

CHAPTER 1

Introduction

In computer graphics, images are synthesized from data that can be processed by a computer. A common usage of computer graphics is to display data that represents real-world objects for the purpose of exploring or editing it. Such data might be given by separate data items or by mathematical formulas, depending on how it was created or modeled. A very simple way to define a (real-world) object is to produce a set of unique point positions in space and so describe the surface of the object. The points have no interconnections between each other, so the space between the points is empty. The set of point positions is referred to as *point cloud* and the points have no given order or other distinctive features except for their position.

Mathematically, a point is a zero-dimensional entity, meaning it has no expansion or area, so displaying a point would therefore be impossible. When drawing points on paper, they are usually approximated by some geometric representative. Such representatives are placed at the positions where the individual points are located. Often dots are used as representatives, meaning a small area around the points is associated with them. In computer graphics, a similar metaphor is used, but instead of dots the points can be displayed as single pixels, which are the smallest distinguishable areas on a screen (in this thesis it is assumed that 2D raster displays are used for presenting computer graphics). An example for this can be seen in Figure 1.1, which shows a monitor on the left displaying colored points. The zoomed-in view on the right shows the separate pixels of the monitor, and how the points are displayed as pixels. So instead of dots, small rectangular (or square) regions are used as representatives for points. Note that the position of a point does not necessarily map onto the center of the pixel, but can be anywhere inside the area of the pixel, so there might be a slight displacement of the representative pixel relative to the point position.

While it is possible to model pure point-based objects, and there might be artistic or practical reasons to do so, points, as already mentioned, can also be used to describe the surface of objects. For displaying a surface, the empty area between the points has to be filled to reconstruct the geometry of the surface. There are different approaches to do this, for example displaying larger areas around the points (consisting of several pixels, not only single pixels) or by connecting the points and reconstructing a polygonal mesh for displaying the surface. Figure 1.2 shows the

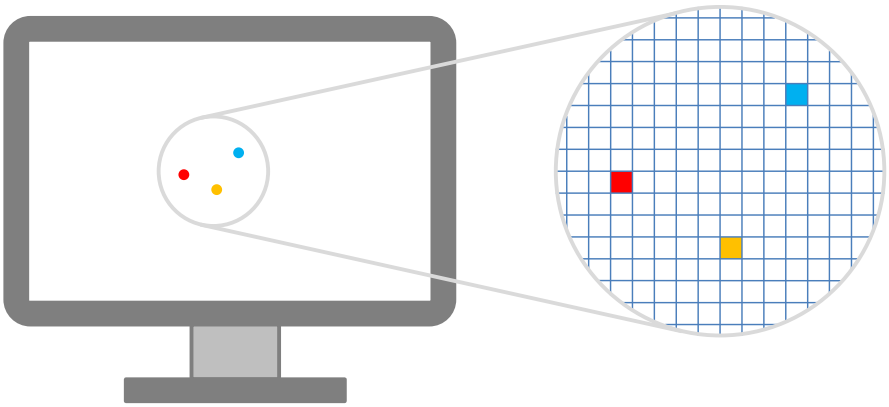


Figure 1.1: The monitor on the left displays 3 colored points. It uses a common raster display, which composes all images from single pixels, as can be seen in the zoomed-in view on the right. On the display, a point is mapped to a certain pixel if its position falls into the area of that pixel.

Armadillo model [99] displayed with points on the left, and displayed with a mesh on the right. The point model uses squares with 3 pixels side length to display the points, as can be seen in the zoomed-in view of the hand. The zoomed-in view shows that the size of the points is still too small for such close up views, and so the points of the back-facing surface can be seen as well. It is assumed that there is a light source in front of the Armadillo figure, therefore the points on the back surface receive no light (which is calculated from the orientation of the surface, not from the visibility of the light source), and hence they are displayed in black. From a distance, the mesh model looks like the point model, but the zoom in of the mesh model shows that the points are interconnected by edges (they are emphasized to make them clearly visible) which form the triangles of the mesh. The triangles define a closed surface, therefore all points and edges behind the front-facing surface are hidden.

Real-world objects like the Armadillo figure are transformed into point-based models by scanning their surface geometry with distance-measurement devices. A number of approaches exist for performing such measurements [43], like triangulation-based scanning or time-of-flight laser scanning.

Triangulation-based methods look at an object from two different positions, and calculate the point where the two viewing rays meet on the object. This is then a single point sample of the object's surface. Figure 1.3 shows the basic setup for triangulation-based scanning. There are two viewpoints which are separated by a fixed baseline. The viewing rays pointing towards the object intersect the gray image planes at single pixels. The whole setup is calibrated, which means the parameters of the image planes and the orientation and distance of the image planes to each other are known. In Figure 1.3 only the basic setup is shown, as there exist several variations for this method. One variation uses two digital cameras at the viewpoints, and correspondences in the captured images are used to determine the 3D positions of points in the captured images. Other variations replace one viewpoint with a light-emitting projector, where a mask can be placed in front of the light source, and so geometric patterns can be projected onto the object

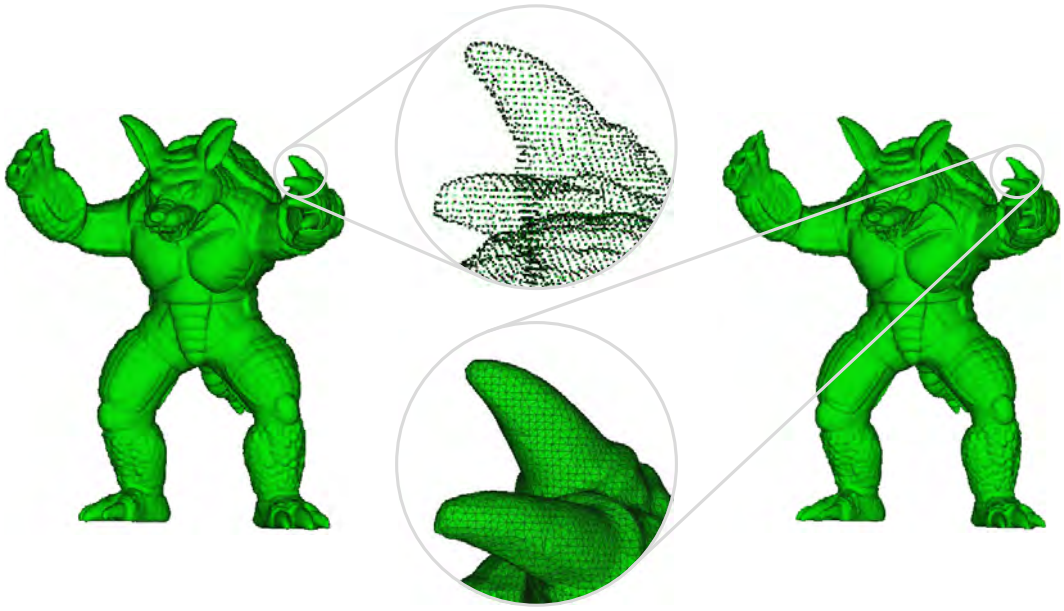


Figure 1.2: The model on the left is rendered with square points with 3 pixels side length. The zoom in on the hand was captured from a view position closer to the model, and it shows points from the front-facing surface and back-facing surface, as the points are not large enough to fill the space between neighboring points from this closer view position. The model on the right is rendered as a polygonal mesh. The zoom in on the hand shows single polygons (triangles) of the mesh, which are only shown in the zoom in to show the resolution of the mesh.

which shall be scanned. The camera at the other viewpoint captures these patterns (which are deformed due to the shape of the object), and from these images it is possible to derive the 3D surface of the object. Yet other variations use a laser light emitter instead of the second camera, and project a laser light stripe onto the object. Again, from the deformed laser light stripe on the object, points in 3D space can be computed.

Time-of-flight laser scanners do not use triangulation to measure distances. Instead, they send out single light pulses, which are reflected by the objects that are hit by the laser, and measure the time it takes for a light pulse to return to the scanner. The speed of light is a precisely known constant, so from the orientation of the laser beam and the measured round-trip time of the light pulse, a point in 3D space can be calculated. For scanning larger areas, the laser emitter can usually be rotated and tilted, so 360 degree panoramic surveys are possible. Also airborne laser scanning is possible, where a plane flying at low height carries a laser scanner and surveys the area beneath it. In Figure 1.4, the basic principle of time-of-flight laser scanning is shown by two situations, which appear in chronological order during a scan. In the upper half of the figure, the laser pulse has been emitted from the scanner and is heading towards the object. The laser pulse is shown like in a time/intensity diagram, where time is measured on the x-axis and intensity on the y-axis. In the lower half, the laser pulse has been reflected from the surface of the object, but its intensity has been decreased due to the material properties of the object's

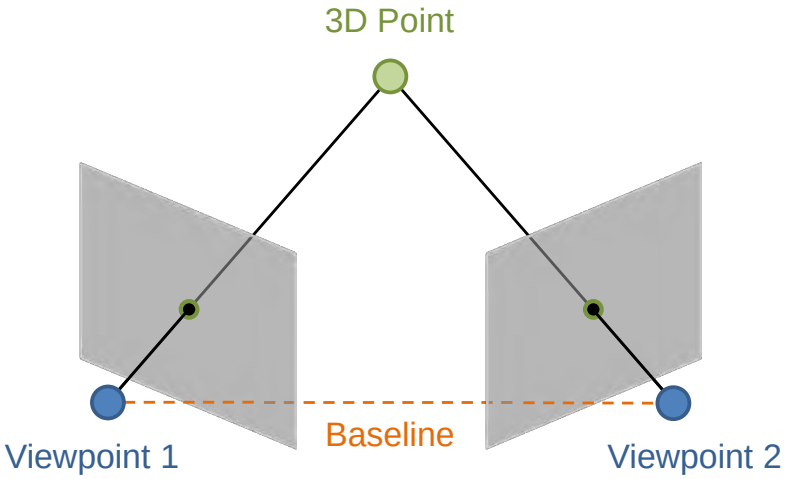


Figure 1.3: The basic setup for triangulation-based scanning. Some object is viewed from two different positions, which are separated by a known distance, the baseline. When intersecting the viewing rays from the two viewpoints, a point in 3D space can be reconstructed. The figure is redrawn as appearing in [43].

surface. The laser pulse is then captured by the laser scanner, and so the distance to the object can be calculated.

The above-mentioned techniques enable the digital scanning of real-world objects, and the resulting point-cloud data can then be displayed on a screen. The applications of such techniques are for example documenting historical buildings, sampling models for computer games, or surveying the changes of stock at industrial plants. These applications have in common that the current state of an object is recorded and archived, so it can be viewed later on or used as input for algorithms that simulate natural phenomena such as water flow, air flow, or light distribution.

For viewing the point-cloud data, it has to be rendered, i.e., it will be displayed with the help of computer graphics. Rendering the data can help to explore it, for example it can be inspected from different view positions, or made available for a larger public if it shows some site where access is limited due to time constraints or other restrictions.

The amount of data produced by scanning is partially dependent on the physical size of the scanned object, and partially on the sampling density (the number of samples taken per unit area) used during scanning. The sampling density can be chosen depending on the application. For example, when surveying large swaths of land from air, the sampling density will be chosen much smaller (i.e., the distance between single samples will be larger) as if sampling pieces of pottery. This is not only due to the absolute size of the scanned objects, but also due to the size of the features which shall be captured by the sampling process.

When scanning large sites or outdoor areas, point-cloud data sets with thousands of millions of points can be created, and such huge point data sets are not uncommon, as the latest generation of laser scanners can produce up to 10^9 points in a single scan [29, 90]. The capacities of digital storage devices have increased tremendously in the last decades, therefore allowing to store such huge point data sets. Handling or exploring those data sets, on the other hand, still poses

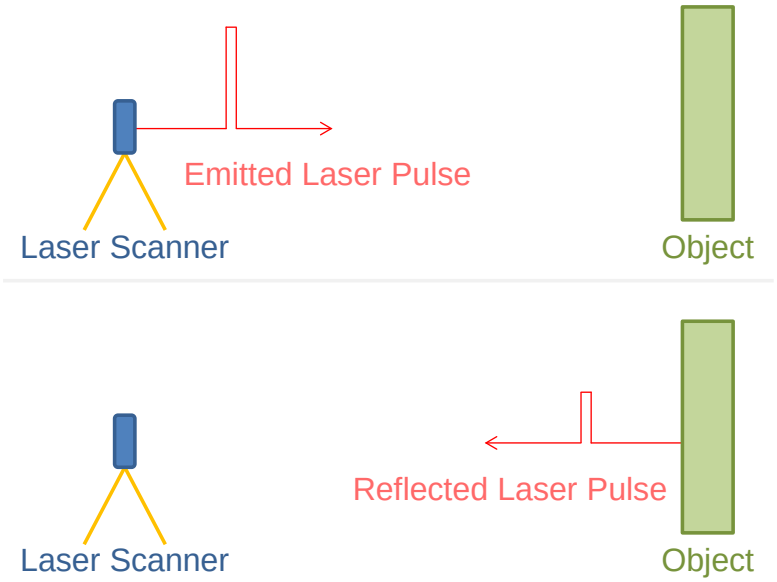


Figure 1.4: The basic setup for time-of-flight laser scanning. The upper half of the figure shows a laser pulse that is emitted from the laser scanner towards the object. The laser pulse is shown like in a time/intensity (on the x-/y-axis) diagram. The lower half of the figure shows the laser pulse that is reflected from the object. Due to the material properties of the surface of the object, some light is scattered into other directions or absorbed by the surface, so the reflected laser pulse has lost some intensity. The reflected laser pulse is then captured by the laser scanner, and from the round-trip time of the laser pulse the distance to the object can be calculated.

challenges. In this thesis, new methods for handling gigantic point clouds are proposed.

1.1 Motivation

Gigantic point clouds that cannot be held completely in the main memory of a computer system can be handled in different ways.

One way to handle them is to resort to external-memory (also called out-of-core) algorithms. Such algorithms hold only parts of the complete data set in main memory (i.e., in-core) at any one time, while loading new data to main memory when needed, and thus allow for viewing, editing, and exploring those data sets interactively. This approach requires to insert points into appropriate data structures, and load them from external memory when needed. While accessing external memory takes time, it has the advantage that the complete data set is available if necessary.

Another strategy would be to split up a gigantic point cloud into smaller point clouds, which can then be processed individually. This is a sensible strategy if only parts of the complete point cloud are required, for example when exploring limited areas within a larger complex. Yet another possibility is to reduce the size of a gigantic point cloud by subsampling the points (i.e., selecting a subset of the points), or resampling them (i.e., creating new samples from the existing

ones). While these methods can be used to give an overview of the complete point cloud, they are not suitable for tasks like walk-through applications or creating cuts and floor plans, due to the reduced sampling density.

In addition to the previously mentioned methods, it is also possible to compress the attributes of the points, e.g., position or color. Most efficient are lossy compression schemes, where some information is sacrificed in order to increase the compression ratio. While this is a viable option for visualizing point clouds, it is usually not efficient if point clouds shall be edited as well, since maintaining the compression scheme during editing operations can become a complex and computationally intensive task. Often points are compressed in groups, so points can become dependent on their neighbors, so when changing the attributes of a single point, the attributes of the neighbors have to be changed as well.

The decision which method shall be used is based on the task that shall be accomplished. The most versatile approach seems to be the out-of-core algorithm, since with this approach it is not necessary to change the original data, and when using an appropriate data structure it is possible to edit the complete data set if needed. Therefore, this is the approach that will be used by the algorithms and data structures developed for this thesis.

Despite the problems of handling large point clouds, the number of application areas that can benefit from such point clouds has increased considerably in the last years. Not only different industries and areas of research use laser scanning for an increasing number of tasks, like oil companies for facility management of their refineries and oil platforms, or mining companies for surveying their mines, or civil engineers for land surveying, or archaeologists for the documentation of excavation sites. Also the number of specialized scanning devices has risen in the last years. Today, not only terrestrial scanning devices are available (scanners that have to be fixed at each scanning position), but also devices for airborne laser scanning, mobile laser scanning, and scanning from unmanned vehicles (like drones). Together, these factors cause a lot of interest in systems that can handle gigantic point clouds.

Another source of huge data sets are the photos which are used to color the points sampled by a laser scanner. Taking photos from the objects' surface captures the surface's appearance, which is not recorded by laser scanning, as the laser measures only the distance to the objects. There are laser scanners where a camera is mounted on top of the device, so the photos can be captured from the position of the laser scanner and these photos can then be mapped to the recorded points. This way, a colored point cloud can be produced, where each point has a single color assigned to it. Such a colored point cloud gives already a good overall impression of the captured object, but the sampling density of the camera (usually a digital camera is used which captures the photos and stores them as 2D pixel arrays) is often higher than the sampling density of the laser scanner in the same area. One possible method to make use of this higher sampling density is to reconstruct a polygonal mesh from the point samples, and map the photographs to the mesh (this process of mapping images to a mesh, or generally to a 3D model, to add surface detail, is called texture mapping [18, 47]). Since there might be slight errors when mapping the photos to the mesh that is reconstructed from the points, it is also possible to use passive stereo scanning (see Figure 1.3), where two cameras are used in the triangulation scanning setup to sample the surface of objects [43]. From the photos taken during passive stereo scanning not only a polygonal mesh of the surface geometry can be derived, but the same photos can also be

used for texturing the resulting mesh, and the mapping of the photos is often less error prone than in the previously mentioned method. The so created textured mesh models show the surfaces in a resolution that equals the resolution in the captured photos, which gives high-quality 3D mesh models, but the memory requirements for the photos can become very large, so rendering methods that can deal with such mesh models are necessary.

1.2 Problem Statement

1.2.1 Handling of Gigantic Point Clouds

The main challenge that shall be tackled in this thesis is the *handling, interaction, presentation, and manipulation of gigantic point clouds*. The term gigantic in this case is used to describe point clouds that are too large to fit completely into the main memory of an off-the-shelf computer system. Such point clouds are the result of extensive scanning campaigns, which are carried out to document the current state of vast cultural heritage sites, archaeological excavation sites, underground galleries, or the like, and which will serve as a reference for future research, also in the case of severe changes to the recorded facilities. Despite the unwieldy size of those data sets, it would be useful if they could be processed on off-the-shelf computer systems.

1.2.2 Simultaneous Display of Textured Meshes and Gigantic Point Clouds

An additional challenge is to *display polygonal meshes textured with high-resolution images at the same time as point-based models*, so as to provide a simultaneous visualization of multi-modal captured data sets. Since the textures used for the polygonal meshes show the original surface of the captured objects, they are non-repetitive, furthermore they are composed of high-resolution images, so these textures require a lot of storage. These aforementioned factors cause the textures to often become too large to fit completely into the graphics memory of an off-the-shelf computer system, making it difficult to show such textured polygonal meshes on such computer systems.

1.2.3 Navigating Huge Data Sets

When visualizing huge data sets from large sites, the orientation within the data set can become a problem. This is especially true when moving through the model of a city, a building, a catacomb, or the like, where the dimensions of the model alone make orientation difficult. While there may be unique features in the model itself to help orientation, this is not always the case, so some additional way of guidance is often required, to provide *help when navigating huge data sets*.

1.3 Contribution

This thesis contributes to the field of handling gigantic point clouds, which involves handling those point clouds on computer systems that do not have enough main memory or graphics card

memory to store the complete data set. The contributions were introduced in several papers, which were presented in journals and at various conferences, and are listed in the following.

1.3.1 Point Cloud Visualization and Editing

In this thesis a hierarchical data structure, called *modifiable nested octree* (MNO), is proposed, which can be used to store, render, and edit gigantic point clouds with out-of-core algorithms. This data structure is based on an octree and stores points at all levels of the hierarchy. It also provides an LOD mechanism for rendering that does not use additionally created points. Another contribution of this thesis is a *point size heuristic* that can be used when rendering points stored in an MNO. It tries to adjust the screen sizes of the points such that a closed surface is mimicked for the rendered point cloud. It does so by increasing the point sizes in areas where the density of the rendered points would be otherwise too low for neighboring points to overlap. The next contribution is a data structure, called *selection octree*, which allows defining regions of selected points for out-of-core managed point clouds. It is based on an octree, and describes the volumes of the selection regions. This way, it is not necessary to store the information about the selection at the points themselves. Another contribution is a *complexity analysis* of several operations conducted on an MNO, like inserting, searching, or deleting points. The complexities are then compared to the complexities of these operations on existing data structures. These contributions were presented in the following papers:

- C. Scheiblauer and M. Wimmer, *Out-of-Core Selection and Editing of Huge Point Clouds*, Computers & Graphics, 35(2), pages 342-351, April 2011
- C. Scheiblauer and M. Wimmer, *Analysis of Interactive Editing Operations for Out-of-Core Point-Cloud Hierarchies*, WSCG 2013 Full Paper Proceedings, pages 123-132, June 2013

1.3.2 Case Studies for the Use of Gigantic Point Clouds in Archaeology

Sampling real-world objects and converting them to virtual models opens up many use cases that are not possible without this digitization. In this thesis, case studies are shown where point models from archaeological monuments support the work of archaeologists. The contribution of this thesis is the demonstration of *tasks that can be carried out directly on point models*, without converting them to higher quality mesh models, making direct point visualization and interaction a very relevant topic for archaeological work. These tasks include exploring the large-scale structure of the model, classification of findings, taking measurements, virtual anastylosis (i.e., creating hypotheses about the original structure of the monument), creating sliced views, visualizations limited to different building phases, and acquiring an understanding of the 3D structure of the monument in general. The case studies were presented in the following papers:

- C. Scheiblauer, N. Zimmermann, and M. Wimmer, *Interactive Domitilla Catacomb Exploration*, Proceedings of the 10th VAST International Symposium on Virtual Reality, Archaeology and Cultural Heritage, pages 65-72, September 2009

- C. Scheiblauer and M. Pregesbauer, *Consolidated Visualization of Enormous 3D Scan Point Clouds with Scanopy*, Proceedings of the 16th International Conference on Cultural Heritage and New Technologies, pages 242-247, November 2011

1.3.3 Case Study for the Use of Virtual Texturing in Archaeology

Virtual texturing is a technique which helps to map very large and detailed textures onto polygon-based models by managing the texture data out-of-core. In this thesis a work pipeline for the creation of virtually textured meshes is proposed and a system for the *combined display of gigantic point clouds and virtually textured meshes*. The combination of those model types helps to emphasize landmarks in a virtual scene, for example when showing a large monument. The landmarks can be modeled as textured meshes, while the rest of the monument can be modeled as a point cloud, as in most parts of the monument the surface details are not as important. Nevertheless, the point cloud provides a context for the landmarks and gives an impression of the size and extent of the whole monument. A case study, where this pipeline and the virtual texturing system were used, was presented in the following paper:

- I. Mayer, C. Scheiblauer, and A.J. Mayer, *Virtual Texturing in the Documentation of Cultural Heritage*, XXIIIrd International CIPA Symposium, September 2011

Additionally, an enhancement to the tile determination step of the virtual texturing algorithm is proposed. This enhancement speeds up the redundancy elimination in the list that contains all tiles visible from the current viewpoint.

1.3.4 Navigation

In this thesis, a *graph-based guidance system* is proposed for virtual models that are so large that a user can lose orientation when exploring them. This navigational help restricts the possible movement directions for the user to the edges of a graph representing the most interesting routes through the virtual model. The user can then follow these routes to find the landmarks inside the virtual model easily.

- C. Scheiblauer and M. Wimmer, *Graph-based Guidance in Huge Point Clouds*, Proceedings of the 17th International Conference on Cultural Heritage and New Technologies, November 2012

All of the previously mentioned contributions are implemented in a software application that was developed in the course of this thesis. It is used to test and benchmark the new algorithms and data structures, and it is also used to compare their time and space complexities with already existing ones. Furthermore, it serves as a testbed for the developed interaction techniques and was used in a user study to evaluate these techniques in an archaeological work flow.

1.4 Structure of the Work

In Chapter 2, the work related to this thesis is listed. In Chapter 3, the MNO data structure, the point size heuristic, the selection octree, a complexity analysis of the MNO, and a comparison to existing data structures is presented. In Chapter 4, case studies are presented that show the use of gigantic point clouds in the field of cultural heritage and archaeology. In Chapter 5, a case study showing the use of virtual texturing for presenting large archaeological monuments is described. In Chapter 6, a graph-based guidance system for navigating through point clouds of great extent is shown. In Chapter 7, the software application that was developed for this thesis is described. Finally, in Chapter 8 the conclusion for the thesis is given.

CHAPTER 2

Related Work

Points as rendering primitives for models with closed surfaces were first considered by Levoy and Whitted [65] (rendering primitives are the basic geometric entities to render 3D models). One reason they chose points for rendering were the increasing geometric complexity of 3D models at that time, which caused other primitives like lines, polygons, or parametric surface patches to become ever smaller when approximating a surface. Ultimately, during rendering, all types of geometric primitives will be converted into discrete pixels on a raster display (see also Figure 1.1), therefore a rendering method that uses point models and maps the points directly to pixels can potentially be more efficient than a method that has to convert complex primitives to single pixels. Figure 2.1 shows a basic rendering pipeline, where 3D models, consisting of

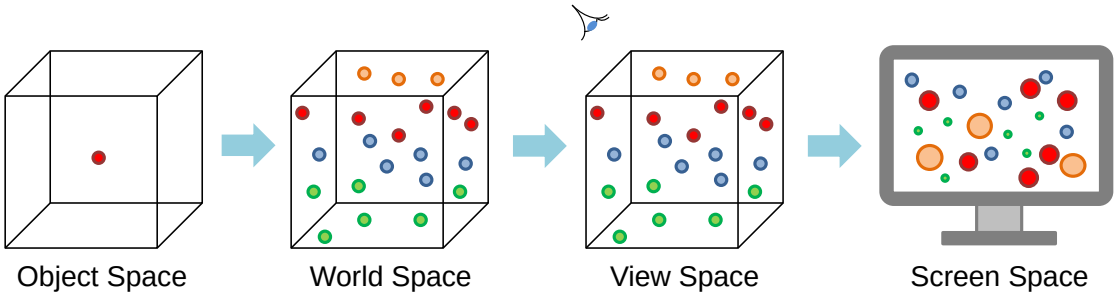


Figure 2.1: From left to right the most important transformations of a basic 3D rendering pipeline are shown. Initially, the geometry of some model exists in its own coordinate system, the so-called object space. Then it is moved into world space, where several models together are forming a scene. In the next step, an observer is introduced, who views the scene from a certain position. Subsequently, all models are moved into view space, where the position of the observer is at the origin of this space. Then follows a projection from 3D space to 2D space, where the models can finally be displayed on a screen. In the shown example, a perspective projection is used, so objects appear scaled relative to the distance of the observer.

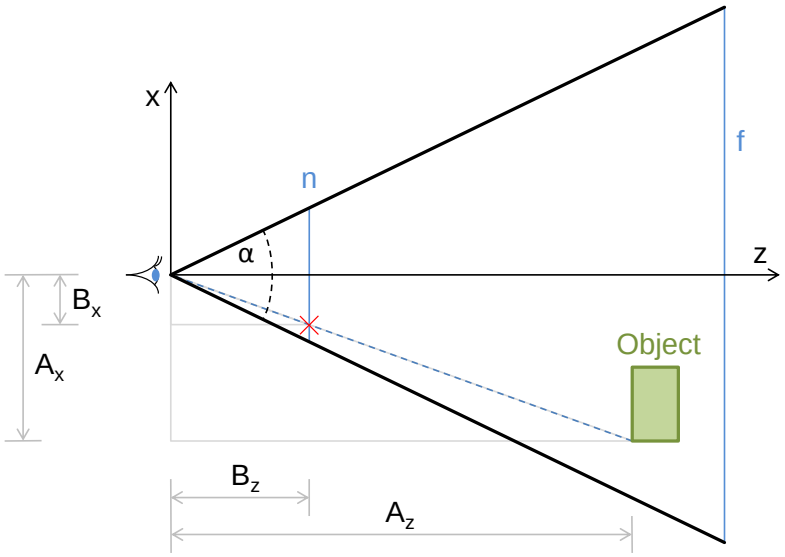


Figure 2.2: The principle of a perspective projection shown in 2D. From the viewpoint, which is the position of the eye at the coordinate system origin, a view frustum is established, with field-of-view α, near plane n, and far plane f. The line-of-sight from the viewpoint to an object intersects the near plane at some position (marked with red X). From the relations of the distances B_x/A_x and B_z/A_z the position of the object's coordinate on the near plane can be drived.

geometric primitives, are transformed from their initial 3D coordinate system (in this thesis it is assumed that models reside in 3D Euclidean space) to a 2D coordinate system on a discrete pixel display. The 3D models are originally in their own coordinate system, the so-called object space. In the next step of the rendering pipeline, they are moved to world space by a coordinate system transformation, where different models (or several instances of the same model) are assembled into a combined scene. Up to this point, it is not clear from where the scene will be looked at, so the position of an observer (also called viewpoint) is introduced. The scene is then transformed to the so-called view space, where the position of the observer is at the coordinate system origin. From this coordinate system, a projection from 3D space to 2D space is performed, so that the 3D models can be displayed on a 2D screen. Finally, the elements, which reside at continuous 2D coordinates, are discretized, so they can be mapped to a raster display. In this thesis, by default a perspective projection is used from 3D space to 2D space, so rendered models seemingly change their size depending on the distance to the viewpoint and appear smaller the further away they are from the viewpoint.

Figure 2.2 shows the principle of a perspective projection in 2D. The viewpoint is at the position of the eye at the coordinate system origin. From there, a view frustum is defined in the xz-plane with the center line along the z-axis, with field-of-view α, near plane n, and far plane f. The object is projected onto the near plane by following the line-of-sight from the viewpoint to the object, and intersecting it with the near plane. From the relations of the distances B_x/A_x and B_z/A_z, derived from the relations of similar triangles, the position of the object's coordinate

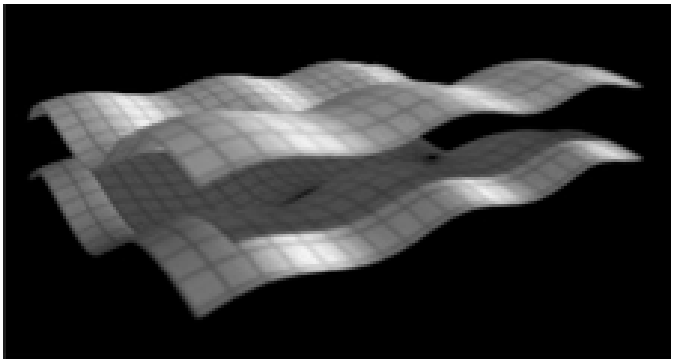


Figure 2.3: Two flat polygons converted to point sets and rendered with the method presented in [65]. The point sets are textured with the image of a grid, and their originally flat surface is offset per point with a discrete height field. Nevertheless, the rendered surfaces appear smooth. Image from Levoy and Whitted [65].

on the near plane, B_x, can be calculated. Since A_z, the object's distance to the viewpoint, A_x, the object's coordinate an the x-axis, and B_z, the distance of the near plane to the viewpoint, are known, it follows that $B_x = (A_x B_z)/A_z$.

An important technique to speed up rendering is view frustum culling. It is based on the observation that only objects inside or intersecting the view frustum (i.e., objects that are partly or completely in the area bounded by n, f, and the connecting outer edges) are visible to the observer. Therefore, all objects completely outside the view frustum do not have to be processed by the rendering pipeline (if no rendering techniques are used that require the interaction of visible objects with objects outside of the view frustum).

Levoy and Whitted showed that models like spheres and planes can be converted into a point-based model before entering the rendering pipeline, and can nevertheless be rendered with a closed surface. Another reason for them to chose points as rendering primitives was the unification of the rendering pipeline. Because they converted all models to point-based models, regardless by which geometric primitives they were originally built, they could render all of them with the same rendering pipeline. They had to make some assumptions though, for example, the distance between the points in the point-based models had to be small enough, such that neighboring points were projected to neighboring pixels (or to the same pixel), so that the rendered surfaces showed no gaps. Pixels where no points were projected to remained empty. Figure 2.3 shows an example, where two rectangular flat polygons have been converted to a point-based representation (a 2D array with 170x170 points). During rendering, the points are shaded, textured with an image of a grid array, and the height of the points is offset by a discrete height field. The offset is chosen such, that the rendered surfaces still appear smooth (there might be two gaps in the lower surface, but since the image quality is not very good, it is unclear if this is due to the original rendering of the surfaces or due to some later introduced artifacts in the image).

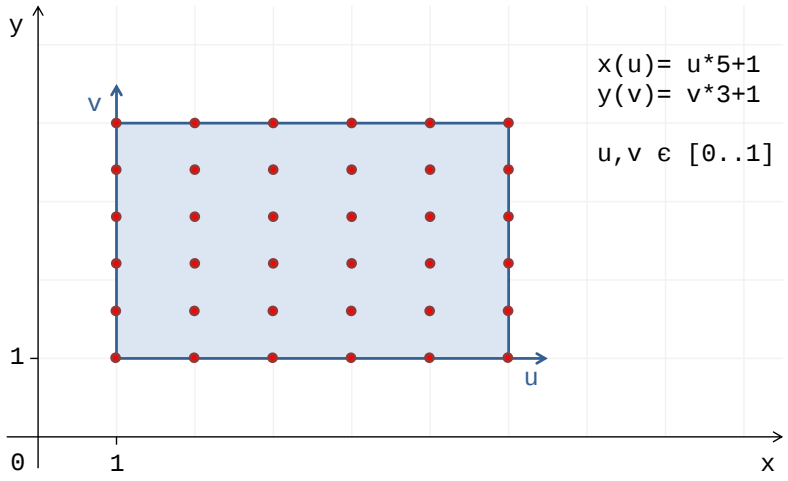


Figure 2.4: Point samples taken from a parametrically defined rectangle. The red point samples are taken at a constant step size of 0.2 of the two parameters u and v, resulting in a higher sampling density along the y-axis than along the x-axis.

2.1 Creating Point-Based Models

In the following, a short overview is given of how point-based models can be created. While this thesis is not concerned with the process of creating point-based models itself, some of the problems that occur when rendering point clouds are a direct result of the creation process. The creation process does not (or cannot) always consider how point-based models will be rendered afterwards, so the point samples might be acquired in an unfavorable order or density.

The creation of point-based models can be divided into two basic methods. The first method is to sample geometric objects already present in another form in a computer, for example as a data structure, mathematical formula, or other geometric primitives. The second method is to sample real-world objects with the help of recording devices, whose output data can be interpreted as point-based models.

2.1.1 Sampling Geometrically Defined Objects

As already mentioned by Levoy and Whitted [65], almost any kind of geometric representation can be converted into a point-based model (there are problems with fractals though). This is most easily done with parametrized geometry, where coordinates on the surface geometry are defined by continuously changing parameters. Point samples can then be taken at parameter values in constant intervals, or by randomly chosen parameter values. Figure 2.4 shows an example of a parametrically defined geometry, namely a rectangle. Any point in the x, y plane that is a solution to the two equations $x(u) = 5u + 1$ and $y(v) = 3v + 1$ with $u, v \in [0..1]$ is part of the rectangle. The point samples are taken at constant intervals with a step size of 0.2, resulting in a 6-by-6 grid of points. The sampling density is higher along the y-axis, so if the rectangle shall be displayed with a solid surface, this has to be compensated for during rendering.

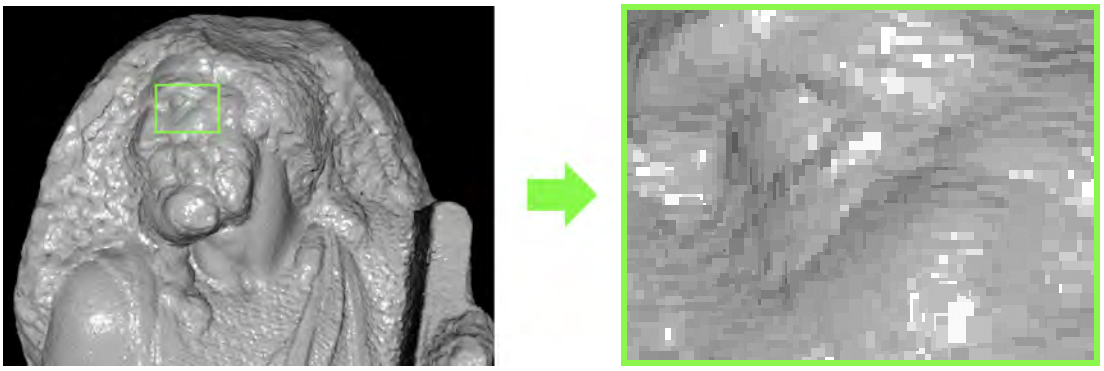


Figure 2.5: A point-based model sampled from a mesh and rendered with square splats per point. It shows Michelangelo's statue of St. Matthew [64]. The splats have a side length of 10 pixels. The zoom in shows a magnification of the rendered image. Images from Rusinkiewicz and Levoy [91].

Rusinkiewicz and Levoy [91] use meshes as sources for point clouds. They sample them by using the positions of mesh vertices as sample points. Since they cannot guarantee that during rendering the density of the points is high enough, so that at least one point is projected into every pixel covering a surface, they are estimating for each point a radius. The points are then rendered as splats, i.e., small geometric shapes that cover a small continuous area consisting of pixels, on a raster screen. The sizes of the splats are chosen such that they are guaranteed to overlap, so that during rendering no gaps appear between neighboring points. Splats can have different shapes, for example circles or squares. Figure 2.5 shows a point-based model sampled from a mesh and rendered with a square splat per point. The splats have a side length of 10 pixels.

This kind of sampling is also referred to object space sampling, as the point samples are generated with respect to the definition of the object without considering the effects of the sampling strategy on the rendered point model. Such sampling can nevertheless be used for rendering point-based models, if it is accounted for by appropriate rendering algorithms. This can be done for example by applying some kind of filtering to the rendered points, to hide the irregular sampling density of the point-model, or by adjusting the size of the splats that are used for rendering.

Image space (or screen space) driven sampling, on the other hand, refers to a sampling strategy that already considers the use of the point samples for rendering. The sampling positions are chosen on a regular 2D grid that resembles the pixel grid on a raster display. The grid is positioned in 3D space in vicinity of the geometry that shall be sampled, and rays normal to the grid plane (originating at the grid vertices) are cast towards the geometry. When recording only the first hit of the rays with the geometry [44], by storing the distance to the grid plane, the resulting sample values can be interpreted as pixel values. The obtained image is then a depth image as seen from the grid plane. This approach is problematic, as several sampling positions have to be used to cover the surface of the whole geometry, and for geometries with convex surfaces (as for example in Figure 2.6) several supplemental sampling positions are required to avoid holes in the sampled surface due to occlusions. A different approach used by Pfister et

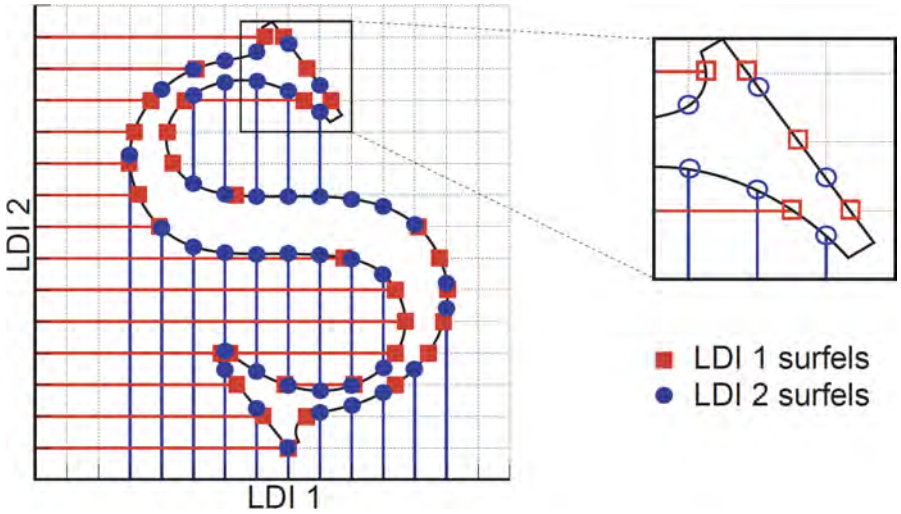


Figure 2.6: Layered depth cube sampling in 2D. Two layered depth images (LDI1 and LDI2) from orthogonal directions are created by shooting rays from the image planes through the geometry, and recording the positions of (and other surface properties at) the intersections. Figure from Pfister et al. [84].

al. [84] records not only the first hit of the rays with the geometry, but all intersections of the rays along their paths through the geometry. At the intersections, the position, color, normal, and other properties can be recorded. The position is stored as distance of the intersection to the grid plane, so several intersections along the same ray can be recorded with increasing distance (or depth) from the grid plane. The data structure that stores the recorded information is called a layered depth image (LDI) [97]. Three mutual orthogonally captured LDIs in 3D space are called a layered depth cube (LDC) [66]. Figure 2.6 shows a 2D example of an LDC [84]. The sampling positions on the geometry itself appear irregular, but the points are rendered as splats, so when choosing a sampling density which guarantees that every pixel in the rendered image is covered by at least one splat, this allows for rendering point-based models without holes in the resulting image. Therefore, for this method to work, the expected resolution of the rendered image has already to be considered during sampling.

2.1.2 Sampling Real-World Objects

Sampling real-world objects requires a kind of scanning device that can record the environment in a way, that distance measurements can be accomplished. In this thesis, only surface scanning devices are considered, i.e., no devices that could penetrate the surface of an object. When using a surface scanning device, sampling the complete surface of an object can only be done by conducting a series of scans from different scan positions relative to the object.

There are different approaches how an object can be scanned from several viewpoints (note that the measurements taken with a triangulation scanner from a single position count as one viewpoint, because the two viewpoints on the baseline are necessary to calculate the object's

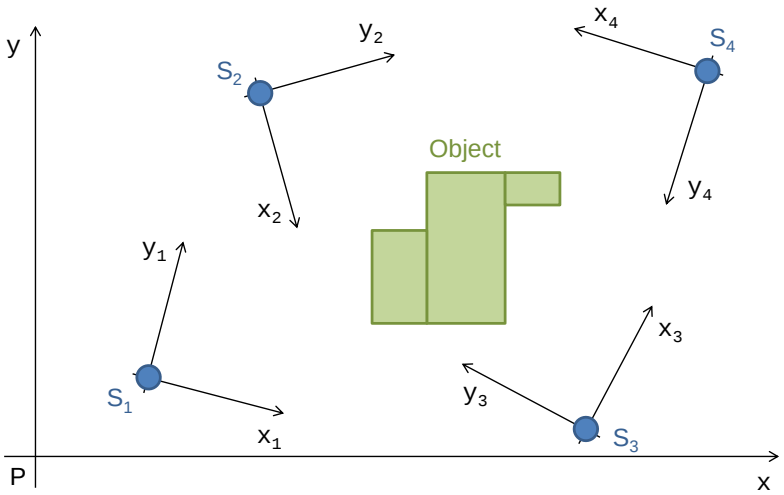


Figure 2.7: A top-down view of scanning an object in an outdoor setting. The four scan positions S_1 to S_4 are used to sample the surface of the object. Usually, a single scanning device is used, so the scan positions have to be completed sequentially.

surface points for this single scanner position). For a small object, it is possible to place it on a platform and rotate the platform in discrete steps. At every step, a scan is taken from the object at the current rotation angle. The point clouds resulting from the different scans are then transformed into a common coordinate system and can then be merged into a single point cloud, which then shows the object with an almost complete surface, except for the top and bottom. When scanning bigger objects or outdoor areas, the scanning device itself has to be moved to capture the surface of the object from different directions.

A scanner setup for outdoor scanning as shown in Figure 2.7 poses several challenges. Each scan has its own coordinate system S_i with the axes x_i, y_i, where the scanner is at the origin. Since multiple scan positions are used, the point samples have to be moved into a common coordinate system, P, the so-called project coordinate system. Finding the best transformation for each scan into the common coordinate system (such that the same areas of the object sampled from different scan positions are aligned in the common coordinate system) is called registration. If the positions and orientations of the scanners are unknown, features within the recorded data have to be searched for correspondences, so that the scans can be aligned according to those features. The kind of features that are searched for depend on the scanning method and will be described at the particular scanning methods below. The scans are also overlapping, which can lead to unevenly sampled areas, if point samples of all contributing scans are used at the same time. These areas might cause problems when rendered, due to their uneven sampling density.

Before the actual scanning starts, usually a plan is prepared to determine the scan positions, so that objects will be sampled completely. This way, large objects like statues [64], underground catacombs [94, 114], or amphitheaters [93] can be captured efficiently during a scanning campaign. The number of scan positions used in a scanning campaign is not limited, as each scan produces a separate point cloud, therefore scan campaigns can consist of several 1000 single scan positions. From the point clouds of these scans a single point cloud can be produced,

which might consist of thousands of millions of points. While there is not always the need for accessing a complete data set of such size, some applications benefit a lot from this possibility, for example exploring the point cloud or presenting a virtual walk-through of the scanned objects. For such applications, specialized algorithms for interacting with the data have to be created, so it will be possible to deal with such huge point clouds.

In the following, a few selected scanning methods are described. The described methods were also employed to create the models that are used later on in the thesis for benchmarking. A more complete overview of scanning methods can be found for example in [43].

2.1.2.1 Passive Stereo Scanning

Passive stereo scanning is a type of triangulation-based scanning, where at the two viewpoints (see Figures 1.3 and 2.8) cameras are positioned. The cameras take each a photo, and from these two photos the distances from the cameras to the sampled (i.e., photographed) objects can be determined. The distance measurements are based on detecting correspondences (e.g., color or brightness) in both photos, which belong to the same 3D point in the two photos. Assuming the photos are available as 2D pixel arrays, then for each pixel of the left viewpoint image a corresponding pixel on the right viewpoint image should be found. Since photos usually exhibit a lot of similarity at pixel level, e.g., due to repeating texture or evenly colored areas, it is more robust to compare small areas of the images instead of single pixels, so the characteristics of the compared areas become more unique. This can be done with a window (i.e., a pixel area) around the initial pixel, for example 3-by-3 pixels in size. A naïve approach to find the correspondences would be to compare the window of the left image with all possible window positions in the right image. A more efficient approach is to use epipolar geometry, which allows to reduce the search area for the window centers to a line across the right image, the so-called epipolar line. The pixels in the right image corresponding to a pixel in the left image can only be on this line.

Figure 2.8 shows the setup for using epipolar geometry. The positions and orientations of the two cameras (the viewpoints are the centers of projection of the cameras) relative to each other have to be known, as well as the intrinsic parameters of the cameras (e.g., focal length, image format, and the principal point, which is the center of projection on the image plane). The image planes record the 2D projections of the 3D objects. The projection of the 3D point X to the left image plane is denoted as X_{Left}. When introducing a plane connecting the two viewpoints and X_{Left}, the so-called epipolar plane, then X also lies on this plane. The epipolar plane cuts the right image plane along the epipolar line, which also means that X_{Right}, the projection of X to the right image plane, has to lie somewhere on the epipolar line. This way, the number of possible window positions can be reduced by a factor of approximately $\sqrt{n}$, with n being the number of pixels in an image. The viewpoint of the left camera is projected to the epipolar point on the right image plane (and vice versa). It is a special point, as all epipolar planes (and therefore all epipolar lines) are passing through this point.

The quality of the 3D positions calculated by passive stereo scanning is dependent on various parameters, including the quality of the cameras, the lines-of-sight for the cameras, and the material of the captured objects. Materials with glossy surfaces make the appearance of objects in the images view dependent, therefore the feature detection might become imprecise. Very bumpy surfaces, which exhibit deep cuts, can pose a problem, as always two free lines-of-sight

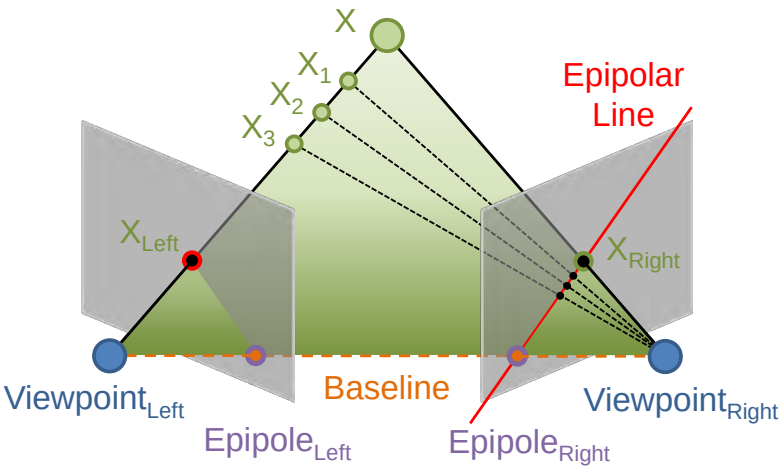


Figure 2.8: Epipolar geometry can be used to find the distance of a point X in 3D space from two cameras, whose position and orientation relative to each other is known. The left and right viewpoint are the projection centers of the cameras, and the grey planes are the image planes of the cameras. The point X, which is projected to the pixel X_{Left} in the left image plane, can only lie on the epipolar line in the right image, which is a projection of the viewing ray from the left viewpoint to X in the right image. Image inspired by Gross and Pfister [43].

are required for the sample measurements. A higher resolution of the taken images is usually advantageous, as characteristics of the captured objects can be sampled at a higher density, so they can be better distinguished.

A color-based (or brightness-based) detection of corresponding pixels might lead to a dense map of distance measurements, but the quality of the measurements is highly dependent on the captured objects. Difficult scenes with repeating texture (e.g., foliage) or almost monotone colored scenes (e.g., house walls) pose problems, because it might not be possible to find unique correspondences. An alternative approach is to look for features, like edges, in the two images, which have a unique correspondence. This might not produce a dense map of distance measurements, but is more robust to false correspondences [43].

While triangulation based scanning methods have advantages, like cheap implementation or increased precision with increased camera resolution, they also have disadvantages, like the requirement for two free lines-of-sight, relative short range of operation, and the handling of glossy materials. The models presented in Chapter 5 were captured indoors with passive stereo scanning and a scanning range of up to 5 meters [73], which results in a sub-millimeter precision of the measured distances. A software was used for the evaluation of the photos [3], which calculates the distances based on the passive stereo scanning method. From the resulting point clouds meshes were created, and they were textured with the photos used for the distance measurements. The rendering technique used for displaying mesh models textured with photos is also described in Chapter 5.

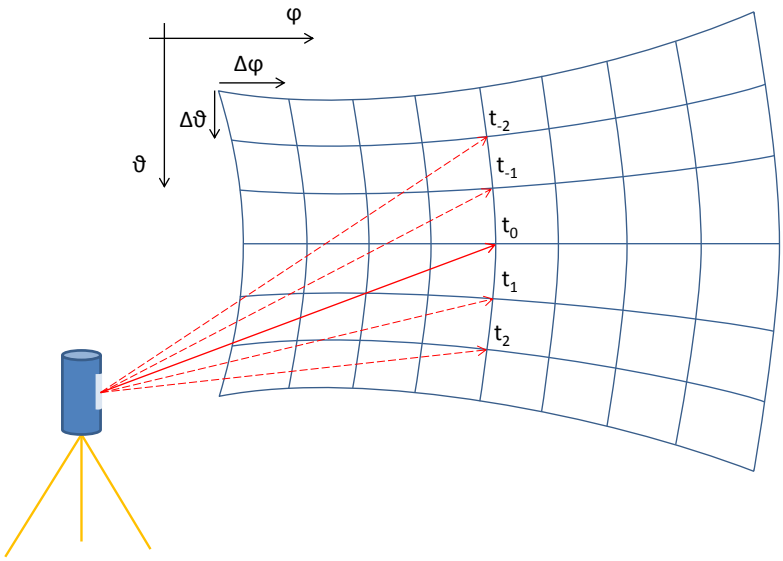


Figure 2.9: A time-of-flight laser scanner (left) measuring distances at different angles. The angles are changing at constant step sizes $\Delta\varphi$ and $\Delta\vartheta$. The samples are taken one at a time (the instants of time $t_{-2}..t_2$ show the order in which the samples are taken). The sampling pattern resembles a grid, which allows to interpret the resulting data either as 3D point samples or as 2D depth image.

2.1.2.2 Time-of-Flight Laser Scanning

As already mentioned above, time-of-flight laser scanners sample objects by measuring the round-trip time of a laser light pulse, which is sent out and captured by the scanner (see also Figure 1.4). This type of laser scanning is also termed lidar scanning (lidar is a blend of "light" and "radar", and short for "light detection and ranging") [81]. In Figure 2.9, the operation of a terrestrial time-of-flight laser scanner is shown. The scanner, depicted on the left side, measures the distances sequentially sample by sample. The laser beam proceeds at discrete steps, changing the line-of-sight about angle $\Delta\varphi$ in horizontal and angle $\Delta\vartheta$ in vertical direction respectively. Due to these constant step sizes, the samples can be interpreted as a depth image (or 2D grid), with the distances of the point samples as the pixel values in the depth image. When only looking at the 3D positions, the samples can be interpreted as a point cloud. While the angular change between neighboring samples is constant, the gap between neighboring samples in 3D space is actually dependent on the distance of the samples to the scanner. Assuming two neighboring samples at the same distance z from the scanner, than the gap between them is $2 \cdot z \cdot sin(\Delta\varphi/2)$ in horizontal and $2 \cdot z \cdot sin(\Delta\vartheta/2)$ in vertical direction respectively. Therefore, a point cloud sampled by a time-of-flight laser scanner is much denser sampled in the vicinity of the scanner than further away.

The precision of the distance measurements in the line-of-sight direction is approximately ± 10 millimeters, while the precision for the angular step size decreases about 1 millimeter per 100 meters [90] (it is approximately 1 millimeter at 100 meters distance).

Often a camera is mounted on top of the scanning device to take photos from the sampled objects. This happens in a second pass, after the distances have been measured. The photos can then be mapped onto the captured point samples, which results in colored point clouds, and the point clouds used in this thesis are colored with this method.

Airborne and mobile laser scanning are working with a different device setup, because in these cases the scanner itself is constantly changing its position, so knowing the precise position of the scanner itself is very important, but the basic principle of time-of-flight laser scanning is the same. In this thesis, only data sets from terrestrial laser scans are used, although this is not a requirement for the later on presented rendering and editing techniques for point clouds.

2.2 Handling Gigantic Point-Based Models

There exists a plethora of data structures that can be used to store and process point data sets. A structured overview of the most well known variants is given by Samet [92]. The decision, which data structure shall be used, is dependent on the specific use case and task that shall be accomplished. Each data structure is designed for a set of use cases, and one data structure is usually not the best choice for all tasks. Another distinction between the different variants of the data structures are their implementation complexities.

For this thesis, the requirements for the data structure holding the point data are mainly given by the need for a fast random access to the data, even when the size of the data exceeds the size of the available memory of a computer system. Furthermore, fast insertion and deleting operations are required, to enable efficient editing operations. Because of this, data structures that use complex compression schemes (for example [61] or [95]) or statistical representations of point clouds [53] are not eligible, because the data is usually stored in a way that does not allow to insert, replace, or delete points easily.

In the following, several data structures for handling point data are presented. Although the point data sets used in this thesis exist in 3D space, the data structures are described first in 2D space, for the sake of convenience. The extension of these data structures to 3D space is usually straightforward.

2.2.1 Quadtrees

A simple data structure for ordering and fast searching of points is the point quadtree [30]. A quadtree can be interpreted as a space dividing hierarchical data structure, where at each node the underlying space is subdivided into 4 subnodes (children) along axis-parallel lines that go through a given data point. The data points are stored at all nodes of the hierarchy, i.e., inner nodes as well as leaf nodes. If no data point is available for a leaf node, it is left empty. Figure 2.10 shows at the very left an example of such a quadtree. In the upper part of the figure, the quadtree is shown as a space partitioning data structure, in the lower part it is shown as a tree data structure with the root node at the top. Each inner node of the quadtree has exactly 4 children, each of them can be empty, hold a point, or link to children of their own. Searching or inserting a point triggers a $O(log(N))$ time complexity search from the root node to the required node (in this thesis the big O notation as defined by Knuth [57] is used), where N is the total number

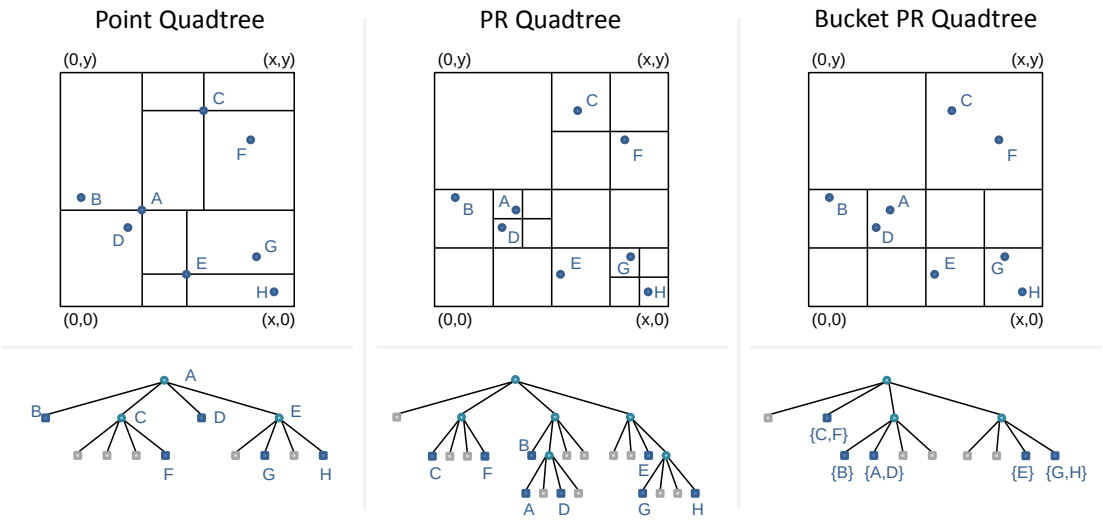


Figure 2.10: Three different quadtree versions holding the same 8 data points. The upper part of the figure shows the quadtrees as space subdividing data structures, while the lower part shows the tree structures of the quadtrees. The point quadtree on the left subdivides the space at data points. The PR quadtree in the center subdivides the space at arbitrary positions (here at the centers of the nodes) and stores the data points only at leaf nodes. The bucket PR quadtree on the right is similar to the PR quadtree, but stores multiple points at the leaf nodes. Figures inspired by Samet [92].

of points inserted. Deleting a point is quite involved, as the hierarchy has to be maintained also when points from interior nodes have been deleted, and the simplest method to do this is to rebuild the data structure from scratch. The extension of a quadtree to 3D space is called octree, as in this case each inner node can hold at most 8 children.

While a point quadtree subdivides the space at the data points, other variants, like the point region quadtree (PR quadtree), subdivide the space according to other rules. In the center column of Figure 2.10, a PR quadtree is shown. Nodes of the PR quadtree are subdivided at arbitrary axis-aligned lines, e.g., the ones through the center of the node. Another difference to the point quadtree is that the data is only stored at the leaf nodes of a PR quadtree, while the interior nodes are used for directing a search to the proper node. The time complexities for inserting and searching points in a PR quadtree are dependent on its height, which in turn is dependent on the minimum distance between 2 neighboring points. To alleviate this dependency on the distance of two points, a bucket PR quadtree can be used. In Figure 2.10, at the very right, such a quadtree is shown. In this variant of the PR quadtree, up to m points are allowed to be stored in a leaf node, and the example in Figure 2.10 uses $m = 2$.

2.2.2 Hardware

When working with large data sets that cannot be held completely in main memory, special external memory data structures and algorithms are necessary, which can be used to store and

process such data sets. The largest bottleneck when working with data stored on external memory is accessing and transferring the data. On current generation hardware, the access times for data in main memory are about 20 nanoseconds (ns), and transfer rates are approximately 20 Gigabytes per second (GB/s). These performances are achieved by read operations for DDR3 memory chips [83]. Write operations can only be done with about 14 GB/s. A current generation hard disk has data access times of about 5 to 10 milliseconds (ms), and transfer rates of about 100 to 200 Megabytes per second (MB/s) [100]. This is 6 orders of magnitude slower for access times, and 2 orders of magnitude slower for the transfer rate compared to main memory performance figures. The transfer rate is furthermore dependent on the transferred block size. In the worst case, for small block sizes, the transfer rate might decrease even further by 2 orders of magnitude. When using solid state drives (SSDs) (which also use memory chips for storage, but another kind than is used for main memory) instead of hard disks, the access times are still about 4 orders of magnitude slower (about 0.1 ms) compared to main memory access times. The transfer rates are roughly one order of magnitude slower with about 500 MB/s. In the worst case, again for small block sizes, the transfer rate might decrease by one order of magnitude [101].

2.2.3 General Data Structure Requirements

From the hardware properties of current generation computers it can be seen that there exists a huge performance gap between accessing data on external memory devices and accessing data in main memory. The largest performance gap is in the access time, therefore the number of accesses to external memory should be kept as low as possible. Vitter [105] suggests, that processing data sets out-of-core can be done efficiently, if an algorithm loads chunks of data into memory before processing them and only works on these chunks. Samet [92] mentions, that holding recently accessed chunks in a cache store could further reduce the number of disk accesses when processing data. Data structures supporting these techniques should hold the data points in buckets, like the bucket PR quadtree, as these buckets can readily be used as chunks that are transferred from and to external memory, and therefore the number of disk accesses can be lowered when compared to transferring single points from and to external memory separately.

Spatial databases provide a higher level of abstraction for accessing point data, as they organize the data in a way that database queries can be processed efficiently [33]. Typical queries are for example point queries, region queries, or nearest neighbors. The quadtree (or octree for point data sets in 3D space) is also used for spatial databases. The data structures used for spatial databases are primarily chosen for fast access to the point data, but not so much for efficient rendering. For example, they do not consider the problem of rendering LODs by providing a subsampled version of a whole point data set.

2.2.3.1 B-Trees

A generalization of binary trees (whose nodes have at most 2 children), quadtrees, and octrees, are n-way trees, which allow for any number (≥ 1) of children at a node. A well known n-way tree is the B-tree [7]. For a B-tree, the maximum number of children for a node can be defined by the user. Like the bucket PR quadtree, a B-tree stores the data in its leaf nodes, and the interior nodes store information to direct searches for the data. A difference to a bucket PR quadtree

is, besides allowing for an arbitrary number of children, that the leaf nodes are all at the same tree level. A B-tree has only 3 to 5 levels typically [8], and therefore it is a very efficient data structure for accessing points on slow external storage devices, because the number of accesses to the external storage device is kept low. B-trees and its variants like the B+ tree (which links neighboring leaf nodes for faster access) or the B* tree (which achieves a higher fill rate in interior nodes by merging two neighboring nodes under certain conditions) are actually used by file systems for operating systems. For example NTFS, the Windows file system [74], uses B-trees to manage the files in large folders.

2.3 Rendering Gigantic Point-Based Models

The basic rendering pipeline as shown in Figure 2.1 gives an overview of the different coordinate systems, which the geometry of a rendered object passes through. In an enhanced rendering pipeline, several other stages can be added, and so shading effects can be applied, tessellation of meshes can be done, or many other things can be performed. Figure 2.11 shows the rendering pipeline of OpenGL 4 [56], which is a graphics library that implements an advanced rendering pipline and provides a standardized application programmers interface (API) for rendering geometry onto 2D raster displays.

The OpenGL rendering pipeline is usually executed on dedicated graphics processing units (GPUs) but can also be executed on CPUs. GPUs are specialized in processing the stages of the rendering pipeline as fast as possible, and wherever possible in parallel (i.e., in stages where no mutual dependencies between processed elements exist). Some stages of the rendering pipeline (shown in blue in Figure 2.11) can be controlled by a programmer. The programs for these stages are termed shaders, although they do not necessarily change the shading of the rendered geometry.

In modern computers, the geometry (and all other data) of the rendered models is usually stored in video memory, which is memory that can be accessed fast by the GPU. This is necessary, since the GPU often resides on a plug-in card, and so the data would have to be transferred over a relatively slow bus from the main memory to the GPU, if the data were stored in main memory. Instead, the video memory and the GPU are on the same plug-in card, resulting in fast access times from the GPU to the data.

The first programmable stage in the OpenGL rendering pipeline is the vertex shader. The vertex shader is executed for every vertex that enters the rendering pipeline. In this shader, the geometric transformations from object space to view space take place, and also the perspective projection. After this projection, the vertices are then in so-called clip space, which is still a 3D space, but the view frustum has been transformed to a cube. This way, objects further away from the viewpoint were moved closer together in clip space.

The next programmable stages are related to tessellating patches of vertex data, i.e., subdividing patches into smaller patches. These stages are primarily for increasing the resolution of polygonal meshes in the rendering pipeline. Following is the geometry shader, which takes a single rendering primitive (e.g., a point or a triangle), processes it, and may output zero, one, or more primitives. This might be used for example to increase the density of point clouds, assuming sufficient information about the sampled surface is available.

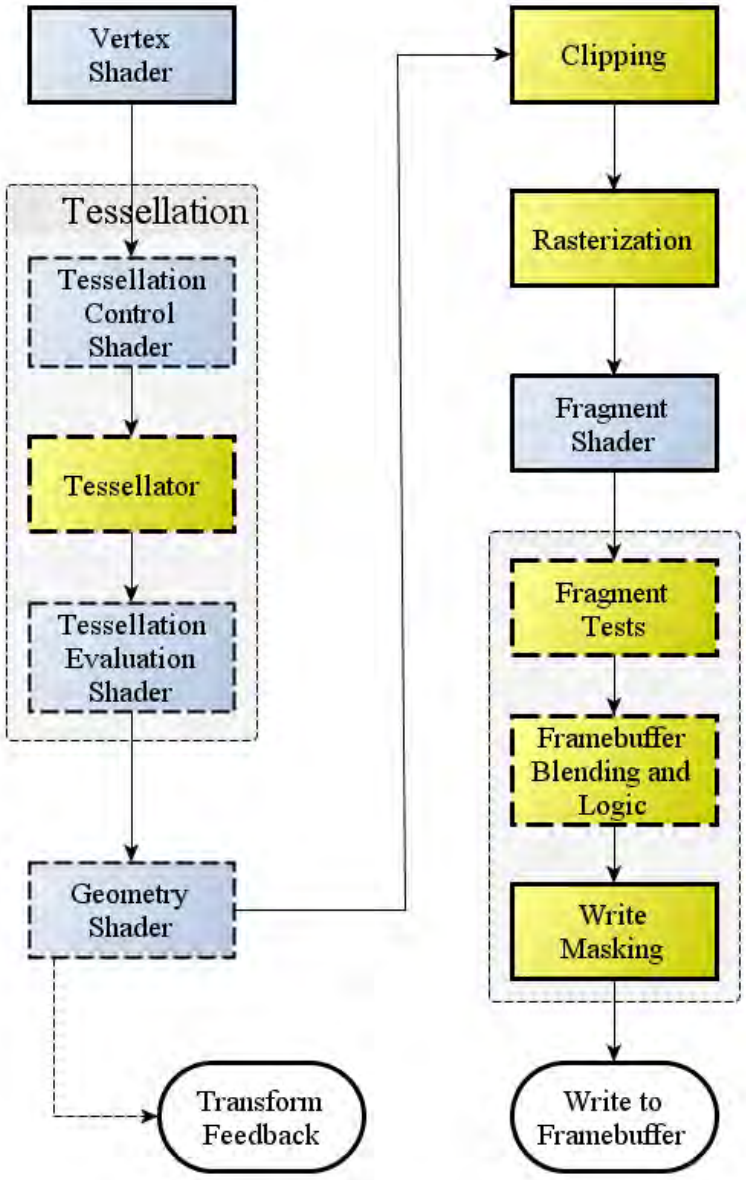


Figure 2.11: The OpenGL 4 rendering pipeline. All these stages are executed on the GPU. For the blue stages, shaders can be written to customize the behaviour of the pipeline. Image taken from the OpenGL website [56].

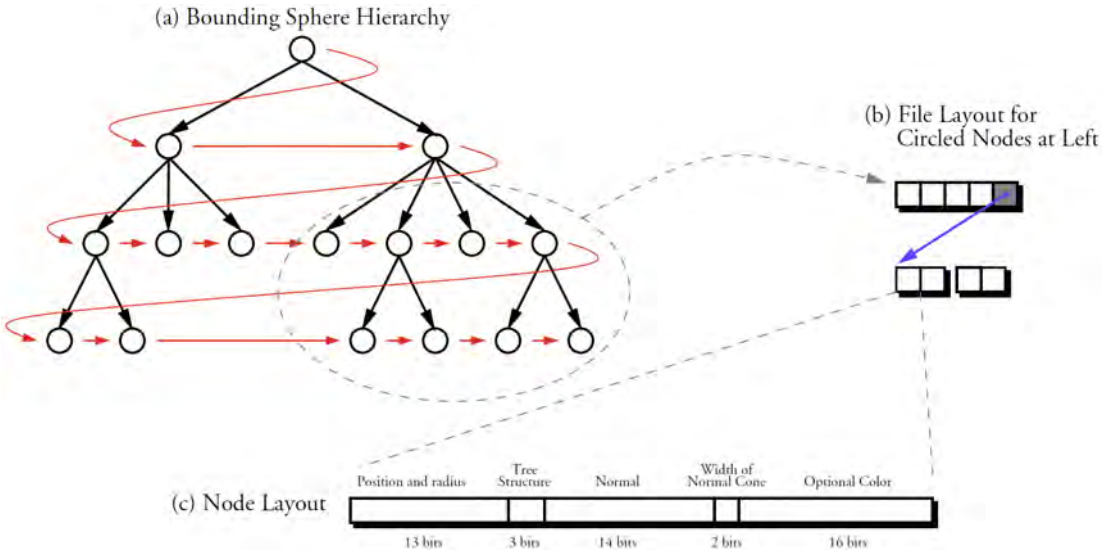


Figure 2.12: The file and node layout of QSplat. (a) The tree is stored in breadth-first order. (b) The link from parent to child nodes is established by a single pointer from a group of parents to the first child. At leaf nodes, the pointer is not present. (c) A single node occupies 48 bits. Image from Rusinkiewicz and Levoy [91].

Before the last programmable stage, the fragment shader, is executed, the rendering primitives have been rasterized. A single fragment maps to a single pixel, but has more information stored than the color of the fragment, e.g., depth or other information that can be used to calculate or modify the final color of the fragment. The result of this stage can be written directly to (or blended with) the according pixel.

Recent point rendering algorithms make use of the programmable stages in the rendering pipeline of modern graphics cards, as can be seen in the following, where different rendering algorithms for gigantic point clouds are reviewed.

2.3.1 QSplat

The first data structure supporting the properties mentioned before (see Section 2.2.3) was proposed by Rusinkiewicz and Levoy [91]. They presented a data structure which they called QSplat. They use a bounding sphere hierarchy to store the points. Such a hierarchy is a tree-like structure, where each node is associated with a bounding sphere. The data points are in the leaf nodes. An interior node has 2 children, and the bounding spheres of the children are completely enclosed by the bounding sphere of the parent node. The interior nodes store averaged values of their child nodes. In an optimization step, interior nodes are merged, so they have 4 children on the average, which also reduces the hierarchy's overall size in memory. In Figure 2.12, the tree structure of the bounding sphere hierarchy is shown. The nodes are stored sequentially on disk by traversing the hierarchy in breadth-first order (starting with the root node), which means that all nodes of a level are stored sequentially in the file before the nodes of the next level are stored.

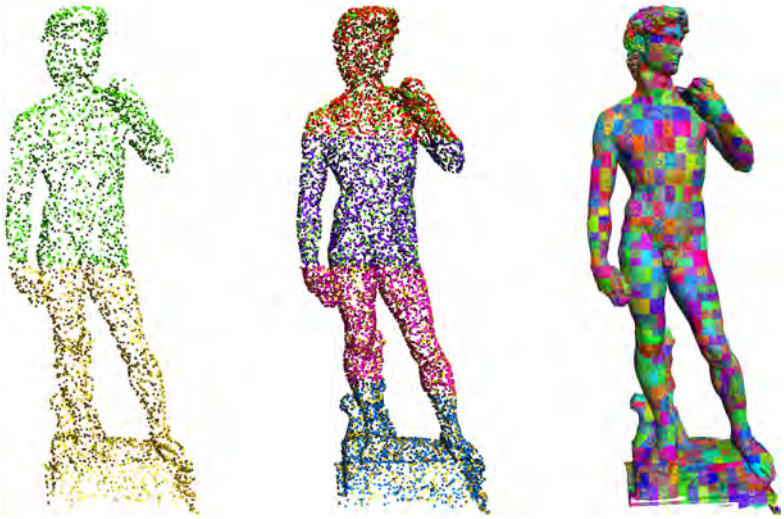


Figure 2.13: The point model of the David statue rendered at different LODs. The model in the left image is composed of 2 smaller point clouds. The model in the center is rendered with 4 more point clouds, refining the 2 point clouds of the previous LOD. The model in the right image is composed of 1720 single point clouds and rendered with a maximum projected point distance of 1 pixel. Image from Gobbetti and Marton [38].

The sequence in which the nodes are processed is shown with the red arrow in the tree structure. The nodes on disk are accessed as memory-mapped files, so the operating system handles the transfer and caching of the points. Each level of the hierarchy represents the complete point cloud at one LOD. The level which is used during rendering depends on the projected sizes of the bounding spheres of the nodes. If the projected size of a node is within certain bounds, then this level is chosen for rendering, and if necessary, the points are loaded from disk. Complete branches of the hierarchy can be skipped for rendering, if the bounding sphere of a node is outside the view frustum.

Due to the fine grained partition of the point data by the bounding sphere hierarchy (each leaf node stores only one point) the points cannot be transferred efficiently to the graphics card, since this is done most efficiently with blocks of data (similar as the data transfer from and to external memory).

2.3.2 Layered Point Clouds

The layered point clouds (LPC) hierarchy presented by Gobbetti and Marton [38] uses a different approach to handle large data sets. Each node of the hierarchy stores a point cloud of up to M points, which is always available in one block. Therefore, this data structure is well suited to provide the points in a form that can be processed efficiently by a graphics card. The data structure is built from a uniformly sampled point cloud (i.e., the distances between neighboring points are approximately equal) by subsampling it. The bounding box for the complete point cloud has to be known. For the first node, the root node, M uniformly distributed points are

chosen randomly, and these points are then stored there. Afterwards, they are removed from the input point cloud. The remaining input point cloud is divided into two parts across the center of the longest axis of the bounding box, and each of the two resulting point clouds acts as input point cloud as before (for its respective hierarchy branch), but with a bounding box half the size of the original point cloud. These smaller input point clouds are then subsampled and subdivided further, until the number of points for a node is smaller or equal to M.

For rendering, the complete hierarchy is split into a point cloud repository and a traversal hierarchy, which only contains the information necessary to traverse the hierarchy and the information which points belong to which node. During rendering, the hierarchy is traversed down to the required level for the current view position, which depends on the maximum projected distance between the points. Only nodes inside or intersecting the view frustum are considered. Point from hierarchy nodes that should be rendered but are not available on the graphics card are requested from the point cloud repository and are then loaded from disk.

The LPC hierarchy does not need additionally created points for the LOD mechanism but uses the points of the original point set to create the LOD levels. Each level in the hierarchy refines the points of the parent level, and together they form a distinct level of detail. The hierarchy is therefore memory efficient, since no additional points have to be created for the LOD mechanism. Figure 2.13 shows the point model of the David statue (which is part of The Digital Michelangelo repository [64]) rendered at different LODs. The model on the left is rendered with 2 point clouds, which are refined by 4 more point clouds for the model in the center of the figure. The 4 additional point clouds refine the sampling density of the model shown on the left, and all 6 point clouds are rendered together for this finer LOD. The model on the right is rendered with 1720 point clouds, with a maximum distance between neighboring points of 1 pixel.

2.3.3 Sequential Point Trees

The sequential point trees (SPT) data structure proposed by Dachsbacher et al. [24] is actually not suited for rendering gigantic point clouds, because it requires all points to reside in graphics card memory. Yet, it servers as the base for other data structures that can handle gigantic point clouds [82, 110], therefore an overview of the data structure is given here.

The SPT uses a hierarchical data structure that is sequentialized so it can be processed very efficiently on the GPU. Since it is sequentialized, it can be processed linearly, and a tree-like traversal is not necessary.

For building up the data structure, the points of the original point cloud are first inserted into an octree. It is assumed that the points are evenly distributed. The original points are in the leaf nodes, and the diameters of the bounding spheres at the leaf nodes should be roughly equal. The points will be rendered as splats, and the diameters of the bounding spheres are used as the splat sizes for the leaf nodes. The inner nodes of the octree represent the averaged information of their children. For an inner node, a splat is computed with a size just large enough to completely cover the splats of its child nodes. The quality of the splat coverage is evaluated by two error measures, the so-called perpendicular error and the tangential error. The perpendicular error measures the error when looking at the silhouette edges of an object. The tangential error measures how exact a splat covers its child splats when looking in the splat's negative normal direction onto the child

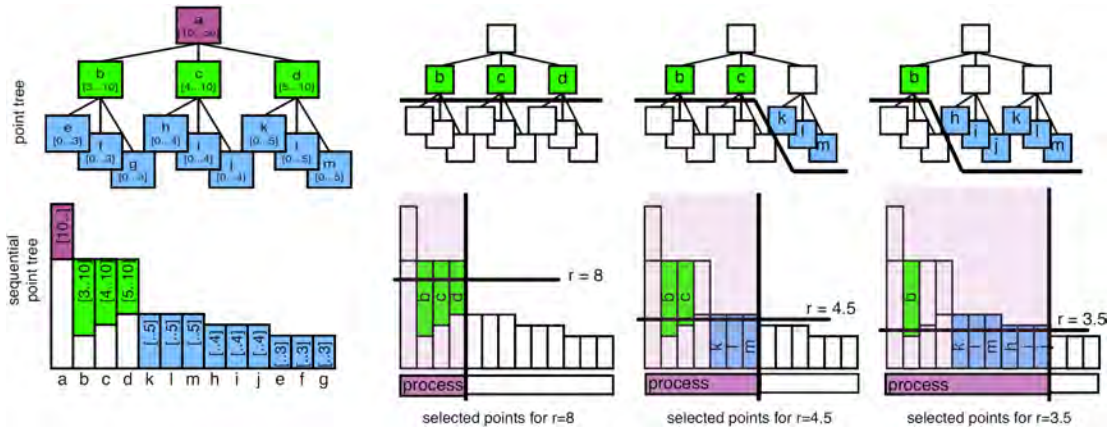


Figure 2.14: The upper row shows points in a point tree, and the lower row shows the same points in an SPT. In the very left column, the nodes a..m are sorted from the point tree into an SPT. In the SPT, they are ordered according to their r_{max} value. In the other columns, the black line in the upper row of the images shows the tree cuts for different view distances. In the lower row, the cut is shown with the horizontal black line. The filled process bar shows which nodes are sent to the GPU, and the nodes with an r_{min} value larger than the horizontal black line are culled by the GPU. Image from Dachsbacher et al. [24].

splats. The errors can be combined to a geometric error $\tilde{e}_g$. If this hierarchy would be used for rendering, the geometric error would be projected to screen space, resulting in an image error $\tilde{e}$, measured in pixels. If this image error would exceed a user defined error threshold ϵ, then the recursion would go one level down the hierarchy, else a splat with the size stored at the node would be rendered.

The data structure so far can only be processed by the CPU. Processing the data structure by the GPU requires a different error measure. When traversing a hierarchy, it is implicitly known if an ancestor node has been rendered, because in this case, the according subtree does not have to be processed. If the data structure is sequentialized, it is not known by the nodes in the according subtree if an ancestor has already been rendered, so the algorithm has to check for this case explicitly. The SPT rendering algorithm uses two simple error measures, which determine if a node (i.e., a splat) shall be rendered. It calculates an r_{min} and an r_{max} for each splat, which is the minimum and maximum distance to the viewpoint for which the splat will be used. If the distance r lies within, the splat will be rendered. The test in the hierarchical traversal is whether $\tilde{e} = \tilde{e}_g/r < \epsilon$, which can be rewritten as $r_{min} = \tilde{e}_g/\epsilon$. The value r_{min} is stored at each node. The value for r_{max} is calculated by taking r_{min} of the parent node and adding an interval overlap (the distance between the two nodes), which will result in a display with no holes in it. These error measures can be checked by a vertex program on the graphics card.

The r_{max} value allows for another optimization. The left column in Figure 2.14 depicts a hierarchical point tree represented as SPT. If the SPT is sorted by r_{max}, then the CPU can cull the nodes whose r_{max} value is too small for the current viewpoint, so that they will not be

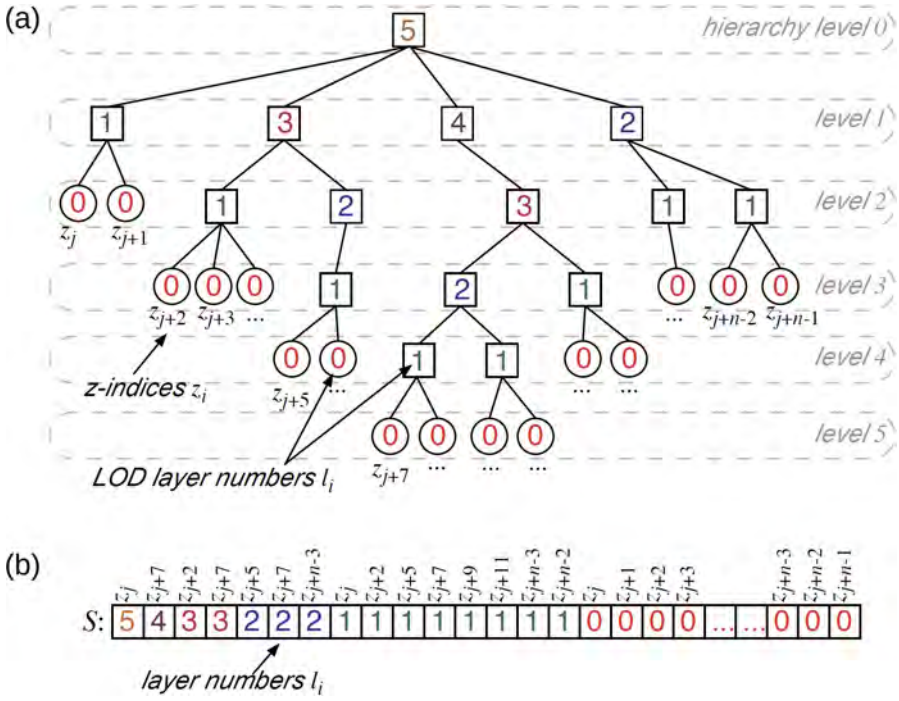


Figure 2.15: Drawing (a) shows a point tree, where for the leaf nodes the z indexes have been calculated, and all nodes have been associated with a layer index. With these indexes, a hierarchical LOD classification can be performed. Drawing (b) shows the sequentialized hierarchy S ordered first by decreasing layer and within each layer ordered by increasing z index. Images from Pajarola et al. [82].

rendered. It then sends all points from the first one in the list to the last one with an r_{max} larger than r into the rendering pipeline, so the GPU is relieved to cull fewer nodes. In Figure 2.14, 3 situations with different r are shown from the second to the left column to the very right column. The SPT is sorted by r_{max} only once in a pre-processing step.

While the SPT rendering algorithm allows for fast GPU processing and rendering of the sequentialized hierarchy, it has some shortcomings as well. It does not allow for view-frustum culling within a point cloud, therefore always all points of the chosen LOD levels have to be processed, even if they are outside of the view frustum. Furthermore, it is limited to rendering point clouds that completely fit into the memory of a graphics card, as no memory management method is implemented.

2.3.4 XSplat

The XSplat hierarchy proposed by Pajarola et al. [82] extends the SPT hierarchy to enable the handling of point-cloud data that does not fit completely into the memory of the graphics card. Furthermore, it enables view frustum culling for parts of the point cloud that are not inside the view frustum. This is achieved by using a block-based sequentialized multiresolution hierarchy.

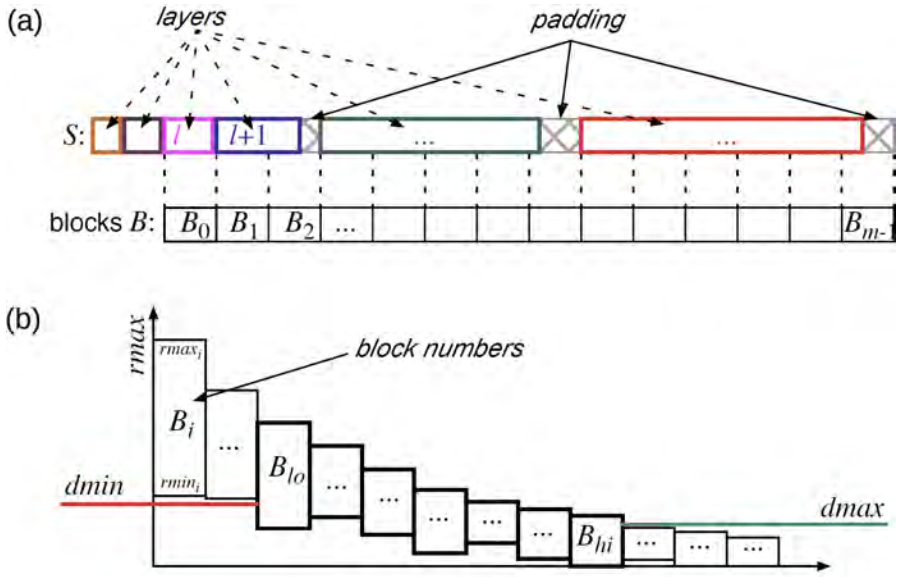


Figure 2.16: Drawing (a) shows S subdivided into blocks. If a block cannot be filled completely, it is padded with NULL points. Drawing (b) shows the ordering of the blocks according to their $rmax$ value. For some given viewpoint, the $dmin$ and $dmax$ values are charted, and the blocks chosen according to these values. Images from Pajarola et al. [82].

The XSplat hierarchy can be built from some input hierarchy, where the original points are at the leaf nodes (each leaf node has to hold one point), and the interior nodes average the information of their children. This can be for example a point octree (see also Section 2.2.1). Each hierarchy node has to store the attributes required for rendered the contained point as splat, like the position of the point and the bounding sphere radius. For each node an $rmin$ and $rmax$ value are calculated, similar to the distances used by the SPT hierarchy [24] (see also Section 2.3.3), but with a simpler error metric that uses only the projected area of the bounding sphere in screen space. In the sequentialized hierarchy, the $rmin$ and $rmax$ values are used to decide if a node with position p and bounding sphere radius r shall be rendered for the viewpoint v, and this is done if $rmin \leq \sqrt{\epsilon} \cdot (|p - v| + r)$ and $rmax \geq \sqrt{\epsilon} \cdot (|p - v| - r)$, where ϵ is the screen space error threshold. Furthermore, for each node also a layer index l is calculated, which is the length of the longest path in the hierarchy from that node to a leaf node (in the subtree below it). It is expressed in the number of edges (an edge connects two nodes) the path consists of. Finally, an index z is calculated for each node. With a proper traversal of the input hierarchy, a linear index on the leaf nodes can be calculated that defines a hierarchical z-order on them [31]. Derived from the z-order of the leaf nodes, an inner node gets the z index that is the smallest one of its direct or indirect children. In Figure 2.15, Drawing (a) shows a point tree where the layer indexes and the z indexes have already been generated, and each node is assigned an index-pair (l_i, z_i).

After the necessary indexes have been calculated, the hierarchy is sequentialized. The sequentialized hierarchy S is then sorted according to the index pairs in lexicographical order, with

decreasing layer index and increasing z index. In Figure 2.15, Drawing (b) shows the sequentialized and ordered hierarchy of Drawing (a). The sorting according to the previously found indexes causes nodes, which are likely to be rendered at the same LOD, to be close together in S.

This sequentialized hierarchy S could already be used for rendering models that fit in the memory of the graphics cards. To allow for rendering of larger models, the hierarchy is subdivided into blocks, as depicted in Drawing (a) of Figure 2.16. The number of blocks is an order of magnitude smaller than the number of points within the hierarchy, and therefore the list of blocks B can be managed in main memory. Each block in B averages the attributes of the nodes contained within, e.g., the $rmin$ and $rmax$ values of a block are the $min(rmin_i)$ and $max(rmax_i)$ values of the all nodes i contained in the block. Drawing (b) in Figure 2.16 shows the blocks ordered by decreasing $rmax$.

For rendering, values $dmin$ and $dmax$ are calculated depending on the current viewpoint and screen space error tolerance ϵ, and all blocks intersecting the interval $[dmin..dmax]$ are chosen for rendering (see also Drawing (b) in Figure 2.16). The blocks can be cached in the memory of the graphics card to increase the rendering performance. The sequentialized hierarchy S and the list of blocks B are accessed with memory mapped files. The blocks can be swapped in and out of graphics card memory, depending on the coarse LOD block selection for the current viewpoint. The fine-grain LOD selection of single points is accomplished similar to the SPT rendering algorithm [24].

Compared to the SPT hierarchy, the XSplat hierarchy has the advantage that large models, which do not fit into the main memory of a computer, can be rendered. Nevertheless, the XSplat hierarchy has a similar problem as the SPT hierarchy, which is the memory requirement. Both hierarchies use additionally created points to represent the inner nodes of the original point hierarchy. These additionally created points prevent the rendering algorithms to reach the maximum number of rendered points per second that a graphics card is capable of. This is due to the inner points that have to be processed by the vertex shader and that will always be culled later on, because they are not required for the current LOD.

2.3.5 Instant Points

The nested octree hierarchy proposed by Wimmer and Scheiblauer [110] is also developed from the SPT hierarchy [24], like the XSplat hierarchy [82], and also allows for rendering huge point clouds that do not fit into main memory. Some optimizations for storing the points are employed, and so the nested octree hierarchy uses less memory than the SPT hierarchy. Since the modifiable nested octree (MNO) hierarchy presented in this thesis is based on the nested octree hierarchy, the nested octree hierarchy is discussed later on in the thesis, before the description of the MNO hierarchy (in Chapter 3).

2.3.6 Multi-way kd-Trees

Goswami et al. [40, 41] proposed a data structure that allows for a uniform subdivision (in the number of points) of large point clouds and is also well suited for a block-based transfer of

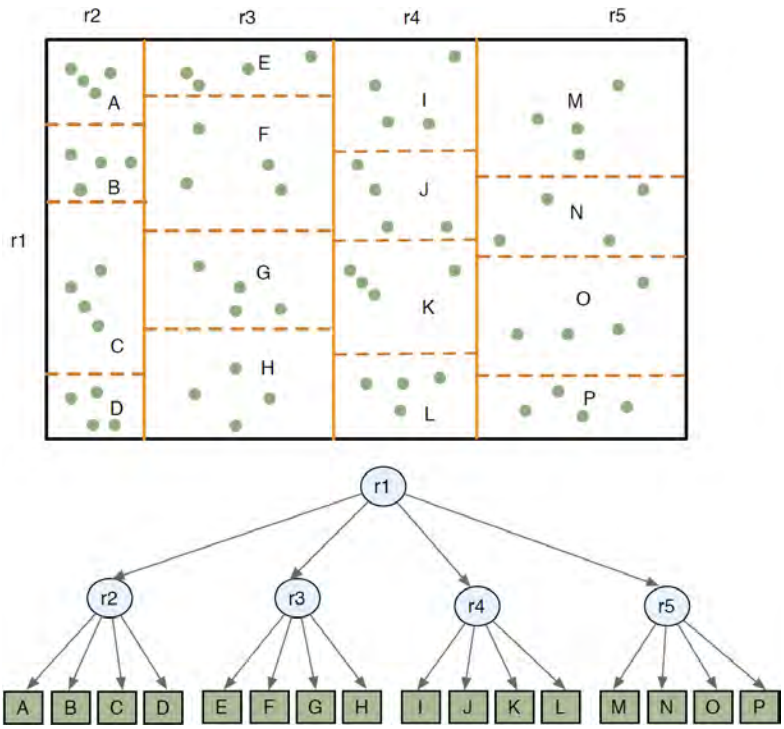
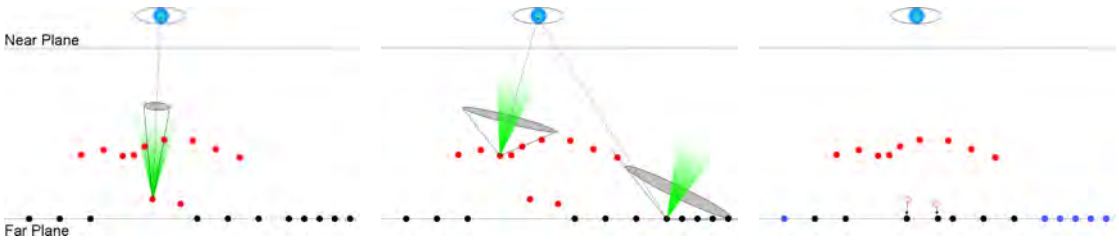


Figure 2.17: A multi-way kd-tree with a fanout of $N = 4$ and a leaf node capacity of $s = 4$. The nodes are subdivided such that the points are distributed equally to the child nodes. Image from Goswami et al. [41].

points. A uniform subdivision of the point-cloud data allows for an enhanced rendering performance and faster hierarchy management, as the points are better distributed across the nodes, resulting in a hierarchy with less nodes, compared to a hierarchy with unevenly distributed points across the nodes. Figure 2.17 shows the proposed multi-way kd-tree (MWKT) as space dividing hierarchy (top) and as tree structure (bottom). The MWKT is derived from the kd-tree, a binary tree that describes a subdivision of space in k dimensions along mutually orthogonal axes. At each level the space is subdivided along one of the axes. There are many variants of the kd-tree, depending in which axes order the space is subdivided, if the subdivision is at data points or arbitrary points in space, if data is only stored at leaf nodes, or if data is stored in buckets. One variant described by Samet [92], namely the bucket adaptive kd-tree, is similar to a MWKT, as it stores the original points in buckets in leaf nodes, and at each level subdivides the nodes, such that the points are distributed equally to the child nodes. Therefore, all leaf nodes hold approximately the same number of points. They differ in the fanout (i.e., the number of children of a node) and on the information stored at the inner nodes, as the MWKT is not strictly a binary tree (it can have a fanout of any $N \geq 1$) and stores an LOD representation of the hierarchy at an inner node.

The MWKT is built from a set of points, which is first sorted along the longest axis of its bounding box, resulting in a sorted list of points. The points are subdivided into N equal sized



(a) An occluded point (b) A visible projected and a background point (c) Visibility output

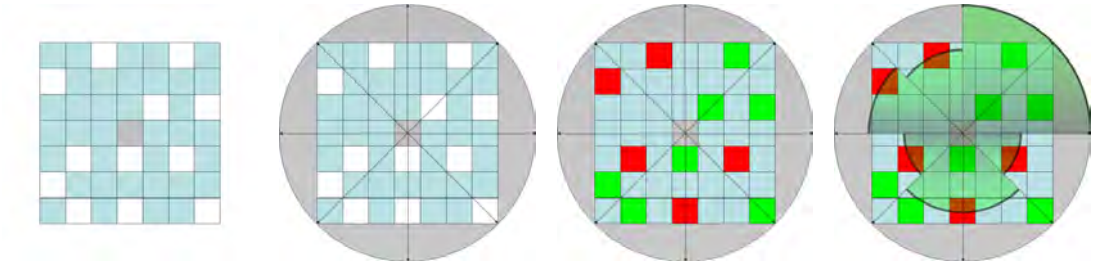
Figure 2.18: Visibility test by calculating the maximum aperture angle from a point to the viewpoint, so that no other points are inside the aperture. Drawing (a) shows the aperture for a point that is considered invisible, as the angle of its aperture is smaller than the angle of the green threshold aperture. Drawing (b) shows the apertures for two visible points. Drawing (c) shows the visible points (red dots), the visible background points (blue dots), and the invisible points (black). The background and the invisible points all have a depth value of 1 (i.e., they are on the far plane). Images from Pintus et al. [87].

point sets (the points in each set are adjacent in the sorted list), and pushed to the child nodes. This step is repeated recursively, and when the number of points in a set is below or equal to a given threshold s, then subdivision stops, and a leaf node is created. The points in a leaf node can be directly used for rendering. After distributing the original data points to the leaf nodes, a bottom up gathering process is started to create sets of points at the inner nodes that describe the points of their child nodes at a coarser LOD. Instead of using a regular grid at the inner nodes and averaging the information of all points inside a grid node to create a new point, different clustering approaches are proposed by Goswami et al. [40], which take the surface properties into account, from which the points were sampled. This way, only points belonging to a single surface contribute to a new point. Furthermore, clustering allows to create exactly s points for each inner node, which is usually not the case when a regular grid is used to subsample the points of the children. The hierarchy is split into an index tree (which is used for hierarchy traversal and references the point set for each node) and the point data, which is stored separately on disk.

The index tree has a small memory footprint and can be held in memory during rendering, while the point sets are loaded from disk as needed. The nodes for rendering are chosen based on a screen-space error metric. When changing the LOD, the points are geo-morphed from the previous LOD to the current LOD, which takes a few frames. During geo-morphing, the positions, normals, and colors of points are linearly interpolated between the two LODs. This reduces the visual artifacts compared to abruptly changing the LOD.

2.3.7 Screen-Space Visibility and Sparse Depth-Maps for Surface Reconstruction

While all previously presented rendering systems rely on already existing per-point normals and radii for rendering point models with closed surfaces and shading effects, Pintus et al. [87] proposed a rendering technique based on screen-space operators, which does so without requiring pre-calculated normals or radii. They use a kd-tree as point model data structure, where the



(a) Visibility search space (b) Sector subdivision (c) Horizon pixels (d) Maximum cones

Figure 2.19: Calculating the solid angle for a point projected to a pixel in screen space. Drawing (a) shows the pixel block where the solid angles are searched. The grey pixel in the center is the projection of the initial point, and the white pixels are other projected points. Drawing (b) shows the search space subdivided into 8 sectors. In Drawing (c), the red pixels are the horizon pixels for each sector, i.e., the pixels representing the points furthest away from the initial point in 3D space, such that no other points are inside the solid angle. The green pixels represent points outside the solid angles. In Drawing (d), the solid angles are visualized for all sectors. Images from Pintus et al. [87].

original points are stored at the leaf nodes in buckets of size M. For build up, the points of the original point data set are inserted into the kd-tree in a top-down manner. When the number of points in a node is larger than M, the set of points is split along the longest axis of the bounding box, and the points are inserted into the child nodes according to their position. After all points have been inserted, additional points for the inner nodes are calculated, which are used for the LOD representation. At each inner node a regular grid is created, where at each cell the points of the child nodes are averaged, creating a new point at each cell. If no points fall into a cell, the cell remains empty.

During rendering, several steps are executed every frame. The points are projected to screen space, occluded points are discarded, a closed surface from the remaining points is calculated, and a shading is calculated that emphasizes the edges of the sampled surfaces. Figure 2.18 shows the visibility test in 2D. The red dots are points projected to screen space that survived the z-buffer test (i.e., points closer to the viewpoint than other points projected to the same pixel), and black dots are background pixels. From each pixel, a ray along the line-of-sight from the viewpoint is set up, and an aperture from the pixel to the viewpoint is calculated. This aperture has the apex at the pixel with an angle that is the largest possible, such that no other point is inside this aperture. If this angle is smaller than a pre-defined threshold, the pixel is assumed to be invisible and will be marked accordingly. Drawing (a) in Figure 2.18 shows a pixel with an aperture whose angle is smaller than the angle of the green threshold aperture. The neighboring pixel is also considered invisible, therefore both pixels are marked as invisible, as can be seen in Drawing (c). In Drawing (b), the apertures for two visible pixels are shown. In Drawing (c), visible pixels from projected points are shown with red dots, visible background pixels are shown with blue dots, and pixels that are considered invisible are shown with black dots.

The visibility test for a point cloud in 3D space is done by projecting the points to screen space, and calculating the object space solid angle from the points represented by the pixels (each

(a) Level 0 (b) $K = 30$ levels, level weight $\xi = 3/2$ (c) $K = 30$ levels, level weight $\xi = 0$

Figure 2.20: Different stages of the shading calculation. Drawing (a) shows the result after directional light shading and edge enhancing. This is also the initial smoothing step 0. Drawing (b) shows the resulting image after adding up the contribution of 30 smoothing steps, with the weighting factor $\xi = 3/2$, which favors the first smoothing steps. Drawing (c) shows the resulting image after adding up 30 smoothing steps with equal weight for each step. Images from Pintus et al. [87].

point is projected to a single pixel). In Figure 2.19, the steps to find the solid angle are outlined in screen space. Drawing (a) shows the initial pixel (grey) and the neighboring pixels (white), which constitute the search space. In Drawing (b), the search space is divided into 8 sectors. For each sector, the horizon pixels are calculated from the 3D positions of the points. These are pixels, whose points, when connected to the point of the initial pixel, are on a generatrix of the solid angle cone (which has its apex at the initial point). Drawing (c) shows the horizon pixels in red, while the pixels that are outside or behind the solid angle cone are green. For each sector, a separate solid angle cone piece is calculated, and the resulting pieces are shown in Drawing (d). Afterwards, the integral from all pieces is taken, and if it is above a given threshold, the initial pixel is marked visible.

The resulting pixel grid, which contains only the front-most visible pixels, can be described as a sparse depth map. An anisotropic filling algorithm then creates a closed surface from the existing pixels, by iteratively closing the holes in a multi-pass GPU fragment shader. For empty pixels, a 3x3 neighborhood is considered. From this neighborhood, averaged information is used to reconstruct the color and depth for an empty pixel.

After reconstructing the surface, a direct lighting, a line drawing, and an ambient occlusion effect are derived from the depth values of the pixels. For each pixel i a 3x3 neighborhood is considered to find the expected depth value μ_i^0 and the variance σ_i^0. From these, the values $d_{i_{min}}^0 = \mu_i^0 - \sigma_i^0$ and $d_{i_{max}}^0 = \mu_i^0 + \sigma_i^0$ are calculated, which can then be used to calculate a shading term, emulating direct illumination from the viewpoint, with the equation

$$\omega_i^0 = \underset{[0..1]}{clamp}\left(1 - \frac{|d_i^0 - d_{i_{min}}^0|}{d_{i_{max}}^0 - d_{i_{min}}^0 + \epsilon}\right) \tag{2.1}$$

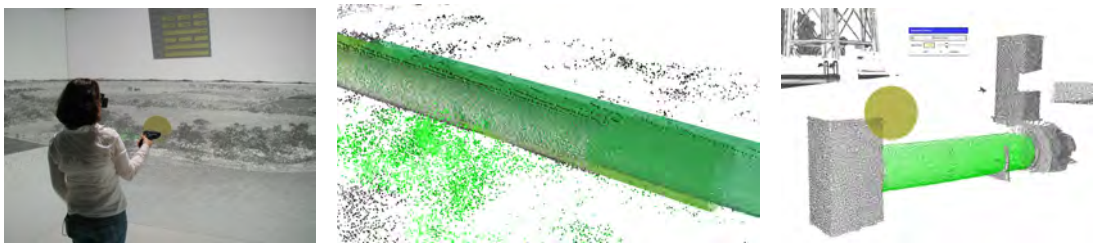
where d_i^0 is the depth of pixel i, and ϵ is a small constant, to avoid division by zero. The distance $\hat{d} = |d_i^0 - d_{i_{min}}^0|$ and the distance $\tilde{d} = d_{i_{max}}^0 - d_{i_{min}}^0$ can be used to shade the surface, and to accentuate elevations and cavities. In flat areas looking towards the viewpoint $\hat{d} \approx \tilde{d}/2$, so the shading value $\omega_i^0 \approx 0.5$. The larger $\hat{d}$ relative to $\tilde{d}$, the smaller ω_i^0, so flat areas tilted away from the viewpoint will be shaded darker. From the same shading term results also a line drawing effect. For edges with curvature towards the viewpoint (i.e., elevations) the distance $\hat{d}$ becomes smaller, so these elevations will be shaded with $\omega_i^0 \approx 1$ (but at most 1). Edges with curvatures away from the viewpoint (i.e., cavities), the distance $\hat{d}$ becomes larger, so these cavities will be shaded with $\omega_i^0 \approx 0$ (but at least 0). Since $d_{i_{min}}^0$ and $d_{i_{max}}^0$ are not the absolute minimum and maximum depth values, but derived from μ_i^0 and σ_i^0, distances beyond σ_i^0 go into saturation in the shading, emphasizing further the elevations and cavities. In Figure 2.20, Drawing (a) shows the shading after direct lighting and edge enhancement. The edge enhancement is especially visible in the hair of the model's head.

Finally, an approximation of a global illumination effect, called ambient occlusion, is derived from the depth values. Ambient occlusion simulates the effect that objects close to a surface cast a shadow onto the surface. As light source, ambient lighting is assumed, which is non-directional light that uniformly illuminates a scene. Since for an exact evaluation of the shadowing a complete global illumination solution of the scene would have to be calculated, it would be a quite expensive computation. Therefore, an approximation is calculated by considering only the distance to the geometry surrounding the surface. This can be done in screen space for a single pixel by looking at the depth values of neighboring pixels, and for example pixels "in cavities" will be shaded as if they would receive less ambient light. Pintus et al. propose calculating the ambient occlusion by iteratively smoothing the depth values. The depth value d_i^k in iteration k is derived from the depth value of the previous iteration $k-1$ by $d_i^k = \left(\mu_i^{k-1} + d_{i_{min}}^{k-1}\right)/2$. For each iteration ω_i^k is calculated. After K shading steps, the total shading value for pixel i is calculated by

$$\omega_i = \sum_{k=0}^{K-1} \frac{\omega_i^k}{(k+1)^\xi} \tag{2.2}$$

where ξ is a weighting factor which determines the amount of ambient occlusion visible in the final image. A value of $\xi = 0$ weights every iteration step equally, while for increasing values of ξ the first iteration steps get ever higher weights. In Figure 2.20, Drawing (b) shows the result after summing up the images of 30 smoothing iterations with $\xi = 3/2$ (favoring the first smoothing steps). The ambient occlusion effect makes the model appear more plastic compared to Drawing (a), but the enhancement of the edges is also softened. Drawing (c) shows the sum of 30 smoothing steps with $\xi = 0$.

In this thesis, a simple surface reconstruction algorithm is presented that works by adjusting the sizes of the rendered splats. The sizes are calculated before the points are entering the rendering pipeline on the graphics card, therefore no screen-space evaluation is required. The increased rendering speed comes at the expense of a lower quality reconstruction (see also Chapter 3).



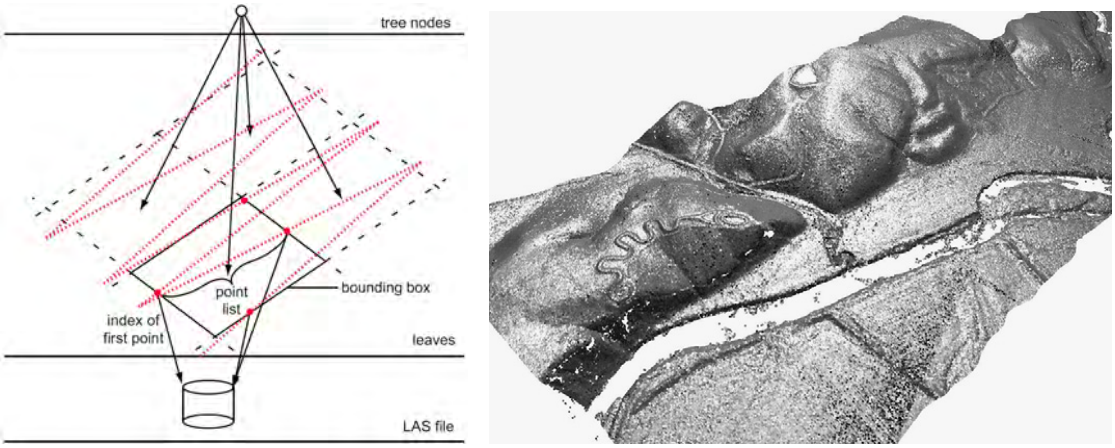
(a) Immersive visualization (b) Misaligned tube from two scan positions (c) Point selection

Figure 2.21: Image (a) shows a user inside a CAVE. The user wears a head-tracked stereoscopic device, and he can interact with the point model by using a 3D input device. Image (b) shows a tube that is scanned from 2 different scan positions, but the alignment of the 2 scans is insufficient. Image (c) shows a tube selected by a selection sphere. The selected points can then be used to fit a mathematically defined cylinder to them, which can then be added to the visualization as a separate object. Images from Kreylos et al. [60].

2.3.8 Applications

Various point rendering systems were developed for the geoscience remote-sensing community for different application scenarios. This is due to the huge point clouds that are acquired by surveying swaths of land by airborne or ground-based laser scanning. In the following, three papers are presented that implement variations of the above described point rendering systems.

Kreylos et al. [60] propose a point rendering system that uses an octree hierarchy, where the original points are all stored in the leaf nodes in buckets. In the inner nodes, the set of points from the child nodes is subsampled by clustering neighboring points. Original points are then chosen as representative points of their clusters (this way, no artificially created points are introduced). The maximum number of points that can be stored at the nodes can be defined by the user. This way, a LOD hierarchy is created, where each level in the hierarchy represents a distinct LOD level. They use the point rendering system for immersive visualization, i.e., users are placed inside a virtual world. This is done with a head-tracking device, and the virtual world is rendered according to the movements of the head. Usually, also a stereoscopic display (where for each eye the world is rendered from a slightly different view position) is part of this immersive visualization, so users are experiencing depth cues from the visualization. The left image in Figure 2.21 shows such an installation where a user is inside a room, and the walls act as screens where the rendered point model is projected onto. Such a setup is called a CAVE. For a stereoscopic display the geometry has to be rendered twice, once for each eye, so fast rendering is essential. Approximately 60 stereoscopic frames per second are required for immersive visualization, therefore the fastest possible point rendering method is chosen, which is rendering the points as square screen-aligned splats. With the stereoscopic visualization, 3D interaction methods are possible, which can be used for quality control of lidar data or for feature extractions from the point data. The visual quality control can reveal misalignments of different scans due to an insufficient registration process. The center image of Figure 2.21 shows a tube scanned from two different positions, where the two scans are not aligned correctly. Additionally, features can be extracted from the point data, which can then be assigned to other geometric primitives, for



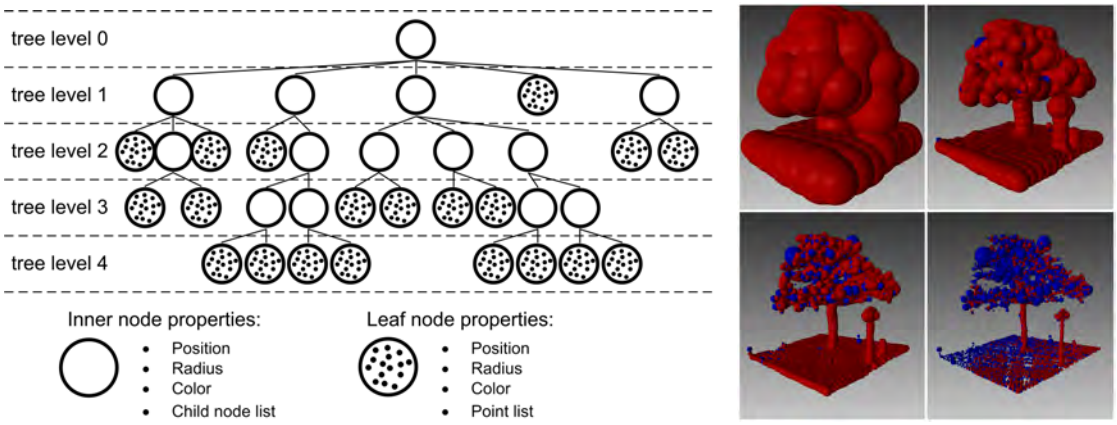
(a) Points from a flight swath indexed in leaf nodes

(b) Points rendered as oriented splats

Figure 2.22: Image (a) shows the sample points of an airborne laser scan along a flight swath. The sampling line starts at the bottom. The image also shows a quadtree leaf node that stores indexes into a LAS point file, and the number of points following the indexes. The indexes are indicated by the large red dots. Image (b) shows a point model rendered with oriented point splats. The normals for the points are calculated after loading them from disk during rendering. Images from Kovač and Žalik [59].

example planes, spheres or cylinders. Those primitives can then be added to the visualization. The right image in Figure 2.21 shows a cylinder selected by a selection brush (see also Section 2.4). The selected points are then used to match a mathematically defined cylinder to the points, according to a least-squares error metric. The selection brush can be swiped across the points, and the user can choose to select or deselect the points currently inside the sphere. A similar selection method is presented later on in the thesis, but instead of storing the selection information directly at the points, it is stored with the help of a selection octree (see also Chapter 3).

Kovač and Žalik [59] proposed a point rendering system, where the points can be directly rendered from a LAS point file [4] (a file format that was developed to store the points along the flight swaths recorded by airborne laser scanning). The distribution of the points in those files exhibits a strong regularity, i.e., the points are ordered along lines. Image (a) in Figure 2.22 shows how a flight swath is recorded by an airborne laser scanner. Since they only target rendering of airborne laser scans, they make the assumption that the data is approximately 2D, since the vertical extent of the data relative to the horizontal extent is small. In a pre-processing step, they build a quadtree from the point data stored in the input file, where the quadtree is divided until a node holds a predefined number of points. The leaf nodes do not store the points themselves, but only indexes into the point file. Image (a) in Figure 2.22 also shows the stored indexes for one leaf node, indicated by the big red dots. They also indicate the start of the point lists inside the bounding rectangle of the leaf node (the points are recorded in one survey, starting at the lower edge). The number of points after the start indexes are also stored. During rendering, the quadtree with the indexes is loaded to memory, while the points remain in the point file. Only nodes inside the view frustum are loading the points from disk. After the points



(a) Octree hierarchy

(b) Bounding spheres for different LODs

Figure 2.23: Image (a) shows the octree hierarchy of a small point cloud. The original points are stored in the leaf nodes, while the inner nodes store only a single point that averages the information from the points of its child nodes. Image (b) shows the bounding spheres of a small point cloud at different LODs. The red spheres represent inner nodes, while the blue spheres represent leaf nodes. Images from Richter and Döllner [89].

have been loaded from disk, per-point normals are calculated on-the-fly by calculating the cross product from nearby points. The points are then subsampled and distributed to the different levels of the hierarchy, creating a LOD representation of a loaded leaf node from the original points, without creating additional points. The normal calculations and subsampling are done in a separate thread, so the rendering thread is not halted during these operations. A problem with the dynamic creation of the LODs appears when the whole point model is viewed from a distance, because then all leaf nodes have to be loaded from disk to memory, in order to calculate their contribution to the coarser LODs. So in this case, actually the complete point data has to be loaded to memory, slowing down the presentation of the point model at the required LOD. Image (b) of Figure 2.22 shows the point model of an airborne laser scan rendered with normal aligned splats (see also Section 2.5), using the normals that were calculated after loading the points to memory.

Richter and Döllner [89] propose a point rendering system that uses a hierarchy similar to QSplat [91] (see also Section 2.3.1). The main difference is the storage of the original points in the leaf nodes. They propose to use buckets at the leaf nodes, so a certain number of points can be stored in one leaf node. This reduces the number of nodes in the hierarchy. The hierarchy is built by inserting points into an octree. The nodes are subdivided, until the number of points in the nodes is below the user-defined threshold. The inner nodes are then averaging the information from their child nodes, but store only one point. Image (a) of Figure 2.23 shows the octree hierarchy created for a small point set. All nodes of the octree store the bounding sphere of the points that are inside their volume. When rendering the points in the hierarchy, the bounding spheres are used for the view-frustum culling, and the projected sizes of the bounding spheres are used for the selection of the appropriate LODs. Image (b) of Figure 2.23 shows the bounding

Figure 2.24: Rendering of a point cloud which consists of $5*10^9$ points. Images from Richter and Döllner [89].

spheres of different LODs for a small point cloud. Red spheres represent inner nodes, while blue spheres represent leaf nodes. Bounding spheres, which are usually less well fitted to the points' distribution than bounding boxes, are used, because intersecting a bounding sphere with the view frustum is 3 times faster then intersecting a bounding box with the view frustum. This is important for the proposed rendering system, as the rendering front, i.e., the set of nodes that are used to render the point model from the current viewpoint, consists of about $2*10^5$ to $4*10^5$ nodes. This is about 2 orders of magnitude more than in other rendering systems (see also Chapter 3), due to the low approximation quality of the inner nodes, because they represent all points of their child nodes with just one averaged point splat. Because of this, the hierarchy has to be traversed to deep levels, if the projected size of the nodes shall only be some pixels. During rendering, a relative constant frame rate is achieved by deliberately limiting the number of nodes that can upload their points to the graphics card per frame. Without this limitation, drops in the frame rate would occur, if too many points were uploaded in a single frame. Figure 2.24 shows a point cloud of $5*10^9$ points rendered with the proposed system.

2.4 Editing Gigantic Point-Based Models

Editing gigantic point-based models requires data structures with different properties than are used for rendering such models, as the points may be changed, deleted, or new points may be added. Interactive editing, in comparison to automatic processing, is favorable for tasks that can be done most effectively by a human, like selecting certain areas inside a model. This interaction not only requires data structures efficient for editing, but also for displaying the models, as otherwise the user could not move around when inspecting the models.

Zwicker et al. [115] proposed an interactive point editing system, called Pointshop 3D, to alter the shape and appearance of point-based models. They developed an interactive point cloud

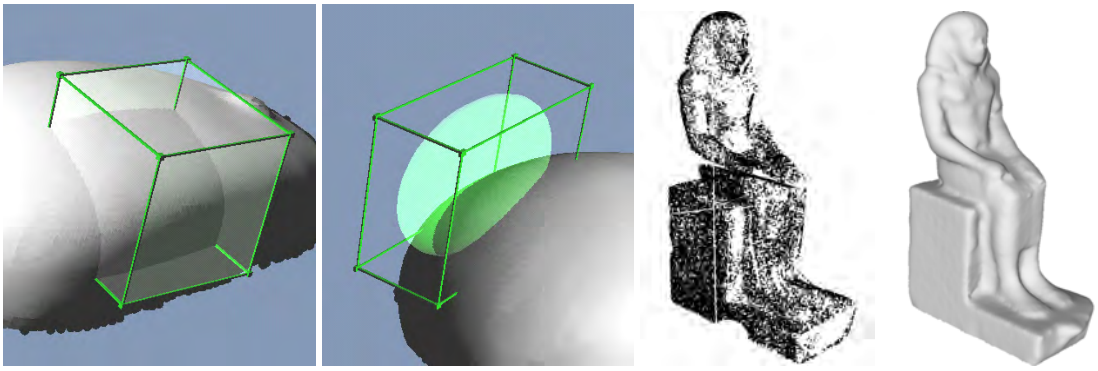
(a) Interactively textured sculpture with normal displaced text

(b) Textured sphere with carved footprints

Figure 2.25: Image (a) shows a point model of a face sculpture, interactively textured with the photograph of a face. The text is created with normal displaced points. Image (b) shows a point model of a sphere textured with a photograph of the moon. The footprints are carved with the help of a displacement map. Images from Zwicker et al. [115].

parametrization method and a dynamic and adaptive resampling method, which together allow for point cloud editing capabilities that are similar to an 2D image editing software, but with additional operations to alter the positions of the points. Image (a) of Figure 2.25 shows the point model of a face sculpture, texture mapped with the photograph of a face. The feature points of the sculpture and the photograph, e.g., the corners of the eyes or the mouth, were matched interactively, and the mapping of the photograph to the point model is done such that geometric distortions are minimized. The text is created with normal displacements, i.e., the positions of the points are changed along the directions of the normals according to some user defined function. Image (b) shows footprints carved into a spherical point model. The carving method also changes the positions of the points according to a user defined function, but along the normal of a plane that is positioned above the surface of the point model. Both of those geometry changing methods are not altering the general form of the point models, but only the shape of the surface. The proposed manipulations work only on a point model that exhibits almost no noise in the positions of the points, i.e., the distance of the points to the sampled surface is within a very tight error threshold.

The raw points clouds coming directly from scanners can exhibit several artifacts, causing the surfaces of the sampled objects to be quite noisy. Often points have to be deleted, such as outliers that were created due to some erroneous measurements, or a surface has to be estimated for the given samples, because they are distributed around the originally sampled surface. A



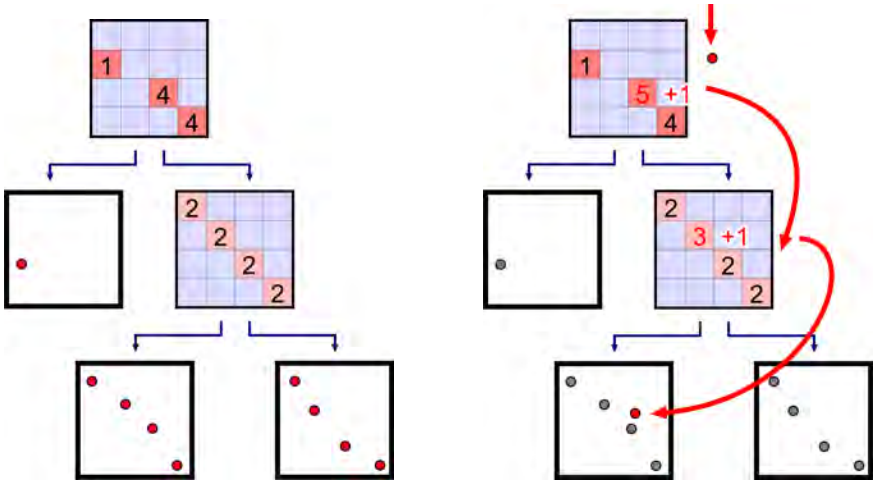
(a) Box selection brush (b) Ellipsoidal selection brush (c) Unevenly sampled (d) Volumetric diffusion

Figure 2.26: Image (a) shows a box selection brush following the surface of the object. Image (b) shows an ellipsoidal selection brush positioned freely in space. Image (c) shows an unevenly sampled point model with several holes. In Image (d), the holes were filled by volumetric diffusion. Images from Weyrich et al. [108].

system for such editing operations was proposed by Weyrich et al. [108] as plugin for Pointshop 3D. They developed a volumetric brush for selecting points (that are not part of a smoothly sampled surface), which allows to select all points inside its volume. It can be moved along the surface of objects by evaluating the depth values at the center of the brush in viewspace, or it can be moved freely in space. Image (a) of Figure 2.26 shows a box selection brush following the surface of the point model. Image (b) shows an ellipsoidal selection brush positioned freely in space. Image (c) shows an unevenly sampled point model, exhibiting large holes in the surface of the model. Image (d) shows the model after a volumetric diffusion has been applied to it, reconstructing the surface in undersampled areas. Volumetric diffusion works by placing the point model inside a volumetric grid, estimating a surface for the point model, and calculating a distance field for the surface inside the grid. In this distance field, holes are detected, which are then filled. Afterwards, the estimated surface is sampled to create a new point model, which is evenly sampled and exhibits no holes. Other tools, e.g., for deleting points or for automatic classification of outliers, were also developed.

Xu and Chen [112] proposed a point editing system for large point clouds sampled by a ground-based laser scanner. They segment the point clouds semi-automatically, to extract objects like buildings. The segmented objects are then organized in an object hierarchy and can then be edited individually. The user can edit the colors of the points, for example to adjust the colors of neighboring points that were colored by different photographs. The positions of the points cannot be changed by their editing system.

Boubekeur and Schlick [15] proposed a texturing system for large models which do not fit into the main memory of a computer. They convert the model to a point cloud that can then be textured in-core. The surface representation of the original model does not matter as long as it can be converted to a point model. The texture is applied interactively. If the sampling density of the point cloud is not sufficient for a texture, e.g., if a texture has high-frequency color changes, the sampling density of the point cloud can be increased, by locally resampling the original large



(a) Quantization tree with leaf nodes

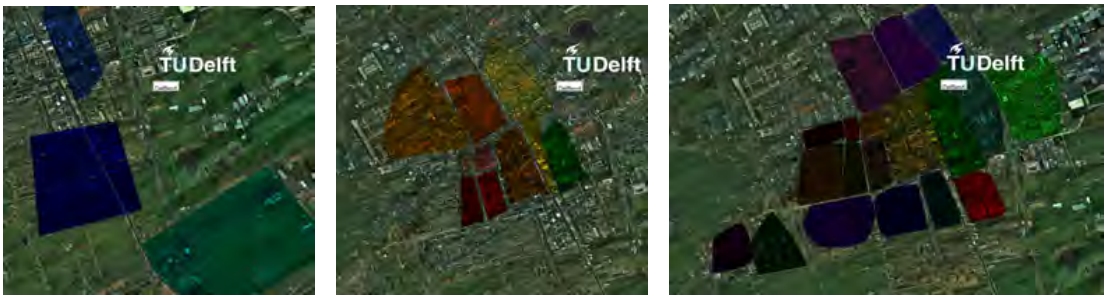
(b) Counters are updated for a new point

Figure 2.27: Image (a) shows a quantization tree, where the points are stored in the leaf nodes, and the inner nodes hold a grid, counting the points that are within their area. In Image (b) a point is added to the tree, so the counters are updated accordingly. Images from Wand et al. [107].

model. From the points, a 3D texture can be derived by upsampling the colored points, and this 3D texture can then be used to texture the original large model.

Levoy et al. [64] presented a set of software tools that were developed during their Digital Michelangelo project to manage large amounts of range images (i.e., images which store a depth value at each pixel) recorded by a triangulation laser scanner. During the project, famous statues created by Michelangelo were scanned, like the David statue or the St.Matthew statue (both are located today at the Galleria dell' Accademia in Florence). The aligning and merging of the range images produced by the scanner was done by a tool called Scanalyze [36], which can also handle large amounts of data out-of-core. When scanning the statues, planning the sequence of scan positions was done by visually examining the already recorded parts of the statues. For this, a point model was composed from all previously recorded range images, and missing parts were identified in the point model. To be able to render this point model, they choose to resample the range images at a resolution which allowed them to display the range images interactively. For this, they built range image pyramids, i.e., sequences of subsampled range images, where every image subsamples the previous one by a factor of 2 along the x and y axes, which are created on-the-fly when loading the range images from disk to memory.

Wand et al. [107] proposed a hierarchy that can be used to render huge point clouds out-of-core, but which can also be used to edit those point clouds. They build up an octree similar to a bucket PR octree, as all points of the original point set are stored at the leaf nodes. In contrast to a bucket PR octree, they store points in a quantization grid at the inner nodes of the octree, and each grid cell stores the number of points that fall within the volume of the cell. Figure 2.27 shows a quantization tree in 2D. In Image (a), the original points are in the leaf nodes, and the counters are in the quantization grid cells in the inner nodes. The counters refer to the number of points that are within their area in the leaf nodes. In Image (b), a new point is added, and while



(a) (b) (c)

Figure 2.28: A point cloud with geo-information integration, using (a) 3, (b) 10, and (c) 20 areas.

it is pushed down the hierarchy from the root node to the appropriate leaf node, the counters in the quantization grid cells are updated accordingly. The points at the leaf nodes are stored in an array, i.e., they are not sorted into a grid. The number of points in a leaf node is limited by a user defined threshold n_{max}, and common values for the threshold are about 100,000 points. In the interior nodes, each cell stores the copy of a single point from the original data set (for example the first point that falls into this cell when building the octree), and those points are then used to create the different LOD levels.

Storing original points in the inner nodes is similar to the idea of inner octrees in [110] where each octree node stores an octree itself, the so-called inner octree. When storing the points only at the lowest level of the inner octree, it would correspond to storing the points in a grid (see also Section 3.1).

The quantization grids allow for efficient updates of the stored point model, i.e., adding new points or deleting existing points, because the LOD levels can be updated easily, without having to perform complicated changes to the hierarchy (only the threshold for the maximum number of points in a leaf node has to be respected). A disadvantage of their implemented system is that they are reserving disk space for up to 3 times of the size of the actual point data, depending on n_{max}. Later on in Chapter 3, this approach will be compared to the new proposed point hierarchy, the MNO.

When applying editing operations to a point model, often a subset of the points is selected on which the editing operations are executed. When using point models that fit in-core, storing the information which points are selected is not a problem, as every point can simply store a flag which indicates if the point is selected or not. This would not work well in the case of out-of-core managed point clouds, as this would require to update all selected points in the external memory, which should be avoided if possible. Another idea is to store the information about the selection not directly at the points, but as a separate data structure that describes the volume of the selection.

Kehl et al. [54] propose a method to select points and change their attributes in large-scale geospatial lidar point clouds during rendering. These point clouds are 2D-like scans of large swaths of land, which are modified by intersecting them with polygons (lying parallel to the point clouds' main 2D plane) and changing their point attributes according to the information available in these polygons. This way, the color, visibility, or position of the points can be altered

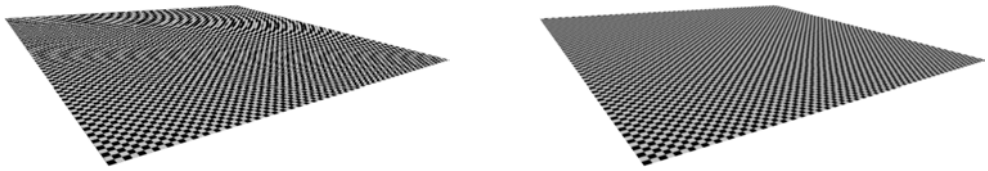
during rendering without pre-processing the points. The polygons are interactively defined by the user. They are stored in textures, so that points can check during rendering if they are inside a polygon. Since a point-in-polygon test is computationally more expensive than a point-in-triangle test, the polygons can be triangulated. During rendering, each point then has to check each triangle if it is inside or not. When there are many triangles which have to be checked, the performance will degrade. Therefore, the triangles can be sorted into a quadtree (which has a maximum depth of 5 levels), where they are stored in the leaf nodes. A point then has to find the leaf node which covers its position, and only has to test against the triangles that are inside this leaf node. Figure 2.28 shows a point cloud with geo-information integration of 3, 10, and 20 areas (i.e., polygons). The areas are subdivided into 20, 42, and 298 triangles respectively. Using 298 triangles for the geo-information cuts the rendering performance approximately by half compared to using no geo-information (and therefore no triangles for the intersection test).

For volume rendering, Bruckner and Gröller [16] proposed to use a selection volume that stores real values in the range [0..1], where 0 means "not selected" and 1 means "fully selected". Transformations can then be applied to the selection object, like moving, rotating, or scaling. Based on the selection, all voxels (volume elements) inside the selection can be rendered with a modified transfer function to emphasize them by a different color or opacity value. The idea of selection volumes was extended by Bürger et al. [17]. They proposed that for every selection volume, the user can also choose a custom higher resolution volume, which allows to support sub-voxel editing operations. The voxels in the higher resolution volume use upsampled information from the voxels of the original volume. Furthermore, the editing operations in this volume can be performed directly on the GPU.

In this thesis in Chapter 3, the selection octree is presented, which describes the volume of a selection, and so the points inside the selection need not be altered and written back to disk to store the information about the selection.

2.5 High Quality Point-Based Rendering

Point sampling the surface of an object means discretizing a (continuous) function, resulting in a point cloud representing the sampled surface. The point cloud usually has to be resampled when it is rendered, such that the points fit into the regular grid of the pixels in screen space (except for the case when the projected points fit exactly to the pixel positions). The problem with this approach is the occurence of artifacts due to the discretization of a continuous surface. It is known from signal processing theory that a continuous signal has to be sampled with twice the frequency of the maximum frequency that appears in the original signal, as otherwise the original signal cannot be reconstructed without errors. This Nyquist-Shannon sampling theorem has to be respected at two stages in the point sampling and rendering pipeline, first when recording the points, and second when rendering the points. At the first stage, this means that details which are not recorded during sampling cannot be reconstructed later on. In this thesis, it is assumed that the objects were sampled at a high enough rate to show all required details. At the second stage, this means that details can be lost during rendering. Therefore, the recorded details are tried to be preserved during rendering as good as possible.



(a) Signal not band-limited before sampling

(b) Signal is band-limited before sampling

Figure 2.29: Image (a) shows a checkerboard texture that is not band-limited before being sampled to pixels. In areas where a minification of the texture happens, severe aliasing artifacts occur. In Image (b), the texture is prefiltered before sampled to pixels, i.e., high frequencies are removed in the minification areas, which also removes the aliasing artifacts. Images from Botsch et al. [12].

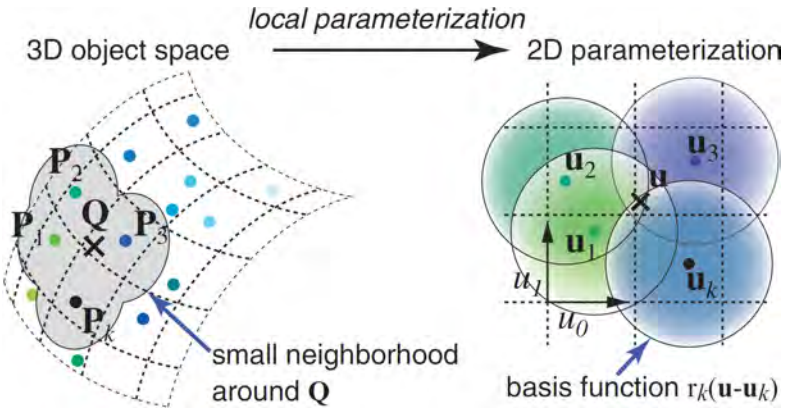


Figure 2.30: In a neighborhood around query point **Q**, a texture function is defined in 2D space using a local parametrization of the 3D surface. Image from Zwicker et al. [116].

In Figure 2.29, an example is shown, where the sampling frequency is too low compared to the frequency of the original signal (images from Botsch et al. [12]). In the example, a checkerboard texture is sampled to pixels, but in Image (a), in the minification area in the back, several checkerboard tiles are projected to a single pixel, resulting in aliasing artifacts. In Image (b), the high-frequency color changes of the texture are smoothed by resampling the texture, so the changes become more gradually. Therefore, when the pixels now sample the texture in the minification area, the visible aliasing artifacts are largely removed.

The principle of high-quality point rendering based on resampling texture functions was introduced by Zwicker et al. [116], by extending the general resampling framework for texture mapping, proposed by Heckbert [48], for point clouds. The idea is to associate every point of a point model with a radially symmetric basis function, project the basis functions of the points to screen space, and sum up the contributions of the basis functions at pixel positions. In Figure 2.30, the parametrization of a surface defined by basis functions, which are centered at the points, is shown. The points $\mathbf{Q}$ and $\mathbf{P}_k$ have 3D object space coordinates, which are transformed to 2D texture space coordinates $\mathbf{u}$ and $\mathbf{u}_k$. The continuous texture function f_c can then be evaluated at position $\mathbf{u}$ with

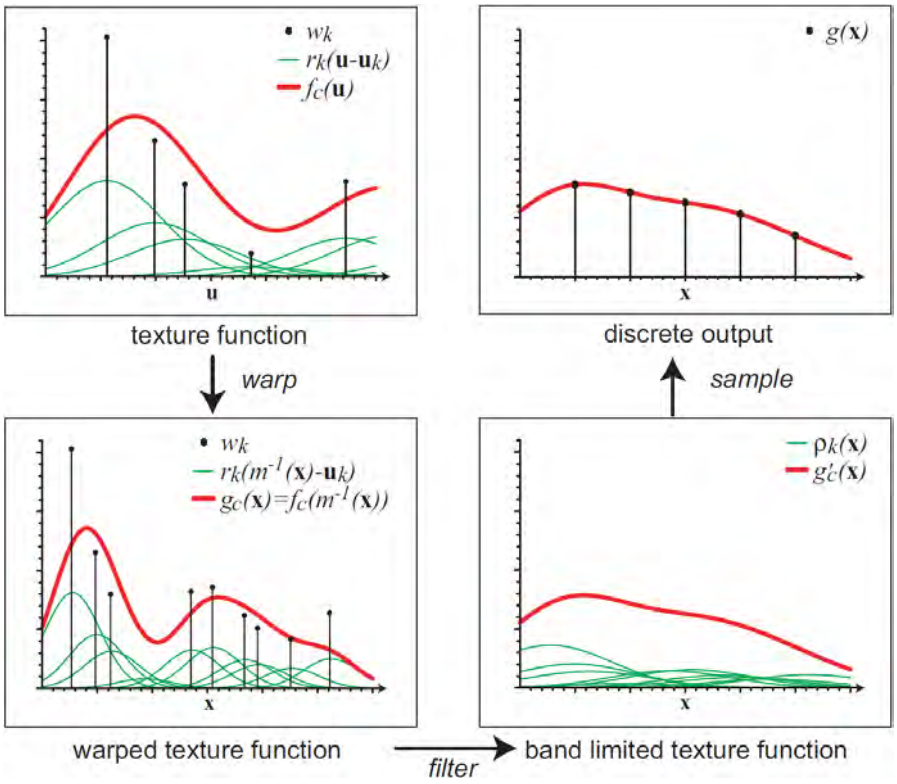


Figure 2.31: Resampling of the texture function f_c, defined by basis functions r_k, during rendering. The texture function is reconstructed in texture space from contributing basis functions, projected to screen space, band-limited to avoid aliasing, and then discretized to pixel positions. Image from Zwicker et al. [116].

$$f_c(\mathbf{u}) = \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{u} - \mathbf{u}_k) \tag{2.3}$$

where r_k is a basis function, and w_k a weight (in this case a color). Note that for every color channel of the texture a separate w_k exists, but, without loss of generality, in the example only a single w_k is used to describe the method). All basis functions have local support, so they only have to be evaluated up to a certain distance from their center, which also means that only the basis functions of points in the close neighborhood of the query point $\mathbf{Q}$ contribute to $f_c(\mathbf{u})$.

Heckbert introduced in his thesis also the elliptical weighted average (EWA) [42] filter for texture mapping, and this is used in the point rendering framework as well. In Figure 2.31, the different stages of mapping a texture from texture space to screen space are shown, when rendering a point model. In the upper left image, the continuous texture function f_c in texture space is shown. It is reconstructed as the sum of the contributing basis functions at position $\mathbf{u}$. It is then projected (or warped) from texture space to screen space by a mapping $\mathbf{x} = \mathbf{m}(\mathbf{u}) : \mathbb{R}^2 \rightarrow \mathbb{R}^2$, which results in the continuous screen-space signal

$$g_c(\mathbf{x}) = (f_c \circ \mathbf{m}^{-1})(\mathbf{x}) = f_c(\mathbf{x}^{-1}(\mathbf{x})) \tag{2.4}$$

where $\circ$ is a function concatenation. The result of this projection is depicted in the lower left image. In the following step, the screen-space signal is band-limited by applying a prefilter h, which results in the continuous output signal

$$g'_c(\mathbf{x}) = g_c(\mathbf{x}) \otimes h(\mathbf{x}) \tag{2.5}$$

where $\otimes$ stands for the convolution. The continuous output signal is then point sampled at the pixel positions by a multiplication with an impulse train $i(\mathbf{x})$, which creates the discrete output

$$g(\mathbf{x}) = g'_c(\mathbf{x})i(\mathbf{x}) \tag{2.6}$$

shown in the last image on the upper right. It can then be shown that the projection and the filtering of the basis functions can be done individually for each of the basis functions, and that the contributions of the basis functions can then be summed up in screen space. This method is called surface splatting. The basis functions are elliptical Gaussians, which cannot be transformed properly by a perspective projection, as the resulting basis functions would not be Gaussians anymore, making the band-limiting of those resulting functions complicated. Therefore, an affine approximation is used for the mapping $\mathbf{x}$, under which Gaussians are closed (i.e., they remain Gaussians after the projection). For the band-limiting low pass filter h also a Gaussian is used. This allows to express the complete resampling function for a point as a single Gaussian by combining the basis function and the low pass filter.

When rendering a point model with Gaussian resampling kernels, a splat is projected to screen space for each point, oriented according to the surface normal and large enough such that neighboring splats are overlapping without leaving holes in the surface. The contributions of all kernels within a certain depth range of the front most surface (as seen from the viewpoint) are evaluated at each pixel, normalized, and after projecting all points to screen space, a deferred shading pass is conducted. The shading calculates the final color for each pixel from the contributions of the projected points.

The surface splatting method was implemented in a software renderer, executing all calculations on CPU. In the following years, after the publication of Zwicker et al. in 2001, the GPUs of the graphics card became more flexible to use, and part of the rendering pipeline implemented on the GPU became programmable. Due to this development of the hardware, the surface splatting method could then be computed entirely on the GPU. This accelerated the rendering speed by a factor of two. Furthermore, it was discovered that the affine approximation of the perspective projection of the basis functions from texture space to screen space could produce splats where the outer contour was not correct in some situations. The center of the basis functions however was projected correctly. Therefore, the affine projection was adapted to produce accurate splat contours, but the center was not projected correctly anymore [117]. For a correct perspective projection (where the contour and the center of the basis functions are projected correctly) no affine approximation could be found. Nevertheless, a correct perspective projection can be found by looking at the pixels in viewspace, and casting a ray through each pixel which intersects the

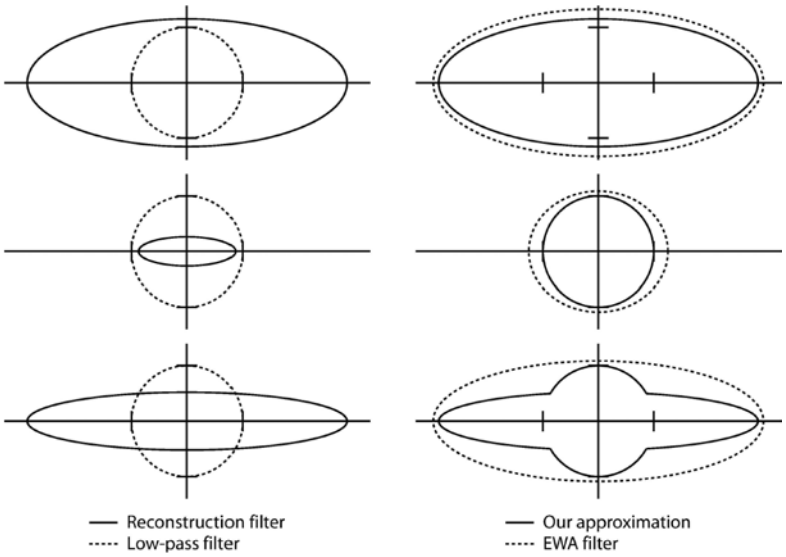


Figure 2.32: In the right column, a qualitative comparison of three typical situations of the EWA resampling filter with the approximation proposed by Botsch et al. [12] is shown. The approximation is composed of the shapes of the reconstruction filter projected from texture space and the low-pass screen space filter, as shown in the left column. Image from Botsch et al. [12].

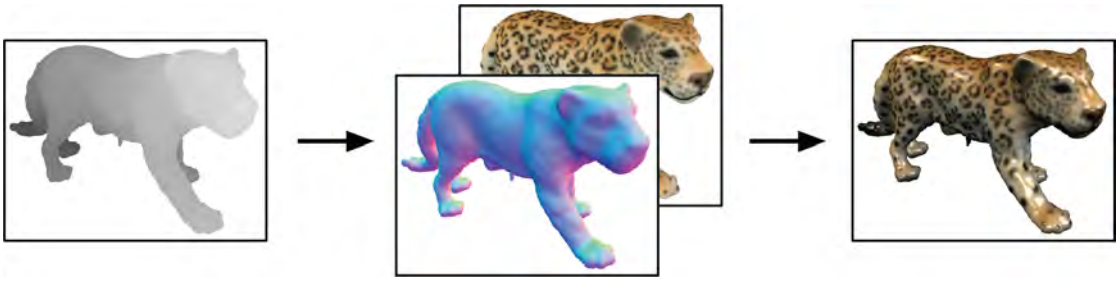


Figure 2.33: In this image, three stages of the point rendering pipeline proposed by Botsch et al. [12] are shown. First, the visibility splatting pass determines the depth up to which projected points will be considered in the next stage. Next, the attributes pass collects the contributions of the visible splats in different render targets. The final pass uses these information to shade the point model. This is a deferred shading approach, as the shading calculations are done in the fragment shader. Image from Botsch et al. [12].

object at some position. The parameters for the current basis function for this position can be derived. This way, it can be correctly decided if a pixel is part of a projected basis function or not [13]. This method is correct and also faster to evaluate compared to the affine approximation of the perspective projection.

In Botsch et al. [12], an approximation to the combined EWA resampling filter was proposed, where the reconstruction filter and the low-pass filter are evaluated separately. In the original EWA resampling approach by Zwicker et al. [116], those filters were combined to a single Gaus-

sian by convolution (see also Equation 2.5), which is computationally expensive to evaluate. By omitting the convolution, the compound shape of the separate filters is used to approximate the shape of the EWA filter, but it does not completely match. When using this approximation, it is necessary to render at least a 2x2 pixel sized splat to make anti-aliasing work. Because of this, more pixels are drawn than would be necessary, but the calculations can be kept simpler. This is a trade-off between the time needed to perform a convolution and drawing more pixels. For the current hardware, it is faster to draw more pixels. Figure 2.32 shows a qualitative comparison how the summed up reconstruction filter and low-pass filter appear in screen space (left column), and how their combined shape compares to the EWA resampling filter (right column).

These aforementioned changes to the original EWA surface splatting method, i.e., executing the complete rendering pipeline on the GPU, ray casting for the projection of the reconstruction kernel, and approximation of the EWA resampling filter by using a separate reconstruction kernel and a band-limiting kernel, led to an acceleration of the rendering speed by an order of magnitude [43]. In Figure 2.33, the GPU-based rendering pipeline proposed by Botsch et al. [12] is shown, which incorporates all above mentioned changes. It is a deferred shading pipeline, since all lighting calculations are done in the fragment shader. This way, the rendering of the geometry and the calculation of the shading can be clearly separated, which also leads to an easier implementation. The rendering is done in 3 passes, and the output of one pass is used as the input to the next pass. In the first pass, the points are rendered as splats into the depth buffer. This is done to determine the visible splats. In the next pass, the attributes from the visible splats are accumulated to different render targets (it is possible to output pixels into several targets). In the final pass, the information from the first two passes is used to normalize the contributions of the different splats to a single pixel, and to perform the shading of the model.

The high-quality rendering of point models allows for visually very pleasing results. A drawback of the method is the extra visibility pass, which requires to render the complete geometry, therefore reducing the possible performance by half. In this thesis in Chapter 4, a somewhat simplified version of this method will be used to render huge point clouds, where the samples exhibit a lot of noise.

2.6 Virtual Texturing

Texturing of polygon-based models is a standard technique to increase the quality of the visual appearance of the geometry by mapping an image or photograph onto the surface described by the polygons [18, 47]. A requirement for texturing is a mapping of the faces of the polygonal mesh to coordinates in the 2D image, and several well known methods exist to define a valid mapping. One example is “UV unwrapping”, where the mesh vertices are mapped into the texture image, such that the wasted space in the image due to the mapping is kept small, and that the distortions of the image, when mapped onto the mesh, are kept minimal. When using several images for texturing, it is advantageous to compile a texture atlas from those images, and use this atlas (which is a single image) as source for texturing. To use a texture during rendering it has to be “bound”, which is an operation that prepares the texture for usage on the graphics card, as the texture has to reside in graphics card memory to be used for texturing. Having only a single texture makes changing the texture binding, which is a costly operation performance

Figure 2.34: A downscaled image of a texture atlas consisting of 931 single images. The original texture atlas has a side length of 131,072 pixels.

wise [77], unnecessary. Figure 2.34 shows a texture atlas consisting of 931 images, as it is used in this thesis in Chapter 5 for texturing several mesh models, which were reconstructed from images taken inside a catacomb. The texture atlas has a side length of 131,072 pixels.

When dealing with large amounts of texture images, the main memory of the computer might become too small to hold all texture data, preventing online access to textures when needed for rendering. Several techniques were developed, which allow storing the complete texture data on disk and loading only the necessary data during rendering. The simplest approach is texture streaming, where single images used as textures are loaded on demand from disk.

Dividing textures into separate tiles increases the granularity at which texture streaming can happen, and reduces the amount of wasted graphics card memory, as during rendering not always a complete image is used for texturing. Typical tile sizes are 128x128 or 256x256 pixels. Note that the tile size for one texture atlas is fixed. Texture tiles were first used in terrain rendering [23]. For example, a single image contained in the texture atlas shown in Figure 2.34 would be split into approximately 256 tiles (when using tiles with 256x256 pixels size).

The method of using a tiled texture atlas was enhanced by “Clipmapping”, a technique for real-time terrain rendering, where the separate levels of a complete mipmap chain, including the original texture, are each tiled and stored on disk, and, depending on the current view position, the necessary tiles are streamed to memory [102]. A mipmap chain is a sequence of images, where each image is a subsampled version of the previous one in the sequence, with half the side length of the previous one. This way, the original image, which is the first in the sequence, is gradually down-scaled. Figure 2.35 shows the mipmap chain of the texture atlas of Figure

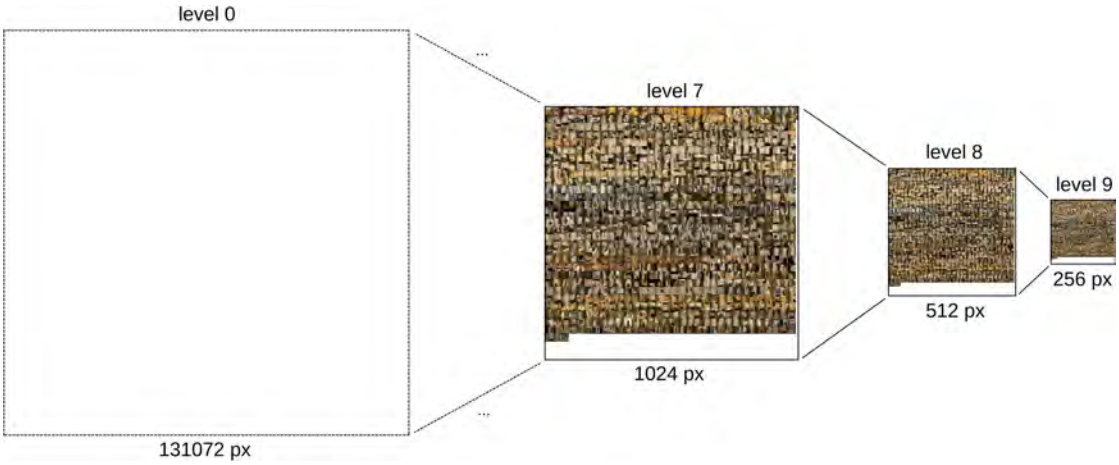


Figure 2.35: A mipmap chain consisting of 10 LOD levels. The original image at level 0 has a side length of 131,072 pixels. This image is subsampled, resulting in an image with half the side length at level 1. The image of level 1 is subsampled again, and after 9 steps the final image of level 9 is created from its preceding image.

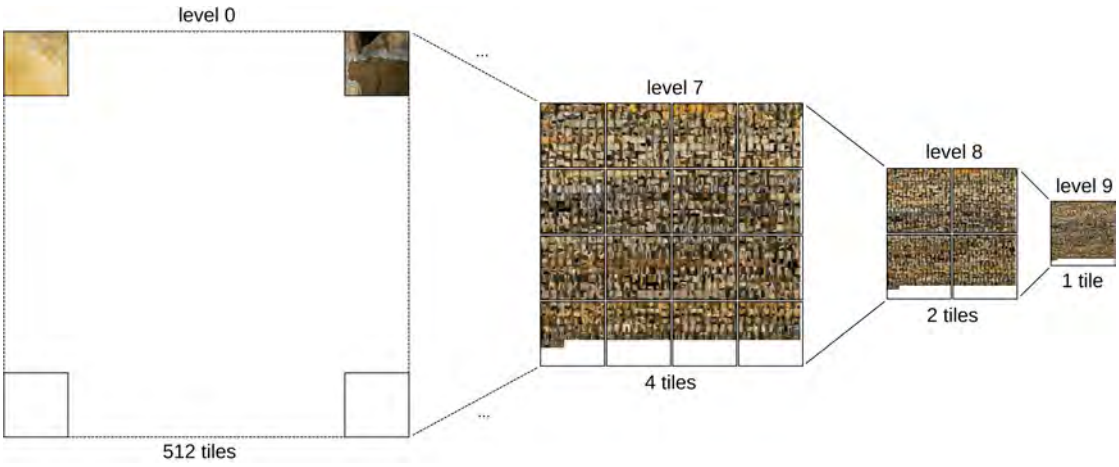


Figure 2.36: The images of the mipmap chain (see also Figure 2.35) are subdivided into tiles with 256 pixels side length. The original image at level 0 is divided into 512 tiles per side, level 1 into 256 tiles per side, and so on. Level 9 is represented by exactly one tile.

2.34. The original image at level 0 (not displayed as it is way too large) is subsampled in 9 steps. The final image at level 9 has a side length of 256 pixels. Figure 2.36 shows the images of the mipmap chain subdivided into tiles of 256 pixels side length. The tiles are arranged in a regular grid. The original image at level 0 is divided into a grid of 512 tiles per row and column. The size of a single tile matches the size of the image at level 9. The tiled images of the mipmap chain can then be used to speed up rendering and reduce aliasing artifacts during rendering.

Another approach for real-time terrain texturing is to separately calculate the necessary mipmap level (and therefore the necessary texture tile) for each polygon that will be rendered, and not rely on the mipmap level calculation of the graphics card. This is done by estimating the size of a rendered polygon in screen space [20].

Previous approaches are only suitable for texturing terrain that resembles approximately a 2D plane. Also, the tessellation of the geometry has to match the borders of the tiles used for subdividing the texture atlas. For generic geometry, another level of indirection when calculating the texture coordinates is necessary, the so-called "pagetable". With a pagetable, the texture management much resembles the way how virtual memory is handled in an operating system, therefore it is also called virtual texturing (although this term is not always used consistently in literature, and it is sometimes also relating to different methods). Virtual texturing allows to calculate the required mipmap level for the geometry per pixel, which was not feasible with previous methods (which calculate the mipmap level per geometric primitive). This can be done by rendering the geometry in a separate render pass into UV space [62] or into screen space [75], determine the required mipmap level and tile for each pixel, and afterwards stream the necessary tiles to the graphics card. Figure 2.37 shows the management of a virtual texture with the help of a pagetable. The pagetable itself is a mipmapped texture. The complete virtual texture is stored on external memory (e.g., on disk). Only the tiles that are required for the current view position have to be streamed to graphics card memory. In graphics card memory, the tiles are stored in the so-called physical texture, which is a large texture where the tiles are copied to, aligned according to the cells of a superimposed grid (each cell has the size of a tile). The page table texture also acts as a directory for the stored tiles, so it can be used to look up which tiles are on the graphics card and where they are stored in the physical texture.

Also the gaming industry makes use of virtual texturing, as mentioned by different game studios [52, 68, 75], to further increase the realism in state-of-the-art games.

Later in this thesis, a huge data set will be presented that is partly available as point cloud and partly available as polygonal meshes. All data is stored on external memory, and when rendering the data set, virtual texturing will be used to texture the polygonal meshes (see Chapter 5).

2.7 Navigating Cultural Heritage Data Sets

The documentation of archaeological monuments, findings, or cultural heritage sites by laser-scanning evolved rapidly over the last years, as scanning devices became more precise and are able to record more points in one pass [29, 90]. The resulting data sets from those scans might eventually become so large that finding places of interest inside the data sets becomes a problem per se. Therefore, while displaying the recorded data has become a challenge due to the size of the data sets, finding interesting places inside the data sets has become a challenge due to

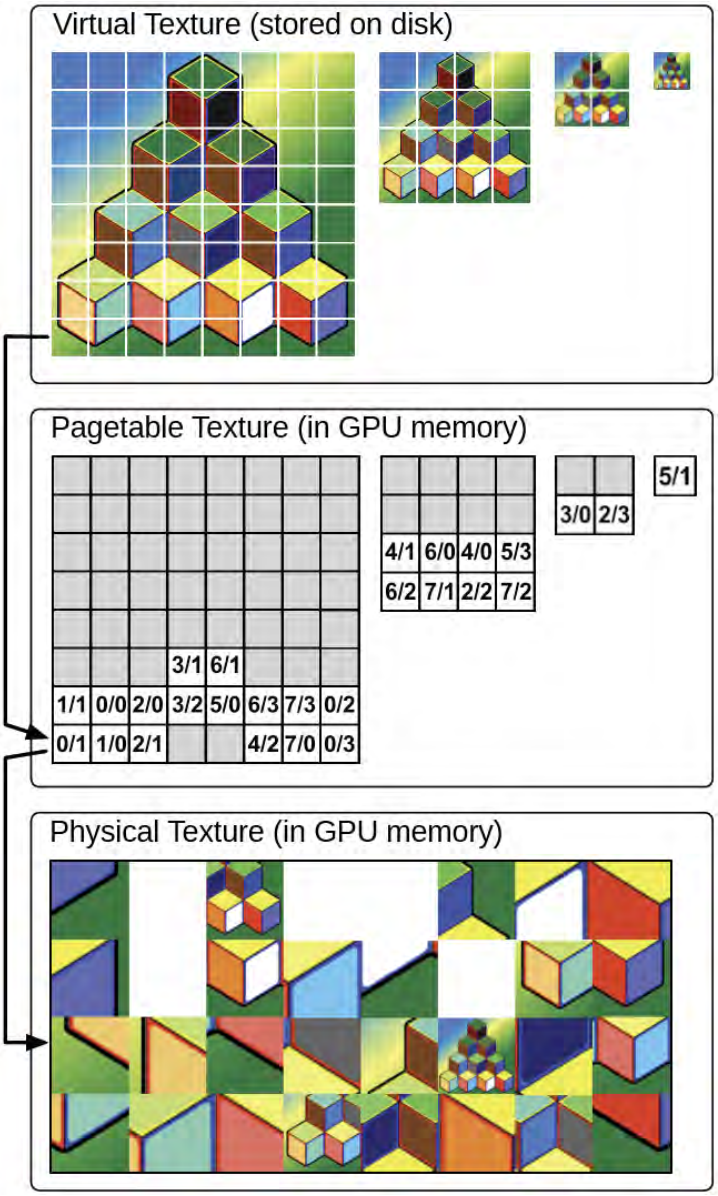


Figure 2.37: Management of a virtual texture with the help of a pagetable texture. In the topmost figure, the tiled images of a mipmap chain are shown (see also Figure 2.36). This tiled mipmap chain is referred to as virtual texture, and it is stored completely on external memory (e.g., disk). In the center figure, the pagetable is shown, which stores the positions of the tiles in the physical texture. The physical texture, shown in the lower figure, and the pagetable are two textures stored in the memory of a graphics card. The physical texture is a single large texture where the tiles are stored in the cells of a superimposed grid (the origin of the cells coordinate system is in the lower left corner of the physical texture).

the extent of the recorded objects. In this section, several large scan projects from the field of cultural heritage will be presented at first, while subsequently methods supporting navigation through large 3D models will be shown.

2.7.1 Scanning Projects

One of the first large scan projects was the Digital Michelangelo Project [64], where 10 statues created by Michelangelo were digitized. The statues were scanned by a triangulation scanner from roughly 100 scan positions for each statue. The range images output from the scanner were converted into triangle meshes and aligned to build a single watertight mesh for each statue. The largest single mesh, the statue of David, consists of $2*10^9$ triangles. For visualization, the models were resampled, and point-based models were created that were rendered with the QSplat point renderer [91] (see also Section 2.3.1). The largest presented point-based model, the statue of St.Matthew, consisted of 127 million points.

Duguet et al. [27] presented a multi-resolution data structure to render point-based models which fit into main memory. Instead of the original point positions they use the centers of the bounding boxes of the nodes of an octree to represent the scanned models. Their data structure was used for the visualization of a laser scanned model of the Ancient Greek "Dancers Column" in Delphi. To limit the approximation error of the nodes' centers to the original model surface, an octree of depth 13 was used, which allows to approximate the surface by about 2 millimeters (the total height of the column is 14 meters). The octree had about 90 million leaf nodes. The visualization was used to perform virtual anastylosis, i.e., the reconstruction of objects by assembling existing fragments based on a hypothesis describing the original shape of the model.

Beraldin et al. [10] scanned a large subterranean area, the Grotta dei Cervi, a Neolithic cave located in South-eastern Italy. The geometry of the cave was captured with a triangulation laser scanner, resulting in 716 range images and 630 million 3D points. The texture of the wall, exhibiting pictographs and petroglyphs, was captured with a digital camera, resulting in 3500 images. They mention that "Our major challenge is associated with the size and resolution of the 3D images which causes computer crashes and excessive processing time", requiring them to develop new tools that are capable of handling such huge amounts of data, but the development of the tools was considered to be future work.

Large cultural heritage sites, like medieval cities, can also be modeled by hand. The complete model of such a city, consisting of several polygonal objects, poses a problem to be rendered due to the high geometric details. Havemann et al. [46] discuss different LOD techniques and rendering acceleration techniques for such large polygon-based models. Amongst others, techniques like occlusion culling, imposters, terrain simplification, or instancing are suggested to enable interactive frame rates when exploring such models.

Boubekeur et al. [14] use a fast processing pipeline to convert large point-based surfaces into a coarse mesh, and recover the details of the point-based surfaces by calculating normal maps. The coarse mesh can be held in-core, as the geometric complexity is drastically reduced, and the appearance of the surface is stored in the normal maps. The simplification allows for fast rendering on today's graphics hardware.

When surveying a large cultural heritage site, it is often not necessary to sample the complete area with the same resolution, as not all areas exhibit the same amount of detail. For example,

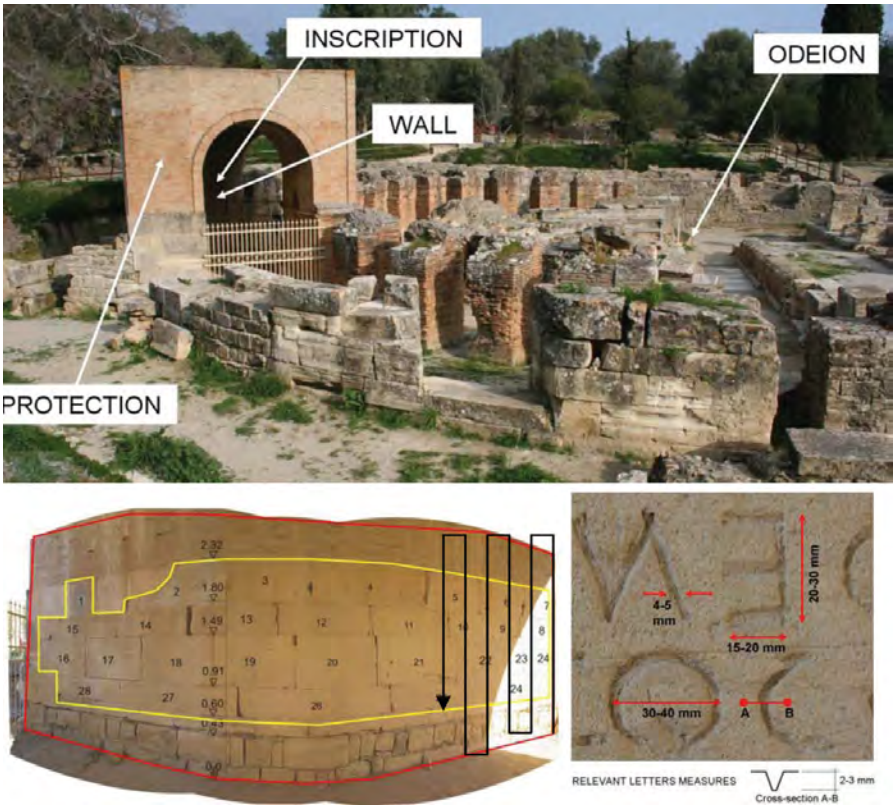


Figure 2.38: The area of the cultural heritage site of Gortyna, Crete, where the wall with the Great Inscription is located. The vaulted brick roof was built in modern times to protect the wall and the engravings of the Great Inscription. The lower left image shows the (numbered) blocks from which the wall is built, and the direction of scanning (black arrow). In the lower right image, the dimensions of the engraved letters are shown. Image from Remondino et al. [88].

simple floors or walls can be recorded with a sparser sampling resolution than wall paintings or pottery. Depending on the situation, also different recording techniques can be used. For example, laser scanning can be used for fast recording of the geometry and passive stereo scanning for recording wall paintings (see also Section 2.1 and Chapter 5).

Guidi et al. [45] presented an approach for the integration of multi-sensor, multi-resolution data, where the archaeological site of the Roman Forum of Pompeii can be explored in resolutions from aerial photographs to very fine details. The high-resolution models are recorded with terrestrial laser scanners for geometric data, and the texture of the objects is recorded with digital cameras. The resulting data (the laser scanners gathered $1.2*10^9$ points), was then converted to polygonal mesh models. Pignatel et al. [86] used the created mesh models to render the Forum of Pompeii with a professional real-time rendering software. They encountered problems with the size of the textures, as the rendering software required the texture data to fit completely into the graphics card memory. Due to this limitation, the texture sizes had to be reduced for most textured objects.

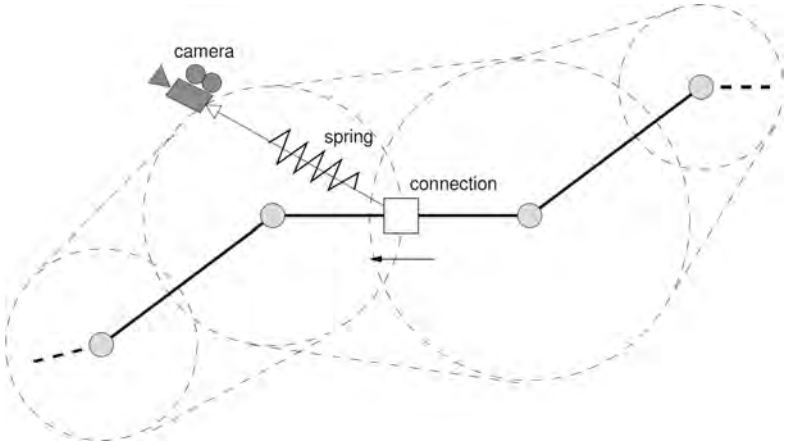


Figure 2.39: The camera is connected to the camera path by a spring-like cord, allowing the user to smoothly move the camera within the area bounded by the outer dashed line, up to the full extent of the cord. Each vertex of the camera path is assigned a circular area where the camera can move into, and the areas of neighboring vertices are connected by lines between their outermost points. Image from Elmqvist et al. [28].

Remondino et al. [88] sampled the Great Inscription of Gortyna, Crete. This inscription is a text engraved into a limestone wall. The wall has a size of about 6 by 1.75 meters. The letters are roughly 30 millimeters in height and engraved 2-3 millimeters deep into the wall. Figure 2.38 shows the area of the cultural heritage site Gortyna, where the wall with the inscription is placed. The lower left image shows the wall and the direction of scanning, while the lower right image shows the dimensions of the engraved letters. The inscription was scanned with a triangulation-based scanner (resulting in about 460 million points), while the surroundings were sampled by a time-of-flight laser scanner (resulting in about 60 million points). The scans were then registered and converted into a mesh, which makes the inscription available to a broader audience (normally access to the inscription is only allowed up to a distance of 4 meters), and saves the current state of the inscription for future research, since the limestone wall is in danger to collapse.

In this thesis, in contrast to previous approaches, methods will be presented, which do not require to reduce the point-cloud data or texture data, and which use the original data during interactive visualization. This is possible by using out-of-core rendering algorithms, and only loading data relevant for the current viewpoint to graphics memory (see also Chapters 3 and5).

2.7.2 Navigation Support

Supporting the user when navigating a large data set becomes essential, if the landmarks in the data set are sparsely distributed, or if the user shall be pointed to especially interesting areas. One possibility to guide a user through a data set is with the help of pre-defined camera paths. This confines the position, direction, and moving speed of the virtual camera to the parameters stored in the camera path, impeding interactive changes to the camera's position and orientation.

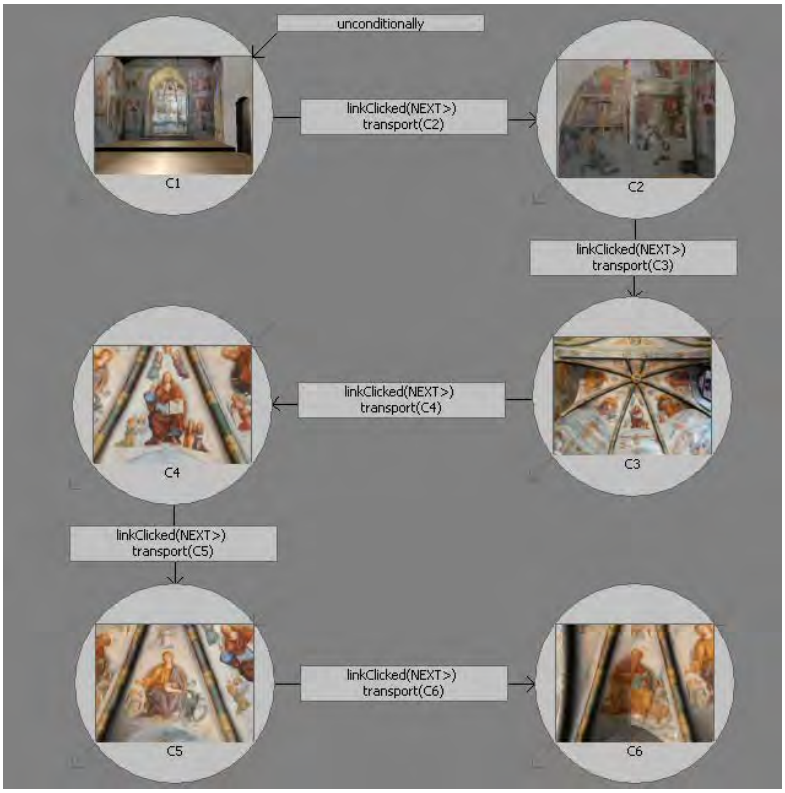


Figure 2.40: The user interface to define a tour. The different views of interest are connected by edges, which represent the transitions between subsequent tour stops. The transitions are defined manually by the curator. Image from Chittaro et al. [67].

Elmqvist et al. [28] presented a 3D navigation guidance, where the camera still moves along a (manually or automatically created) camera path, but where it is not tightly bound to this pre-defined path. The user can control the movement speed, the viewing direction, and also to some degree the positioning of the camera, because it is possible to leave the camera path up to a certain distance. This way, the user can explore the data set more interactively, while still being guided along an interesting path. Figure 2.39 shows a camera path and the surrounding area, where the camera is allowed to move. The camera is attached to the path by a spring-like cord, which allows the user to move the camera within the area bounded by the dashed line, up to the full extent of the cord.

Chittaro et al. [67] presented a supporting software tool for curators to set up virtual exhibitions. With this tool, 3D models can be placed into virtual rooms, views of interest for each model can be defined, and tours can be composed, where a pre-defined set of models can be visited (which are connected for example by some mutual topic). The transitions between the tour stops can be performed in different ways, either by teleporting (i.e., jumping immediately to the next), by automatically moving the camera (similar to following a camera path, i.e., without user interaction), or by drawing a path from the current to the next stop, and the user has to

navigate through the environment manually to reach it. Figure 2.40 shows the user interface of the software tool, where the tour stops and the transitions between the tour stops can be defined.

Later in this thesis, a navigation system for large 3D models will be presented, which allows the user to choose between different pre-defined paths interactively, so he does not lose orientation inside a model (see also Chapter 6).

2.8 Summary

Points as rendering primitives have gained a lot of attention in the last years. The reason for this is the wide range of applications for which points can be used. Although they are the most simple rendering primitive, they can be used to describe any mathematically smooth surface correctly.

Point clouds can be acquired from different sources. Examples are point samples taken from real-world objects by scanning devices, point samples taken from mathematically defined objects, or point samples taken from objects described by other geometrical primitives. Point clouds can be rendered very fast, if a low visual quality is acceptable, or with high visual quality, when using computationally more expensive algorithms. A high visual quality can by achieved by reconstructing the surfaces of the sampled objects, and resampling the points according to the resolution of the display where the points are rendered to. Also data structures for organizing huge point clouds out-of-core have been proposed, allowing to render point clouds whose size is only limited by the amount of external memory. But not only rendering, also editing of point clouds has been researched, and also the tools that can be used to edit them efficiently. The selection of points in huge point clouds was also investigated.

While point clouds are well suited to describe geometrically complex surfaces, other rendering primitives have advantages when describing geometrically less complex surfaces. The complexity of surfaces can not only come from the geometry, but also from the texture. For detailed non-repeating textures, virtual texturing has been developed, which allows to manage large textures out-of-core.

Finally, for the navigation of large data sets in the context of virtual walkthroughs, several approaches were proposed, which restrict the possible directions a user can follow, but still allow interactive exploration to some degree. One way to do this is by binding the user to a pre-defined path, which can only be left up to a certain distance. Another way to do this is by pre-defining the possible target areas, and restricting the movement to the paths connecting the target areas.

CHAPTER 3

Point Cloud Visualization and Editing

In this chapter, the modifiable nested octree (MNO) data structure is introduced, a hierarchical data structure that can be used to manage huge point clouds out-of-core. The MNO is an enhancement of the nested octree data structure, presented by Wimmer and Scheiblauer [110], allowing for efficient editing operations on the points stored inside the data structure while still enabling fast rendering of the points. Fast rendering is important for interactive navigation through the point data. Additionally to introducing the MNO, a data structure for selecting points is presented, the so-called selection octree, which can also be used together with out-of-core point rendering systems. A selection octree describes the space inside which points are being selected, so the point data itself is not changed when selecting points. Besides introducing enhancements for editing huge point clouds, also a point-size heuristic is presented, which tries to fill gaps in surfaces that are not sampled densely enough. Furthermore, an analysis of the time and space requirements of some typical operations performed on the points stored in an MNO is presented, which also compares the performance of these operations to previous work.

In the following, a recap of the nested octree data structure will be given first, then the MNO data structure will be presented, pointing out the changes made to the nested octree data structure. Afterwards, the selection octree data structure and the point-size heuristics will be introduced, and how they are used with the MNO. Following this is a complexity analysis of editing operations conducted on the points in an MNO. The complexity of the operations are then compared to the complexity of the operations on a hierarchical data structure presented by Wand et al. [107], which in this thesis, for the sake of convenience, will be referred to as Michael Wand octree (MWO). An enhancement to the deleting operation of points from the MWO will also be introduced. This enhancement makes it possible to use more efficient hierarchy traversals when rendering points stored in the MWO. Finally, benchmarks for the editing operations (like adding points to or deleting points from the presented data structures) and for different hierarchy traversals, which are executed during rendering, will be given.

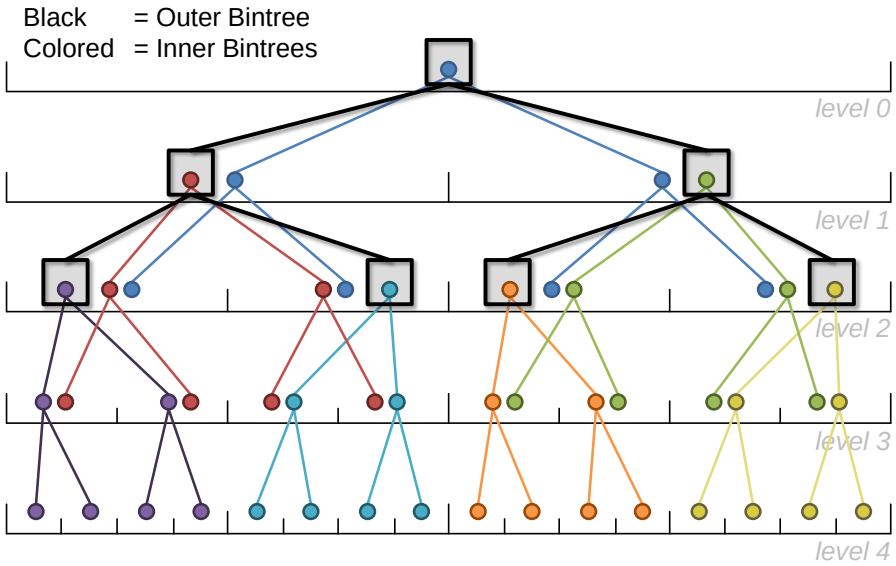


Figure 3.1: The hierarchy of a completely filled nested bintree (which is a nested octree in 1D) of depth 5. All inner bintrees and the outer bintree have a depth of 3. The root nodes of the inner bintrees have the same bounding box as the nodes of the outer bintree at the same position. The points reside at all levels of the inner bintrees.

3.1 Nested Octree Recap

The nested octree data structure was introduced by Wimmer and Scheiblauer [110]. It is a hierarchy to manage huge point clouds out-of-core such that they can be rendered and explored interactively. To make this possible, the data structure has to fulfill certain criteria. For supporting out-of-core and real-time rendering, it has to support LODs, because when rendering a huge point cloud, not all points can be loaded into memory and displayed at once. Another aspect to consider is the representation of the LODs in the hierarchy. When dealing with huge data sets, additional data to represent LODs will add significantly to the amount of data that has to be stored on disk. Therefore, a compact representation of the LODs is favorable, i.e., a representation that does not need additionally created points. Furthermore, the build-up process and the data structure should support unevenly sampled point clouds, so the original point data sets do not have to be resampled when building the hierarchy.

The nested octree does this by subsampling a point cloud and storing sets of points at the octree nodes in a way, such that the union of all point sets stored in the octree again results in the original point cloud. Additionally, the union of the point sets from level 0 (i.e., where the root node lies) to a level k shall result in the LOD level k. This means, the point sets of a level further down the hierarchy refine the point cloud represented by the union of the point sets of all levels above. During rendering, the number of points that are projected to the same pixel is reduced by using the LODs, and from those points only one representative point is required to fill the pixel.

3.1.1 Hierarchy Layout

The hierarchy layout of a nested octree can be subdivided into two parts, the so-called outer octree, which defines the traversal hierarchy, and the so-called inner octrees, which are octrees at the nodes of the outer octree. The root node of an inner octree has the same bounding box (orientation, size, and position) as the outer octree node it is inscribed to. The maximum depth of the inner octrees is bounded (e.g., to 8 levels). Points that are inserted into the nested octree are actually inserted into the inner octrees. Each node of the outer octree holds one inner octree, and each node of an inner octree can hold at most one point. The point cloud is therefore subsampled by inserting the points into the inner octrees. The outer octree and inner octrees regularly subdivide the space at each level and store several points per node, similar to the bucket PR octree described above, so the point cloud will be uniformly subsampled. In contrast to the bucket PR octree, points are stored at all nodes, not just at the leaf nodes.

Figure 3.1 shows an example of the nested octree data structure in 1D, for the sake of convenience, so it is a nested bintree. The outer bintree is drawn with black edges and square nodes. Each outer octree node holds the root node of a colored inner bintree. In this example, the inner bintrees are fully occupied, i.e., all levels of the inner bintrees hold points. The outer bintree and the inner bintrees share the same space. Notably, as can be seen for example at level 2, points from several different inner bintrees (in this example up to 3) can exist within the bounding area of a single bin.

3.1.2 Build-Up

The points of the original (unorganized) point data set are inserted one-by-one into the nested octree. In the beginning, only the root node exists, which is large enough to encompass all points that will be filled into the hierarchy (the exact bounding box of the original point data set is either known, or has to be calculated in a previous step by traversing all points once, and it is then transformed to a box, which also encompasses all points). First, the cells of the upper levels of the inner octree at the outer octree root node are filled, until subsequent points finally fall into the cells of the lowest available inner octree level. If a point is inserted into an inner octree and falls into a cell that has already been occupied, than the point is filtered down to the next lower level, until a free cell is found. If no free cell can be found, the point is rejected from this outer octree node and pushed down one outer octree level (to the appropriate child node of the outer octree node), where it is tested if the point can be inserted there. This procedure is recursive, until a free cell in an inner octree can be found, where the point is then stored.

When saving the points of an inner octree to disk, the inner octree is linearized in level order, i.e., all points from level 0 are stored first (in case of level 0 this is only 1 point), then all points from level 1 are stored, and so on. This way, the points can be stored in an array, and the array of one inner octree is stored in one file. When the points of an inner octree are required during rendering, the points stored in the file are loaded from disk and put directly into a vertex buffer object (VBO). There is no need to rebuild the inner octree hierarchy.

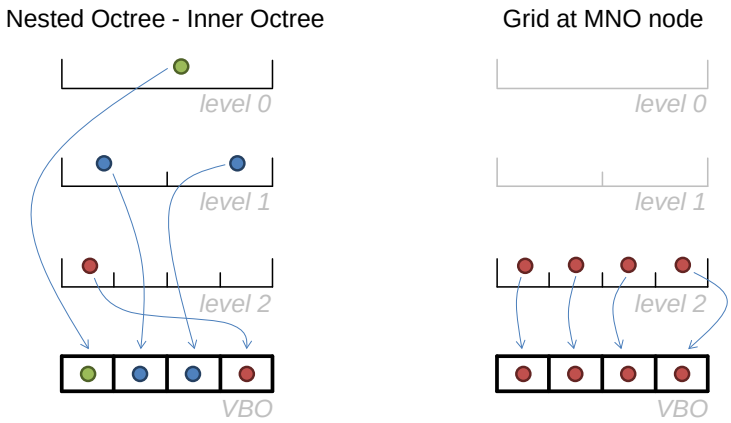


Figure 3.2: The difference between a nested octree and a modifiable nested octree is in the storage of the points (in 1D for the sake of convenience). The nested octree stores the points in its nodes in inner octrees (left), while the MNO stores the points in its nodes in grids (right). Inserting and deleting points are costly operations for an inner octree, because the octree has to be rebuilt. An inner octree also has to maintain a certain order when mapping the points to a VBO, while a grid can map the points sequentially to a VBO.

3.1.3 Inner Octrees LODs

While the inner octrees represent point subsets of the original point cloud, they contain themselves also a LOD hierarchy. This can be seen in Figure 3.1 (in the 1D case), where the inner bintrees hold points at all their hierarchy levels. The LODs of the inner bintrees are created by adding up the points from the hierarchy levels, beginning at the root node. This is similar to creating the LODs of the nested octree, but instead of adding up inner octrees, only the points inside the inner octree levels are added up.

The LODs of an inner octree are actually only advantageous at the root node of the outer octree, when the viewpoint is at a great distance to the nested octree. When the viewpoint is so far away that the child nodes of the outer octree root node are not used anymore, the LODs of the inner octree at the root node can be used to further reduce the number of rendered points, until only one point from the coarsest LOD is rendered. On the other hand, when the viewpoint comes closer to the nested octree, and the points of the inner octrees are added up to create the appropriate LOD, then actually more points then necessary are rendered. This can also be seen in Figure 3.1, where actually a single point in each bucket would suffice for a valid nested octree LOD representation. The LODs of the inner octrees are of no advantage in this case, as the inner octrees are immediately rendered at the finest LOD when they are chosen for rendering.

3.1.4 Rendering

When rendering a nested octree, the rendering algorithm traverses the outer octree from the root node to find the nodes whose projected size in screen space is large enough and therefore shall be rendered. For this purpose, the bounding box of the front-most cell of the lowest inner

octree level of a node, with respect to the current viewpoint, is projected to screen space onto the near plane. If its size (e.g., the projected diameter of the cell) is above a predefined threshold (e.g., 1 pixel) the node is inserted into a priority queue. The priority queue is ordered by a calculated importance of the nodes. The most important nodes are the ones which are (partly) within the view frustum, close to the viewpoint, and in the center of the screen. There also exists a threshold on the number of points that can be rendered in a single frame. Only the nodes with the highest importance values are chosen for rendering, up until the budget for the maximum number of rendered points is filled, or until no more nodes are large enough when projected to screen space.

After the nodes eligible for rendering have been determined, the rendering of the points starts with the most important node. When rendering a node, all points of the node are rendered, and they are rendered at the same size. The size of the points can be chosen in different ways, for example as fixed point size for all rendered points of all nodes. Another option for choosing the size is to use the point-size heuristic described below (see Section 3.6).

Screen-aligned square splats are used as geometric primitives, as they are the fastest primitives to render. The points of the nodes that are chosen for rendering but are currently not in memory are swapped to memory in a separate thread. This is done by sending a request to the reading thread to read the files containing the points of the nodes. The reading thread then passes back a memory buffer with the loaded points, which can then be moved directly to the GPU for rendering. Points that are not used for rendering any more are dropped from the GPU and stored in a least-recently-used (LRU) cache [25] in main memory. Only when points fall out of the LRU cache, they have to be loaded again from disk if needed.

3.2 Modifiable Nested Octree

The nested octree is an efficient data structure for rendering points out-of-core, but it has deficiencies when editing points. This is due to the inner octrees, which have to be rebuilt every time when new points are added to or existing points are deleted from the hierarchy. Also, after editing the points in the inner octrees, the hierarchy has to be sequentialized before it can be written to disk. Furthermore, as mentioned above, the points stored in the upper levels of the inner octrees create some overdraw during rendering.

To accommodate for those deficiencies, the inner octrees can be replaced by a regular grid, which makes the maintenance of the inner octree hierarchies unnecessary. Using a regular grid corresponds to removing the upper levels of the inner octrees and only retaining their leaf nodes. To express the improved ability for conducting changes to the points stored in the hierarchy, the new hierarchy will be called modifiable nested octree (MNO) in the following. Figure 3.2 shows the difference in how the points are stored in a nested octree and an MNO respectively. A node of a nested octree holds an inner octree, which in turn holds the points. Converting an inner octree to an array (either for storing it on disk or in a VBO) requires re-ordering the points. Instead, an MNO node holds the points in a grid, and the order in which the points are stored in the array does not matter, because the positions of the points in the grid can be directly derived from the 3D positions of the points (if the points have to be sorted into the grid at a later time).

With these changes, the LOD hierarchy now works similar to the LOD hierarchy of the LPC by Gobbetti and Marton [38]. They uniformly subsample an input point cloud at each node (using a kd-tree) to choose the points for this node. The difference of the MNO hierarchy to the LPC hierarchy is the usage of a regular grid at each node of the hierarchy instead of an array of uniformly subsampled points. Since for an MNO node it is not necessary to uniformly subsample the points, inserting and deleting points becomes easier, so the MNO is better suited for editing operations on points in the hierarchy.

There are also drawbacks when using an MNO for processing points. Due to the nested representation of the LODs, a neighbor query, for example, has to search nodes at all hierarchy levels, because the potential neighbors of a point can be at any hierarchy level. The most efficient way to do a neighbor query is to first determine a maximum radius in which neighbors shall be searched, and then collect all points from all hierarchy levels that are within this radius. This is not necessary in an octree where the points are only stored at the leaf nodes. A complexity analysis for in-core and out-of-core processing tasks is given later in Section 3.9.

3.2.1 Inserting Points

Inserting points into a new MNO is very simple compared to a nested octree. The points are inserted one-by-one, starting at the root node of the MNO. For each point, an empty cell in the root node's regular grid is searched. The 3D position of the point is used to determine the cell into which the point would fall. This is done by subdividing the node's bounding box to the resolution of the regular grid, and calculating the cell index according to the 3D position of the point. If the point falls into an empty grid cell, it is stored there. If it falls into a grid cell that has already been occupied, the point is pushed down to the appropriate child node of the root node. This is done by deriving the so-called child node index from the point coordinates and the center of the root node's bounding box. The child node index is the index of the octant of the root node (with the root node's bounding box center as the origin) the point falls into. Inserting the point into an MNO is done recursively, so it is tested if the point falls into an empty grid cell in nodes further down the hierarchy, until an empty grid cell is found.

Each node of an MNO is required to have more than m_{min} points. To make sure this condition is fulfilled, two measures are taken. First, points can be stored at a node even if they are not stored in a grid cell (e.g., in a separate array), but only if the number of points in a node is still below m_{min}, and second, the number of points that will be pushed to a new child node has to be at least m_{min}. This is done, because it is inefficient to store nodes at disk that hold too few points.

In the implementation of the MNO used for this thesis, the grid is represented as a hash table. To check if a position in the grid is still free, the hash table key for the according cell is calculated. The grid is subdivided into 2^7 cells along each of the 3 space axis, so up to $(2^7)^3 = 128^3$ points can be stored at one node. For best performance, a speed-optimized hash map implementation from Google [39] is used. Algorithm 3.1 shows the algorithm for inserting points into an MNO.

```
1  foreach point P in source point data do                    // Main loop
2  |  root node R->insert(P)
3  end
4  method node::insert(P):                                    // Method for inserting a point
5  |  calculate the grid cell GC containing P
6  |  if GC is empty then
7  |  |  store P in GC
8  |  |  if (A_PP not empty) && (num points in A_PP + num points in grid > m_min) then
9  |  |  |  get surplus point Q_s from A_PP
10 |  |  |  if child node C_k1 for Q_s exists then
11 |  |  |  |  C_k1->insert(Q_s)
12 |  |  |  else
13 |  |  |  |  store Q_s in array of points A_k1 for child C_k1
14 |  |  |  end
15 |  |  |  delete Q_s from A_PP
16 |  |  end
17 |  else
18 |  |  if num points in grid < m_min then
19 |  |  |  store P in A_PP
20 |  |  else
21 |  |  |  if child node C_k2 for P exists then
22 |  |  |  |  C_k2->insert(P)
23 |  |  |  else
24 |  |  |  |  store P in array of points A_k2 for child C_k2
25 |  |  |  end
26 |  |  end
27 |  end
28 |  if num points in any array of points A_k for child C_k ≥ m_min then
29 |  |  create C_k
30 |  |  foreach point Q_i in A_k do
31 |  |  |  C_k->insert(Q_i)
32 |  |  end
33 |  |  delete A_k
34 |  end
35 end
```

Algorithm 3.1: Inserting points into an MNO. A_{PP} is the array of padding points, which holds the points that did not find an empty grid cell. Those points are used to increase the number of points at the node. Only when the number of points at the node goes beyond the minimum threshold m_{min}, points are removed from A_{PP}, and sorted into the according child nodes.

3.2.2 Build-Up

The build-up process starts by reading point data files, and converting them into a canonical binary data stream to get a uniform representation of all points, independent of the data format they were originally stored in. When reading the point data files, the bounding box of the points contained in the files is calculated. The binary data stream stores the points intertwined, i.e., a point with all its attributes is stored as a continuous array of bytes. Examples for attributes are position, color, normal, or scan position index.

Note that many out-of-core algorithms rely on a first pass to create locally coherent work items (or buckets) that can be treated in-core and that are limited by user-defined boundary values (e.g., distribution sort [58]). However, in the case of point clouds, it is difficult to find sensible boundary values for these buckets, because the density of the points can widely vary. Therefore, it is difficult to know in advance how many buckets are sufficient to achieve a subdivision such that each bucket can be treated in-core.

After creating the binary data stream, the points are read in batches of several 1000 points from this stream and inserted according to the insertion algorithm (see Section 3.2.1). The basic insertion algorithm only works for models that completely fit into main memory. To make it work for larger models, the nodes of the MNO can be managed in an LRU cache. The points of the nodes are swapped in and out of memory according to the requirements of the insertion algorithm. The LRU swapping algorithm uses the main memory as cache for the nodes of the MNO, so if a point model fits completely in main memory, the build-up algorithm is as fast as the basic insertion algorithm.

The points are stored in one file per node on disk (and the nodes of each hierarchy level can be stored in a distinct directory). The access to the files is managed by the operating system's file system, e.g., on the test computers used for this thesis, Microsoft Windows is used as operating system, therefore the files are managed by the Windows NTFS file system (see also Section 2.2.3.1).

This out-of-core algorithm works especially well if there is spatial coherence among points, which is often the case in scanned models. The original build-up algorithm for the nested octree [110] swaps points to disk every three levels they are filtered down the octree hierarchy. This is reasonable, when the amount of main memory is no more than 512MB, but today the main memory on high-end PCs is often 4GB or more, which thus can be used as cache during build up. The number of points that can be held in-core depends on the attributes per point and on the size of the LRU cache. Assuming that one point has the attributes position and color represented in 16 bytes and that the LRU cache has a size of 10GB, then a point cloud with some 625 million points can be built in-core.

If there is no spatial coherence in the original point data set, i.e., the points are more or less randomly distributed in space, then an external memory sorting algorithm can be employed before building up the MNO. In Section 3.11.1.3, results are discussed, where an MNO is built up once with and once without sorting the points beforehand. The general outcome is that large and densely sampled point data sets benefit a lot from sorting them before inserting them into an MNO.

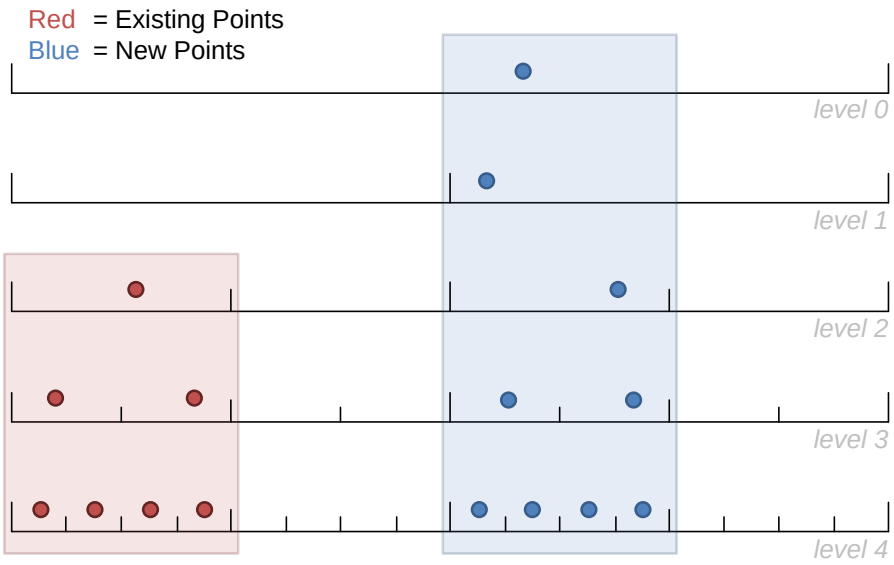


Figure 3.3: Schematic illustration of the LOD hierarchy of an MNO after inserting new points. The red points are part of the old MNO (inside the red bounding box), and the blue points (inside the blue bounding box) are newly inserted. The bounding box of the new points does not intersect with the bounding box of the old MNO, so in level 1, the LOD hierarchy from the new root node to the old root node is broken (there should be a point in the left bin of level 1).

3.2.3 Modification Operations

In this section, the modification operations on the MNO are described, which are adding new points to and deleting points from an existing MNO.

3.2.3.1 Adding Points

When adding new points to an existing model, two cases can occur. In the first case, all new points fit into the bounding box of the existing model, which means that the new points can simply be inserted into the existing model, i.e., the new points are converted to a binary stream and then inserted. In the second case, some (or all) of the new points are outside the bounding box of the existing model. In this case, the bounding box of the existing model has to be enlarged to enclose all the new points before they can be inserted. Furthermore, the LOD hierarchy of the MNO has to be updated after inserting the new points to keep it consistent.

To find the new bounding box of the model (including the old and new points), superior octree levels are added to the bounding box of the root node of the existing MNO. This is done iteratively. From the 8 possible superior bounding boxes for a new octree level, the one bounding box is chosen that minimizes the distance (center to center) between the bounding box of the new superior octree level and the bounding box of the new points. Superior octree levels are added this way until the root node of the expanded MNO encloses the bounding box of the new points completely.

After the bounding box of the MNO has been determined, the new points are inserted into the MNO. In the beginning, the MNO has one or more empty nodes in the levels above the old root node. For inserting the points, again the LRU cache is used to manage swapping the points of required nodes in and out of memory. When all new points have been inserted, it can still be the case that there are "holes" in the LOD hierarchy between the old root node and the new root node, as illustrated in Figure 3.3. To find points that fill the LOD hierarchy properly, the points of the old root node (actually a copy of them) are inserted into all newly created octree nodes above the old root node, to find empty grid cells (so the points from the old root node are not stored at the grid cells). If a point falls into an already occupied grid cell, nothing has to be done. If a point falls into an empty grid cell, the bounding box of the grid cell is then used to find a leaf node of the MNO from which a point can be inserted into this grid cell. When an appropriate leaf node has been found, one point is copied into the new grid cell and deleted from the leaf node. If a leaf node becomes empty due to this procedure, it is deleted from the MNO.

Note that an octree node can only be determined unambiguously as long as the bounding box of the new grid cell is smaller than or equal to the bounding box of the octree node. If the cell's bounding box is larger than the bounding box of the octree node, then it does not matter which child to follow to find a leaf node. In this case, one child node is randomly selected and the search for a leaf node is continued from there.

3.2.3.2 Deleting Points

The LRU cache is also used to manage the points in the nodes of the MNO when deleting points. After points have been deleted from the MNO, holes in the LOD hierarchy can appear. The holes in the LOD hierarchy are filled again by pulling up points from a leaf node that has a bounding box that is inside or encompassing the bounding box of the cell that holds the point to be deleted. If no point can be found to replace the deleted point, then this cell remains empty. The deleting algorithm traverses the MNO in postorder, so leaf nodes are processed first, and a leaf node is simply removed from the octree hierarchy when all points held by this node have been deleted.

Algorithm 3.2 shows the algorithm for deleting points from an MNO (the management of the additional point arrays at the node is omitted for the sake of convenience). The points to be deleted are first selected with a selection octree (see also Section 3.4). Points are deleted from leaf and interior nodes. Afterwards the hierarchy is checked for consistency.

3.2.4 Rendering

The rendering management of the original nested octree implementation is extended to allow more than one point cloud to be rendered out-of-core at a time. When several point clouds are loaded into a scene, they use the same node budget, i.e., the same priority queue, and the importance function designating which node to render is evaluated globally on all available and visible nodes from all point clouds. To have better control over the rendering performance, the number of points loaded per frame from main memory to graphics card memory (and vice versa) can be adjusted by the user. The reason is that loading too many points to (or from) graphics card memory in one frame could noticeably decrease the frame rate. For the test systems used in this thesis, the number of points was set to 100,000.

```
1  method delete(): // Method for deleting selected points
2      root node R->delete()
3      root node R->validate()
4  end
5  method node::delete(): // Method
6      foreach child C inside or intersecting SelOctree do
7          C->delete()
8      end
9      if node is inside SelOctree then
10         delete node from disk
11     else
12         if node is leaf then
13             delete points inside SelOctree
14         else
15             foreach point P inside SelOctree do
16                 find all direct and indirect child nodes that intersect grid cell GC
                   containing P
17                 delete P
18                 if at any child node exists a point PC inside GC then
19                     insert PC into GC
20                 end
21             end
22         end
23     end
24 end
25 method node::validate(): // Method
26     foreach child C that is an inner node do
27         C->validate()
28     end
29     foreach child C that is a leaf node do
30         if num points in C < m_min then
31             insert all points from C into this node
32             delete C from disk
33         end
34     end
35 end
```

Algorithm 3.2: Deleting points from an MNO.

3.3 External Sorting

In this section, the use of an external sorting algorithm before inserting huge point data sets into an MNO is discussed. The benchmarks done on a huge point data set show that the total build-up time, including the external sorting, can be accelerated by an order of magnitude. This section is not peer-reviewed, but gives valuable insights on the limitations of build-up strategies that use an LRU cache for the temporal storage of points, as the efficiency of such a cache is dependent on the order of the points in a data set.

In the course of a bachelor's thesis by Leimer [63], different implementations of an external merge-sort [58] algorithm were tested. These were applied on point clouds before inserting them into an MNO. The external merge-sort step was employed to counter a specific problem when dealing with large point data sets that are composed of several distinct smaller point data sets. Figure 3.4 gives an example of the problem. It is caused by the random spatial distribution of point samples in large data sets on the one hand, and the size of the data sets on the other hand. The size of the data sets comes into play as soon as the complete data cannot be loaded completely into memory, because then some out-of-core strategy has to be employed to process it. An out-of-core strategy is most efficient if the number of accesses to external memory can be kept low, as each access to external memory is about 6 orders of magnitude slower than an access to the main memory of a computer (see also Section 2.2.2).

When building an MNO from an almost unsorted point data set, a lot of points have to be swapped to external memory and loaded back again during the build-up process. Using the scan positions shown in Figure 3.4, the points from the green scan position on the very left could be inserted into the MNO first, and the points from the red scan position on the very right next. For this example, the assumption is made that the main memory is not large enough to cache all points from two scans. Therefore, points from the green scan position will already have to be swept out of memory when inserting the points from the red scan position into the MNO. When finally the points from the blue scan position are filled into the MNO, the points from the green scan position have to be loaded back to memory (to insert the points into the MNO nodes where the green and blue scans overlap), swapping out the points from the red scan position.

As can be seen by the previous example, the order in which points are inserted into an MNO will have an impact on the performance of the build-up algorithm if the main memory cannot hold all points. A possible improvement can be to sort the points before inserting them into an MNO.

For example, if the points are sorted along the horizontal axis in Figure 3.4 (from left to right), then less nodes will have to be loaded back to main memory during build-up. Another sensible choice is sorting according to Morton order [76], which is also called Z-order curve. Sorting points according to Morton order has the advantage that points are mapped from 3D space to 1D space. This means, points will be ordered along a space filling curve, which means they will be totally ordered. When sorting points along an axis it can happen that the order is not unique, as all points with the same position on the one axis but different positions on the other axes will remain unsorted (as long as the points are not sorted along the other axes as well). Another advantage of sorting according to Morton order is the fact that the resulting sorting is equivalent to traversing an octree (built from the points) depth-first. Therefore, the Morton

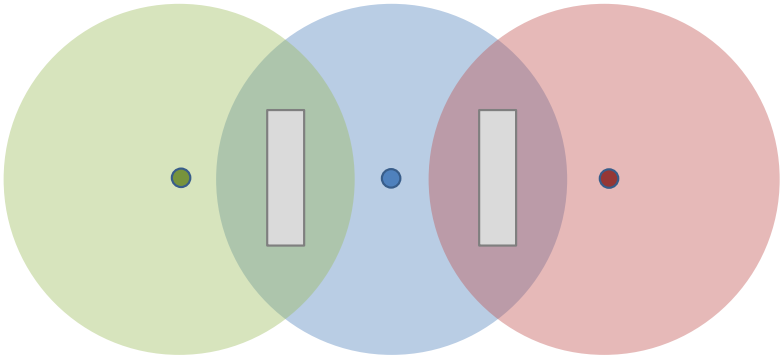


Figure 3.4: A top-down view onto the scanned areas of 3 distinct scan positions. The position of the scanner for each area is in the center of each circle, and the circles represent the maximum range of the scanner from each scan position. The gray rectangles represent footprints of objects in the scanned environment.

order matches very well the distribution of the octree nodes during the build-up of the MNO. A disadvantage is the time necessary to evaluate the Morton order, which can be somewhat larger than the time required to evaluate the order of the points along an axis (or along 3 axes, in case of ambiguity), but this might be compensated by a shorter build-up time of the MNO afterwards.

3.3.1 Sorting along an Axis

Sorting along an axis can be done easily by comparing the according coordinates of the points. Ambiguities may appear, however, when two or more points have the same coordinate for this axis. Then the coordinates of the 2 other axes (assuming the points exist in 3D space) have to be compared as well.

Usually the points are sorted along the longest axis of the bounding box of the point cloud. The longest axis is chosen to reduce the chance that points have the same coordinate on the sorted axis. The points along the other two axes remain unsorted though (if they do not have the same coordinate on the longest axis). Depending on the density and distribution of the points and on the size of the main memory cache, it can thus still be the case that points have to be swapped out of and into main memory several times during build-up.

The advantage of sorting along an axis is the simplicity (and therefore speed) of the comparison operation between two points. If the main memory cache during build-up is large enough to handle a situation like the one described above, then this sorting method and the build-up of the MNO afterwards are very fast.

3.3.2 Sorting according to Morton Order

Sorting according to Morton order establishes a total order on the points in 3D space, avoiding partially unsorted points as could happen when points are only ordered along one axis of 3D space. Figure 3.5 shows an example of a 2D grid, with grid cells numbered from 0 to 7 (also in binary code). The Morton order of the cells is calculated by intertwining the binary codes of the

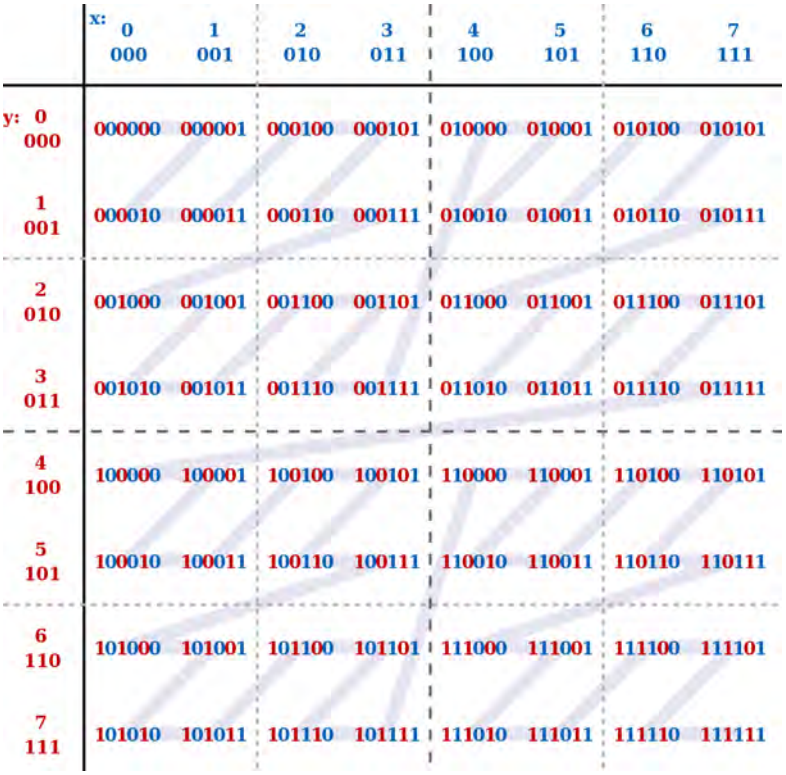


Figure 3.5: Binary numbered cells in a 2D coordinate system. The Morton order starts at coordinate (0,0) and then follows a z-shaped path (at different scales) which connects all 64 cells. Due to the shape of this path the Morton order is also called Z-order curve. The Morton order code is created by intertwining the bits of the binary representation of the coordinates.

2 dimensions (x and y) such that the most significant bit (MSB) of the y coordinate is followed by the MSB of the x coordinate, then the next less significant bits of the y and x coordinate follow, and so on, until the least significant bits (LSBs). Connecting the cells according to increasing Morton code draws z-like shapes into the grid (at various scales), hence Morton order is also called Z-order curve.

There are other mappings from 3D space to 1D space, e.g., Peano curve or Hilbert curve, but the Morton order has the property that the points will be sorted in a way that resembles a depth first traversal of an octree. Therefore, the Morton order is well suited for sorting points that will be filled into an octree.

In its original form, the Morton code can only be calculated for positive integer numbers, since the binary representation of the numbers is required for its calculation and so negative integers cannot be handled directly. Also floating point numbers cannot be converted directly to Morton code. In programming languages, floating point numbers are usually encoded according to the IEEE standard 754 (current revision of 2008) [98], where the bit pattern of a floating point number is defined by 3 parts. The first part is a sign bit. The next part is reserved for the exponent. Finally, the remaining bits are reserved for the mantissa, which is a decimal

number between 0.5 (inclusive) and 1 (exclusive). For example, a 32-bit floating point number is composed of a sign bit, 8 bits for the exponent, and 23 bits for the mantissa (which has a 24 bit precision, since the first bit is always assumed to be 1 except for the case when the exponent is 0).

For the sorting operation only the order of the points is important, not their binary representation. There are methods, as mentioned below, which can sort the points according to Morton order without changing their binary representation. This has the advantage that the points do not have to be converted during sorting, and that negative and floating point numbers can be used.

Instead of intertwining the bit representations of integer coordinates, the Morton order of two points can also be evaluated without actually calculating an integer Morton code, since the bit interleaving can be a time-consuming operation. Chan [19] presented a method where the bit representations of two coordinates are compared with the XOR logical operation (per bit). For two points in 3D space, the coordinate axis with the most significant differing bit (MSDB) has to be found, and only this coordinate has then to be compared according to Morton order.

This method can also be modified to compare two points, which have floating point coordinates. Connor and Kumar [21] suggest to do so by comparing first the exponent of the floating point coordinates, and only if the exponents are equal, the mantissae have to be compared. This means that at first, the coordinate axis with the highest exponent in one of the two point coordinates is chosen for comparison. If both points have the same highest exponent at the same coordinate axis, then the mantissae of those coordinates have to be compared. This is done by first XOR-ing the mantissae, then converting the result to a floating point value and adding its exponent (which is always negative) to the original exponent.

The sign of the floating point numbers is also used to decide the order of the two floating point numbers. In the following, it is assumed that the center of the bounding box is at the origin of the coordinate system. Then, if any coordinate of the two points has differing signs, the coordinate with the negative sign is smaller, and no further comparisons are necessary. Only if the signs are equal, the exponents (and maybe the mantissae) have to be compared. In case the signs of both numbers are negative, the numbers are multiplied by -1 and then compared.

Algorithm 3.3 shows the COMPARE method for two points with floating point coordinates. It returns true, if the point P is in front of point Q according to Morton order. It searches for the coordinate axis (or dimension) where the MSDB is at the highest bit position, and returns the result for the comparison of this coordinate. The input parameter $D = 3$ for 3D points. The EXPONENT method returns the integer exponent of the floating point number given as parameter and the MANTISSA method returns the mantissa of the given floating point number, respectively. The MSDB method returns the most significant differing bit as a floating point number.

The algorithm is based on Connor and Kumar [21], only the handling of the sign was explicitly added in the COMPARE method, and the check for an equal mantissa was added in the XOR_{MSB} method.

3.3.3 External Merge-Sort

External merge-sort is a well-known method to sort large data sets out-of-core [58]. The method can be divided into two main steps. In the first step, chunks of data that can be held in-core

```
1  method COMPARE(Point P, Point Q, int D):          // D = num dimensions
2      x ← 0
3      dim_max ← 0
4      foreach dimension d in D do
5          if SIGN(P(d)) ≠ SIGN(Q(d)) then return P(d) < Q(d)
6      end
7      foreach dimension d in D do
8          p_d ← P(d)
9          q_d ← Q(d)
10         if p_d is negative then
11             p_d = p_d * -1
12             q_d = q_d * -1
13         end
14         x_test ← XOR_MSB(p_d, q_d)
15         if x<x_test then
16             x ← x_test
17             dim_max ← d
18         end
19     end
20     return P(dim_max) < Q(dim_max)
21 end
22 method XOR_MSB(float a, float b):
23     a_exp ← EXPONENT(a)
24     a_mant ← MANTISSA(a)
25     b_exp ← EXPONENT(b)
26     b_mant ← MANTISSA(b)
27     if a_exp == b_exp then
28         if a_mant == b_mant then return 0          // Return min exponent
29         k ← MSDB(a_mant,b_mant)
30         return ← a_exp + EXPONENT(k)
31     else if a_exp < b_exp then return b_exp
32     else return a_exp
33 end
```

Algorithm 3.3: Comparing 3D points with float coordinates according to Morton order. Based on algorithms by Connor and Kumar [21].

are loaded from external memory into main memory. In main memory, they are sorted and then written back to external memory as distinct files. In the second step, the sorted files are merged to compile a completely sorted data set. For this, the first element of each file is loaded into main memory again, and the smallest (or largest, depending on the desired sorting order) is written to a new file. Then the next element from the file which held the just chosen element is loaded to memory, and from all elements in main memory the now smallest element is chosen. This process continues, until all input files have been read completely, and the result of the second step will be a completely sorted file.

When after the first step only two sorted files have been written to external memory, merging these two files in the second step is easy, as only the smaller of the two elements at the beginning of the files has to be chosen and written to the output file. As soon as there are more files and therefore elements to choose from, this merging task becomes more complex. One way to choose the smallest element of a set of k elements is by using a priority queue, which is often implemented as a heap sort on the elements (see also Section 3.3.3.1). Since using a priority queue changes the order of the elements, they have to store the index of the input file they belong to. When an element is then written to the output file, the next element can be loaded from the according input file. If there are in total N elements to be sorted, then the merging of the k sorted files with the priority queue can be done in $O(Nlog(k))$ time.

Note that external merge-sort is in a way the opposite of external distribution sort, where in a first step buckets for data ranges and their borders are determined, and the data elements are then sorted into those buckets. The buckets have the important property that their data ranges are mutually not overlapping, but forming a continuous partitioning of the data range. After filtering the data elements into the buckets, each bucket can be sorted in-core, and when concatenating all sorted buckets in the correct order, a completely sorted file can be written to disk. The problem with distribution sort is to find "good" values for the borders, as each bucket should fit completely into main memory for sorting. Since such a guarantee is difficult to make without first reading all data elements to get some statistics about the distribution of the values, this requires an additional step where all points have to be streamed through main memory one time more compared to external merge-sort.

For sorting the points in-core, any suitable sorting algorithm can be used. For this thesis, heap sort [109] and radix sort were chosen, and their performances were compared for a large point data set. Heap sort has an average and worst-case complexity of $O(Nlog(N))$, while radix sort has a worst-case complexity of $O(kN)$, which is better than the complexity of heap sort. In both cases, N is the number of elements to be sorted, while k is the number of passes for radix sort. However, radix sort has some disadvantages compared to heap sort. First, it is not an in-place sorting method, which means that it requires additional memory during sorting. Radix sort needs to store temporary results with worst-case $O(k + N)$ space complexity, while heap sort has no additional memory requirements and therefore a $O(1)$ worst-case space complexity. Note that the space complexity refers to the auxiliary space required for the sorting algorithms. The second disadvantage is related to the way radix sort works, as it directly compares the bit representation of the elements to be sorted. Therefore, it cannot be used when sorting points according to Morton order as shown above, since the Morton code is actually never calculated. Besides these issues, radix sort performs faster than heap sort.

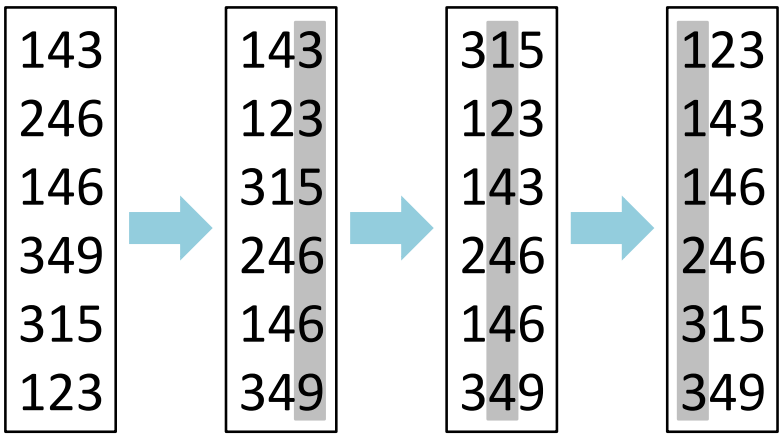


Figure 3.6: Radix sort in its least significant digit (LSD) variant with decimal numbers. The digits of the units, tens, and hundreds columns are used as keys. In a first pass, only the units column is sorted. In the following two passes the tens and hundreds columns are sorted respectively. The result is a completely sorted list.

3.3.3.1 Heap Sort

Heap sort works by first inserting all elements into a heap, which is a binary tree, where, in the case of a max-heap, the largest element is the root node (this property is valid for all nodes, i.e., each node's element is larger than the elements of its children). A heap is also a complete tree, i.e., all levels are completely filled except for the lowest level, depending on the number of elements the tree holds. For a min-heap, the root node would hold the smallest element (and each node would hold a smaller element than its children).

After inserting all elements, the largest element is removed from the heap and inserted into an array at the last position. The heap is reordered so that the largest element of the remaining elements is at the root node. This element is then again removed from the heap and inserted into the array at the second last position, and so on. The result of this operation is a sorted array.

3.3.3.2 Radix Sort

Radix sort was originally developed for tabulating machines by Hollerith in 1889 [51], and was restated by Seward in 1954 for the use with computers [96]. Radix sort does not compare the elements of a data set, but looks at the binary representation of the elements, and sorts the elements according to "keys", which are groups of binary digits. For example, a 32-bit integer can be divided into 4 keys of 8 bits. Although initially developed for use with positive integer numbers, radix sort is not limited to those numbers, and extensions for floating point numbers exist [103] [49].

The sorting algorithm can be easily explained with decimal numbers by using the digits as the keys. The numbers will be sorted subsequently according to their digits, starting with the units column. After the numbers have been sorted according to their units column, they are sorted according to their tens column, and so on. This eventually results in a completely sorted

set of decimal numbers. There are two variants of radix sort, one variant is the least significant digit (LSD) radix sort, and the other is the most significant digit (MSD) radix sort. The LSD radix sort starts sorting with the least significant digit and moves subsequently towards the most significant digit. The MSD radix sort starts with the most significant digit and then moves in the opposite direction. Figure 3.6 shows an example of the LSD radix sort variant on a list of decimal numbers.

3.3.4 Parallel Sorting

The performance of the external merge-sort can be increased by exploiting the parallel processing capabilities of computers that have multiple (or multi-core) CPUs available. Another way to increase the performance would be to use multiple hard disks to decrease the latency of the external memory accesses, but since this is a less common configuration for off-the-shelf computer systems, such a configuration was not tested for this thesis. Parallel processing can be exploited at two stages of the external merge-sort algorithm, first, when sorting the chunks of the point data sets in-core, and second, when merging the single files to a completely sorted file.

In the first case, when sorting chunks of points in-core, multiple threads can be used to sort one chunk per thread. Since each chunk requires its own memory space, the chunks are smaller compared to sorting one chunk in a single thread at one time. However, the sorting performance depends on the number of items in the chunks, so when sorting smaller chunks, this will already speed up the sorting performance. But sorting smaller chunks also has disadvantages, as due to the smaller size of the chunks, more files are created, therefore more files have to be merged in the second step of the external merge-sort algorithm, and thus the merge step will become slower. Nevertheless, often the increased sorting speed outbalances the slower merge speed, so parallel sorting is often advantageous.

In the second case, when merging the single files to a completely sorted file, parallel merging can be implemented by merging subsets of the elements in multiple threads parallel, and then merging the resulting subsets for the final output in a single thread. The possible performance improvement is dependent on the access time to external memory, as several threads try to fetch elements from the sorted files at the same time, which might even cause idle times for some threads, until the elements are available.

3.4 Selection Octree

Selecting points in an out-of-core setting is not trivial, since not all points are available in memory at a time. If selection were to happen on individual points of a surface, moving closer to the surface could make a more detailed LOD level appear, and the points of this "new" LOD level would not be selected, although they belong to the selected part of the surface. To overcome this problem, a data structure is proposed in this thesis, which allows to define the selection not on individual points, but on the volume that the selection encompasses. The so-called selection octree is introduced as a selection primitive. The selection octree is a separate data structure with the same root node as the MNO of the point model. All points whose positions are within the selection octree will be marked as selected. Points that are loaded after defining the selection

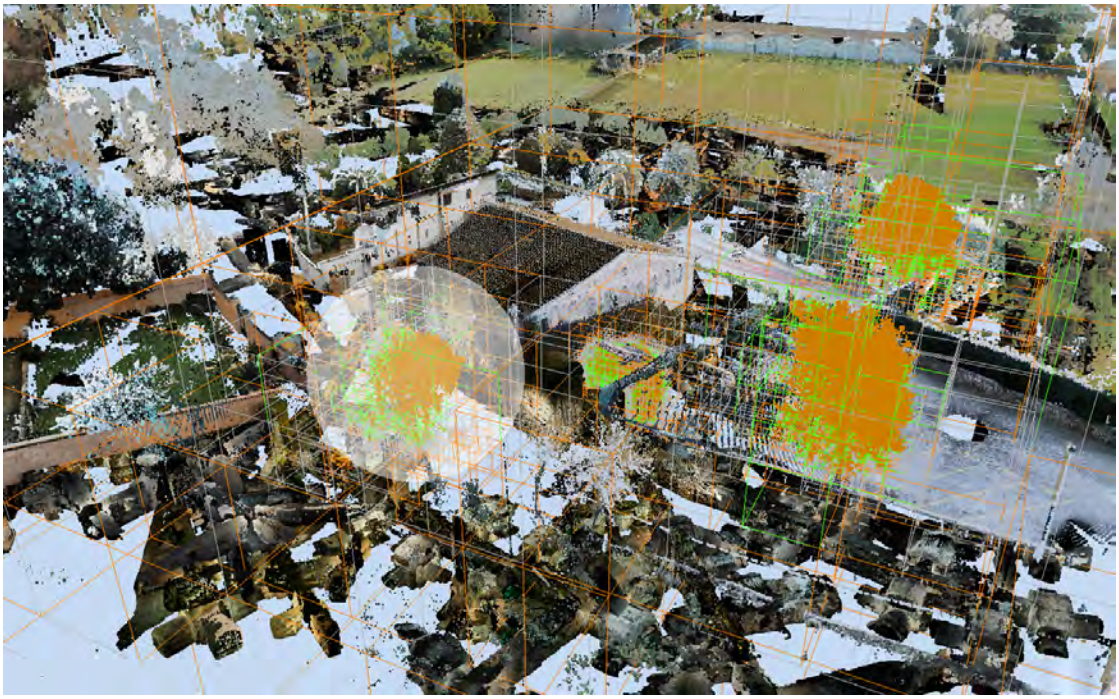


Figure 3.7: The nodes of the selection octree that encompass the selection are shown in green. All selected areas are managed by the same selection octree, the nodes connecting the areas are shown in orange. Empty leaf nodes are shown in grey. The semi-transparent sphere is the selection sphere used to select and deselect points (see also Section 3.4.1).

just have to check whether they are inside the selection octree or not, and they are then marked accordingly. The points on disk are not changed during selection. The selection remains valid when the user changes the viewpoint.

Contrary to selection volumes used in volume rendering [16, 17], the selection octree is not based on a regular grid, as the points sampled by laser scans are often irregularly distributed, but (as the name implies) on an octree, as an octree is built hierarchically and therefore uses less memory in areas with low point density. Figure 3.7 shows an selection octree, where the edges of the nodes of the octree are colored by their state. The leaf nodes encompassing the actual selection volumes are colored green, empty leaf nodes are colored grey. The other nodes building the hierarchy are colored orange. Note that the selection does not have to consist of neighboring leaf nodes, so different areas can be selected by the same selection octree.

A selection octrees typically has a small memory footprint, so it can be stored in-core. The number of selection octrees that can be handled at the same time is not limited per se, as the points are only tested against a selection octrees when they are loaded from disk, or when a selection changes. Therefore, when simply navigating through the point model, the rendering performance is not dependent on the number of selection octrees. Limitations that might occur are the time it requires to test the points against the existing selection octrees, and also the memory consumption when handling a lot (more than 100) selection octrees.

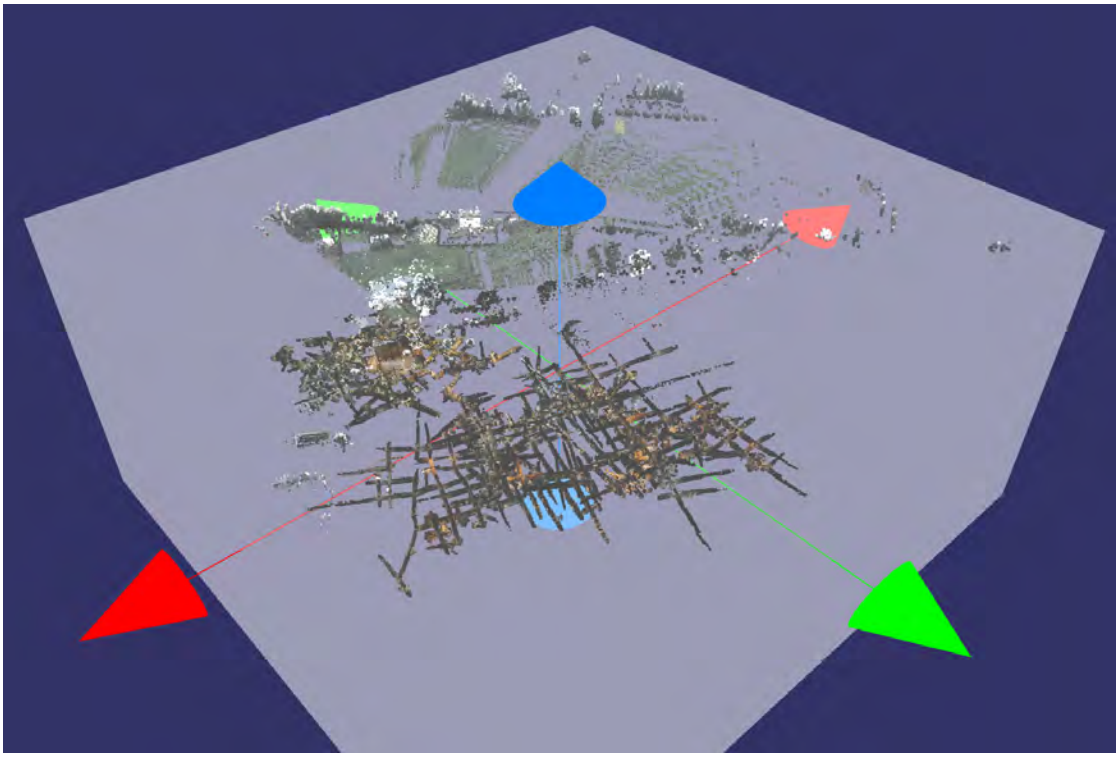


Figure 3.8: The box tool can be used to select large, continuous areas.

When loading points from disk, each point streamed into memory has to be tested against all existing selection octrees. This will increase the time it takes from requesting the points for rendering until they are finally available on the graphics card. The test of the points against the existing selection octrees can be sped up by implementing the test directly on graphics card (e.g., by using a general purpose GPU (GPGPU) computing approach), as shown in a diploma thesis by Tragust [104]. The benchmarks presented there show that the test time can be decreased by about 50% in general. Another limit that might occur is the amount of available main memory. A typical selection octree uses about 1 Megabyte, where one selection octree node uses 12 Bits for its state and 1 pointer to its children, which are stored in an array. Several selection octrees can be used for different selections at the same time, and points can be part of different selection octrees at the same time.

3.4.1 Selecting Points

A selection octree is built from the points that are selected by the user. For user interaction, a volumetric selection brush [17, 108]is provided, which follows the surface of a point model by reading the Z-Buffer. If there is no valid entry in the Z-Buffer for a position, i.e., the Z-Buffer is at negative “infinity” for this position, than the z-value of the last valid position is used. The user can interact with the brush by moving the mouse. On pressing the left mouse button, all points that are loaded to the graphics card and that are inside the selection brush are marked

by a flag in the alpha channel of the color of the point (the alpha channel is not needed for rendering) by continuously intersecting the volume of the selection brush with the nodes of the MNO. On releasing the mouse button, all points (marked and not marked) from the nodes that were intersected by the volumetric brush since pressing the mouse button are copied into an array and inserted into the selection octree.

While the brush tool is well suited for freely selecting areas of a point model by moving the brush over the model's surface, sometimes the volume that shall be selected cannot be easily defined this way. For example, when outliers in a scan shall be selected, it is difficult to use the brush tool, as the outliers are usually scattered in space, and do not constitute a surface. In such situations, it might be easier to define the volume in another way, and for this the so-called box tool is provided. The box tool is a selection primitive, which can be adjusted in size and orientation before selecting the points inside its volume. In Figure 3.8, the box tool is shown, spawning over a point cloud. The colored cones, which act as manipulators, can be used to change the positions of the box's borders, and the box can also be rotated as a whole. When the desired position, size, and orientation has been set, the user can then select all points inside the box's volume. For example, if the user fits the box tool tightly to the boundaries of the object represented by the point cloud, he can first select all points belonging to this object, and then by inverting the selection, i.e., all unselected points become selected and vice-versa, he can select all the outliers outside the box in one pass. Another application for the box tool is quickly creating cuts through a point cloud. Once the orientation of the box has been fixed, two opposing sides of the box (e.g., the upper and lower side) can be shifted to select points inside a slim cuboid, for example to create a floor plane from the scanned object.

3.4.2 Building a Selection Octree

After the points have been selected by the user (either by using the brush tool or the box tool), they are inserted at once into the selection octree, and they are filtered down the hierarchy according to their position. Creation of new child nodes is continued until either only one point is left at a node, or if only marked or unmarked points exist at one node. Note that actually no points are stored in the selection octree permanently, the points are only used to build the octree hierarchy. The points that have not been marked also have to be inserted into the selection octree, as they are necessary to determine the exact border of the selection octree.

Building the selection octree is quite fast (see Section 3.11). The build-up time is longest for very large selections (up to 2.5 seconds for about 13 million points), while for smaller selections it is unnoticeable by the user. The selection octree can be updated in the same way as it is built, i.e., inserting new points into the already existing octree hierarchy. For further usage, selection octrees can also be saved to disk and loaded again later.

3.4.3 Building a Deselection Octree

While a selection octree defines the volume inside which points are selected, a deselection octree defines a volume inside which points become deselected. A deselection octree is a temporary data structure in a sense, as its only purpose is to reduce the volume of a selection octree, and afterwards it is deleted again. Only one instance of a deselection octree exists at any time, so

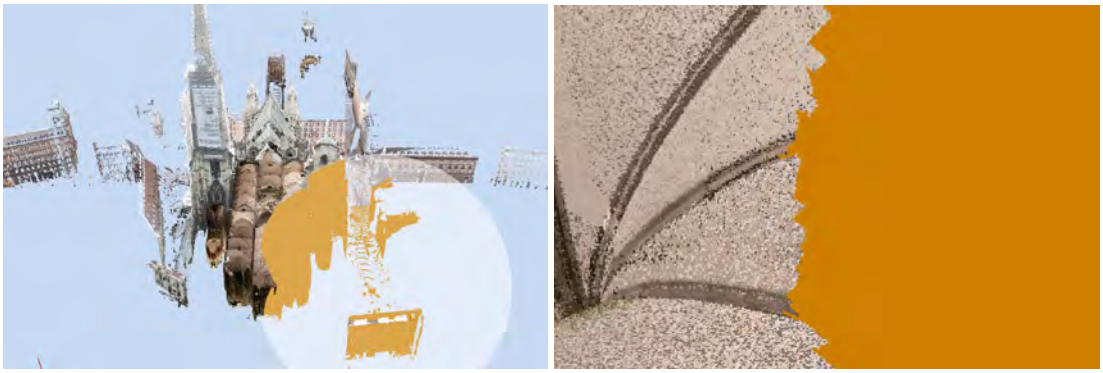


Figure 3.9: Selecting points in some distance to the model results in aliasing for the border of the selection when zooming into the model, as can be seen in the right image.

when deselecting several areas from a selection octree, several deselection octrees have to be built sequentially, one for each area that shall be deselected.

Building a deselection octree is done in a similar way as building a selection octree. The user again marks points with one of the selection tools, but now he can only mark points that are already part of a selection. On pressing the right mouse button, all selected points within the selection tool are marked by a flag that they will be deselected. On releasing the mouse button, all points of all MNO nodes that were intersected by the selection tool are inserted into the deselection octree. For collecting the points, they are copied from the graphics card into an array in main memory. After they have been copied (the copied points retain their alpha channel flags) the points on the graphics card reset all flags in the alpha channel, so they become unselected points again.

The deselection octree represents the volume described by the deselected points, i.e., points which are still selected and normal points are not included in this volume. Afterwards, this volume is subtracted from the currently active selection octrees (one or more selection octrees can be active at the same time), which lasts only some milliseconds. The now reduced selection octrees are optimized by merging nodes with the same properties to larger nodes. The deselection octree is then deleted from memory.

3.4.4 Properties of the Selection Octree

All points that are marked as selected can be rendered as invisible by projecting them to infinity inside the vertex shader (by setting the point's w-coordinate to 0), which enables a preview of the point model without the selected points.

The maximum resolution of the selection octree is bounded by the size of the grid cells (that are used to build the MNO, see Section 3.2.1) of the currently rendered octree nodes. If the user is selecting points at a coarse LOD, the resolution will be low, e.g., the bounds of the selection will exhibit aliasing artifacts. This can be seen when approaching the model and therefore rendering it at a finer LOD, as shown in Figure 3.9. There does not seem to be an obvious solution to this problem, as an automatic refinement of the selection octree, without

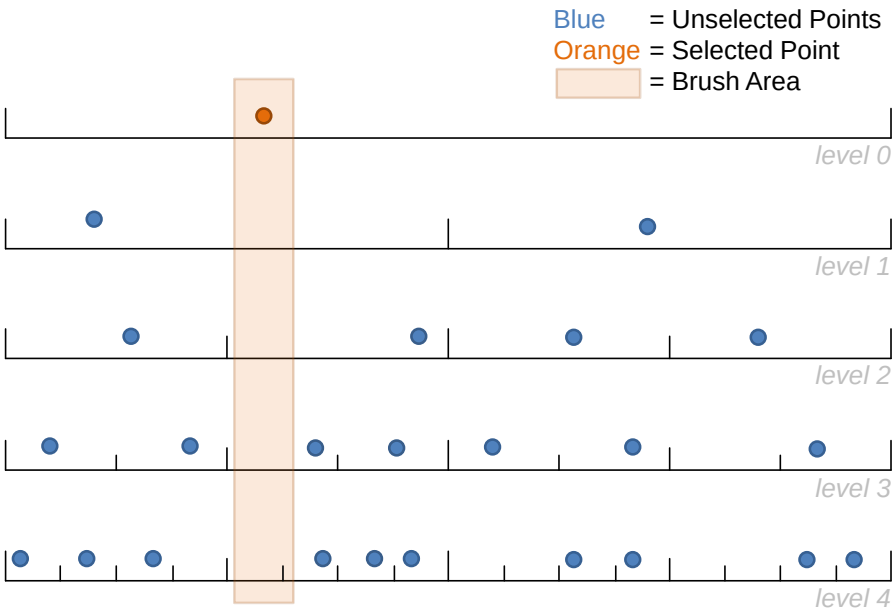


Figure 3.10: Schematic illustration of a brush (orange shaded area) intersecting an MNO hierarchy. The MNO is not completely filled with points. The point in level 0 is inside the brush, therefore it is selected. In the other levels of the MNO, no points are inside the brush.

inserting new points, might not be in coherence with the points of the model. Another option would be to define a hull around the selection octree, and when loading new points from disk, to insert all points into the selection octree that are in the space between the outer border of the selection octree and the border of the hull around the selection octree, but this was not tested. On the other hand, the selection octree represents exactly the LOD that was available to the user when he created the selection, so the result is not unexpected.

3.5 Selection Octree Accuracy

In this section, a modification to the build-up process of the selection octree is proposed. More specifically, points of certain unselected MNO nodes shall be inserted into a selection octree as well, to improve the accuracy of the selection volume. The proposed modifications are not peer-reviewed, but are relevant in the scope of this thesis.

3.5.1 Updating

While initially, the selection octree was mainly designed to describe continuous areas in point clouds, it is also possible to use it for selecting single points such as outliers. In such cases, care has to be taken that the selection octree properly represents the selected single points. Since the selection octree is built from points that are visible in the viewport, it can be the case that the density of the points inserted into the selection octree is quite low, for example if they are part of

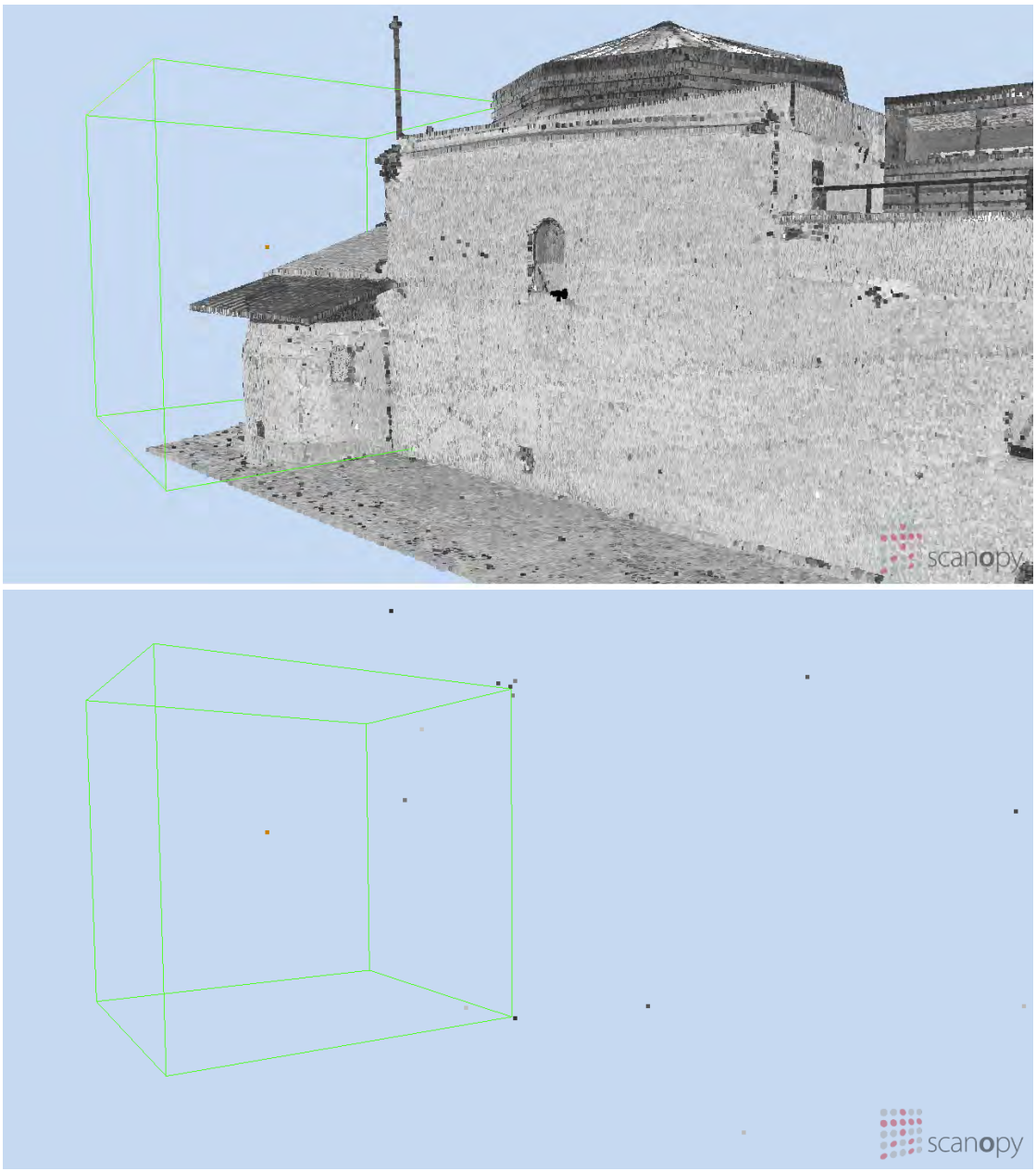


Figure 3.11: In both images a single point is selected, which is rendered orange. Only the points of the MNO node which this point belongs to are inserted into the selection octree. The points of this node are shown in the lower image. The resulting selection octree node is shown with its bounding box rendered in green. In the upper image, it can be seen that the bounding box intersects large parts of the building, although the selection octree node should only encompass the selected point. The points are rendered with a splat size of 5 pixels, so the single points in the air are easier to see.

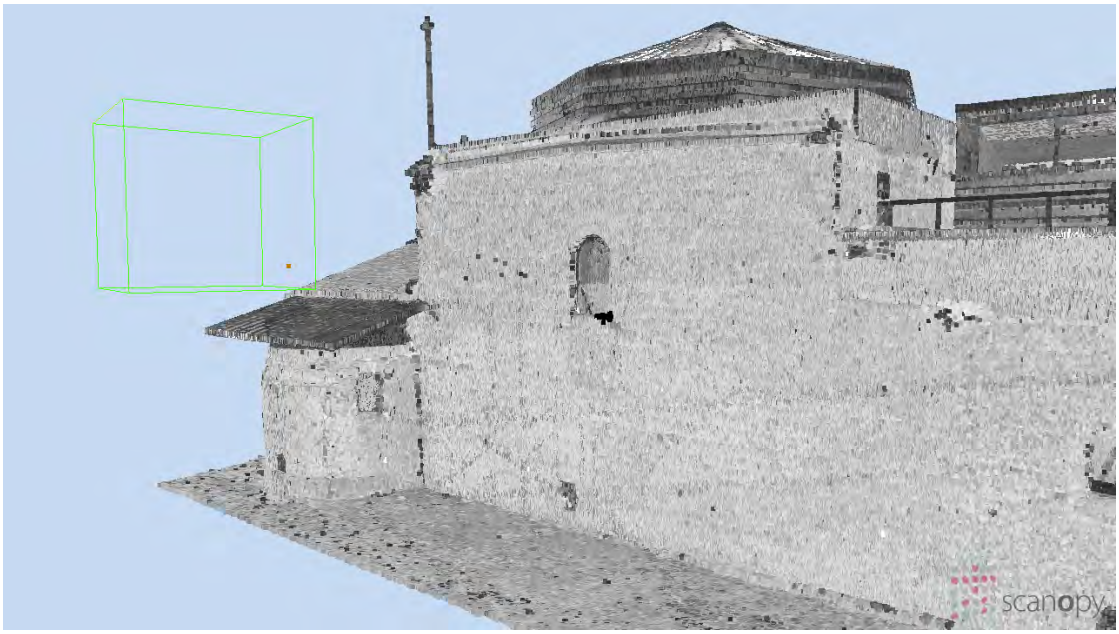


Figure 3.12: In this image, all points of the MNO nodes intersecting the brush have been used to build the selection octree. The result is a selection octree node that only holds the single selected point.

a high MNO hierarchy level. In a naïve implementation of the selection octree, only the points of the MNO nodes where the selected points reside in are used to update the selection octree. This will create a selection octree that does not represent the shape of the point model, and it might even intersect the point model in unselected areas.

Figure 3.10 shows an example of a single selected point residing in MNO level 0. The bounding area of the brush intersects the lower levels as well, but no points in the lower levels are inside the brush. When using only the point of level 0 to update the selection octree, the point will be filtered down the hierarchy levels of the selection octree, and early, at a high level, it will be the only selected point inside a node, therefore subdivision stops, and a large node will be created. This node does not represent the shape of the point model though. Therefore, to increase the fidelity of the selection octree, it is not enough to use only the point of the MNO node where the selected point belongs to. Instead, the points of all nodes that are intersected by the brush and that are available in graphics card memory have to be used. In the example of Figure 3.10, this would mean that not only the point at level 0 has to be used, but also the points of all nodes that intersect the brush from level 1 to 4 (e.g., the left node at level 1, the second node from the left at level 2, etc.). This way, the resolution of the selection octree volume can be increased, and the shape of the point model can be approximated more precisely.

In the top image of Figure 3.11, the problem is shown on a point model. The selected point (orange) created a selection octree node (green bounding box) that intersects the point model. The selected point resides in the root node of the MNO. The points of the root node are shown in the bottom image. In total, the root node holds 23 points, and only these points were inserted

into the selection octree. Their sampling density is very low, as the root node bounding box is very large (with a side length of about 1600 meters) compared to the size of the building (on the longest side it is about 40 meters long). When using the points of all nodes that are intersected by the brush, the selection octree will be further subdivided, and a smaller node, as can be seen in Figure 3.12, will be created. In total 113,039 points from 8 nodes at different hierarchy levels were inserted into the selection octree, and these points represent the local shape of the point model already quite well.

3.5.2 Optimization

After updating the selection octree with new points, the "resolution" of the nodes in the selection octree is often too high, as the nodes are not aware of the state of their parent node or sibling nodes, i.e., if they hold points, hold selected points, or are empty. These aforementioned states are also the three states a selection octree node can have. So in a final step, after updating the selection octree with new points, an optimization traversal on the hierarchy is performed.

The optimization is done in a post-order traversal starting at the root node, so the optimizations are performed bottom-up, starting at the lowest hierarchy level. During the optimization, a node checks the states of its child nodes, and if they fulfill certain conditions, they can be merged (i.e., deleted), and the node updates its state accordingly.

First, the child nodes are checked whether they are all selected. If that is the case, it is safe to assume the parent node is selected as well. Note that the number of children of every node is always 8 (i.e., the maximum number of children for an octree node), as this makes sure that the volume of the selection octree is properly defined. Next, the child nodes are checked if they are all empty. If that is the case, the empty state can be taken over by the parent node. At last, it is checked if the sum of all empty and filled (but not selected) child nodes equals 8, which allows the parent node to set its state to filled (but not selected). Algorithm 3.4 shows an implementation of this optimization strategy.

3.5.3 Deleting

Deleting nodes from a selection octree is the result of first deselecting points in the MNO, then building a deselection octree from the deselected points, and finally subtracting the deselection octree from the selection octree.

Algorithm 3.5 shows how nodes are deleted from a selection octree by subtracting nodes of a deselection octree. First all "deselected nodes" (i.e., nodes representing deselected points) of the deselection octree are gathered in a list $bblist$, and then the bounding box of each deselected node is intersected with the selection octree. If a node in the selection octree is found that has the same bounding box, then that node is marked as not selected, and all its child nodes are deleted. A special case occurs when a leaf node of the selection octree is larger than a deselected node. In this case, the leaf node of the selection octree has to be subdivided, so 8 child nodes are created, and the recursion continues until a node is found that is small enough to fit the deselected node. When all nodes in the $bblist$ have been intersected with the selection octree, a final optimization traversal (as described in Section 3.5.2) is performed on the selection octree.

```
1  method SelectionOctree::Optimize():
2      rootnode->OptimizeTraversal()
3  end
4  method SelectionOctreeNode::OptimizeTraversal():
5      if node is not a leaf node then
6          foreach child node c do
7              c->OptimizeTraversal()
8          end
9          count number of selected child nodes
10         count number of empty child nodes
11         count number of filled child nodes        // filled but not selected
12         if number of selected child nodes == 8 then
13             set node is selected
14             set node is not empty
15             delete all child nodes
16         else if number of empty child nodes == 8 then
17             set node is not selected
18             set node is empty
19             delete all child nodes
20         else if number of filled + empty child nodes == 8 then
21             set node is not selected
22             set node is not empty
23             delete all child nodes
24         end
25     end
26 end
```

Algorithm 3.4: Optimizing the nodes of a selection octree after it has been updated.

3.6 Point Size Heuristic

One of the most critical aspects when interacting with a huge point cloud is choosing the point sizes in a way that holes are avoided when rendering closed surfaces. This is important to allow selection operations, because otherwise, connected surfaces would kind of "dissolve" into single points when viewed from a closer distance. On the other hand, often no assumptions can be made about sampling densities or the existence of surfaces in a point cloud, since point clouds composed of scans from several different scan positions often have wildly varying sampling densities. Therefore, in this thesis a point size heuristic is presented that is based on the depth of points inside a hierarchy and on a local density estimate. Figure 3.13 gives an example of the varying sampling densities when rendering a point-based model and the effect of the proposed point-size heuristic.

For the points in an MNO node whose children are not rendered (i.e., a leaf node or a node whose children are too small to be rendered when projected to screen space), simply the side

```
1  method SelectionOctree::Subtract(SelectionOctree deSelOct):
2      bblist ← deSelOct->GetBoundingBoxesFromSelectedNodes()
3      rootbb ← GetBoundingBox()
4      foreach BoundingBox bb in bblist do
5          rootnode->SubtractTraversal( rootbb, bb )
6      end
7      Optimize()
8  end
9  method SelectionOctreeNode::SubtractTraversal(BoundingBox nodebb, testbb):
10     if testbb ≤ nodebb then
11         if testbb == nodebb then
12             set node is not selected
13             set node is not empty
14             delete all child nodes
15         else
16             if node is leaf then
17                 set node is not empty
18                 create 8 children with same state as node
19             end
20             foreach child node c do
21                 childnodebb ← nodebb->CalculateBoundingBoxForChild( c )
22                 c->SubtractTraversal( childnodebb, testbb )
23             end
24         end
25     end
26 end
```

Algorithm 3.5: Deleting nodes from a selection octree by subtracting nodes from a deselection octree.

length of a cell of the node's inscribed grid is chosen as the world-space point size. The point size is then projected to the near plane of the view frustum, and so determines the point splat size for the points of the node in screen space.

The problem is choosing the point sizes for points that are stored in the upper nodes of the MNO. The point sizes should fit the sizes of the surrounding points, which are stored in nodes at lower levels. If the grid cell sizes were used, as for the leaf nodes, points would become huge in upper hierarchy levels and hide all surrounding points. Note that no assumptions are made on the distance between neighboring points.

Instead of using a point-size heuristic, also other methods could be used to render a closed surface from non-uniformly sampled point clouds. One possibility would be to use the point sizes of the rendered nodes that are furthest down the hierarchy for all points inside the volumes of these nodes. But this approach tends to create abrupt changes in the point sizes between neighboring areas, especially where the depths of the nodes that are used for the point sizes

Figure 3.13: A view inside the basilica of the Domitilla catacomb model. On the left side, the point cloud is rendered with a fixed point size of one. The sparsely sampled areas are partly due to the different sampling densities, partly due to the LOD algorithm, which prefers to load nodes in the center of the screen. On the right side, the point cloud is rendered with the weighted point size. The level weight is calculated from the exact distribution of the points in the MNO.

differ too much. Other methods, like calculating the point size for every rendered point, would also be possible, but were not tested during this thesis.

The heuristic calculates a weighted point size that reflects the number of points in the children of a rendered node as a local density estimate. Only nodes that are rendered contribute to the point-size estimation. The point size is influenced by two weights, namely the number of points in an MNO node and the so-called level weight. From those two weights a virtual depth is derived, which then determines the point size (the terms depth and hierarchy level are used interchangeably).

3.6.1 Virtual Depth

The first weight is related to the number of points in a node. The more points in a node, the greater its influence on the point size.

The point size is calculated for each node that is rendered in the current frame. The heuristic calculates a (potentially non-integer) virtual depth of the points in a node, so that they are rendered as if they were at another depth in the hierarchy. The virtual depth vd_q for one query node q is calculated as

$$vd_q = \frac{\sum_i n_i \cdot d_i + n_q \cdot d_q}{\sum_i n_i + n_q} \tag{3.1}$$

where n_i is the number of points at node i and d_i is the depth of node i in the hierarchy. Note that the index i goes over all rendered direct and indirect children of a node, i.e., not only over the direct children of the node. Intuitively, this means, if a node has many points in a child of depth d, then the point size of that node will be similar to the point size at depth d, in order to let the points in the inner node blend into the points of the "dense" children.

When only using this weight, the points of the upper levels in the hierarchy are still rendered too large compared to the points of the lower levels. This can be accounted for with the level weight.

3.6.2 Level Weight

The second weight is related to the number of points per hierarchy level. This so-called "level weight" is basically a normalized histogram, where the bins correspond to the levels in the hierarchy, and the values of the bins are the weights of the according hierarchy levels. Those weights are calculated by counting the number of points at the hierarchy levels, and then normalizing them by the total number of points. This means, the bin where the most points are counted gets the highest weight. The sum of the weights of all bins adds up to 1 (i.e., the weights form a partition of unity).

There are different ways to count the points per level, because the MNO stores the points in the leaf nodes and in the inner nodes, but the points in the inner nodes actually contribute to the wrong level, as they should be counted at the levels of the leaf nodes they would fall into. Later in this section, three different methods for counting the points are discussed. Each method causes a different distribution of the points in the bins of the histogram.

The effect of the level weight is a harmonization of the point sizes, and since most of the points are in the lower levels of the hierarchy, the influence of the points in the upper hierarchy levels is reduced. These points would otherwise cause larger point sizes for upper levels, and so they would stand out from the rendered points of the lower levels, due to their size. This is especially useful for nodes whose sub trees differ a lot in height, because then (direct or indirect) child nodes at middle levels with a lot of points have less influence on the point size than (direct or indirect) child nodes at lower levels with not so many points.

While the virtual depth as calculated above (see Equation 3.1) captures the influence of the local sampling density on the point size, the level weight accounts for the global distribution of the points in the hierarchy. When factoring in the level weight, the virtual depth is calculated as

$$vd_q = \frac{\sum_i n_i \cdot lw(d_i) \cdot d_i + n_q \cdot lw(d_q) \cdot d_q}{\sum_i n_i \cdot lw(d_i) + n_q \cdot lw(d_q)} \tag{3.2}$$

where $lw(d_i)$ is the level weight for node i at depth d (all nodes of the same depth have the same level weight). This way, the level weight can be used to influence the virtual depth by scaling the number of points per node.

The three different methods for counting the points in the MNO levels are presented in the following.

3.6.2.1 Simple Level Weight

The simple level weight just counts the points at each level, and stores the results in the bins of the histogram. After counting the points, the histogram is normalized. As mentioned above, the points in the inner nodes should be counted at the levels of the leaf nodes they would fall

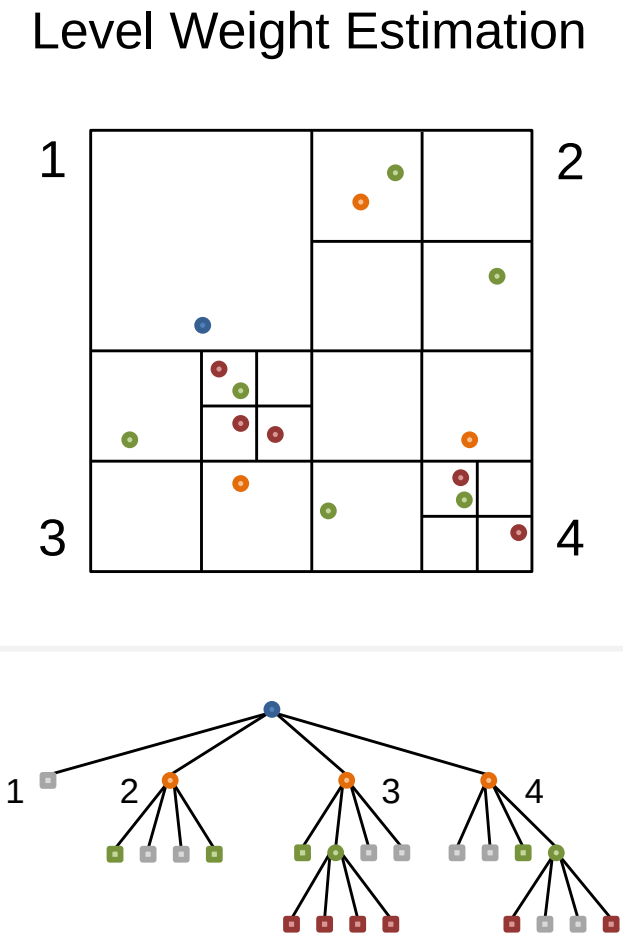


Figure 3.14: The 2D version of an MNO used for the level weight estimation example.

into, but the simple level weight does not account for this, in contrast to the two other methods presented below. The number of points nl at depth d can be calculated as

$$nl(d) = \sum_i n(i_d) \tag{3.3}$$

where $n(i_d)$ is the number of points at a node i residing at depth d. The index i goes over all nodes at depth d.

3.6.2.2 Exact Level Weight

The exact level weight calculates the number of points inside the leaf nodes by counting all points (from all hierarchy levels) that fall into the bounding boxes of the leaf nodes (note that there also exist so called "semi-leaf nodes", which are inner nodes that do not have the maximum number of child nodes, so for the points inside the bounding boxes of the empty child nodes, these inner nodes are actually leaf nodes).

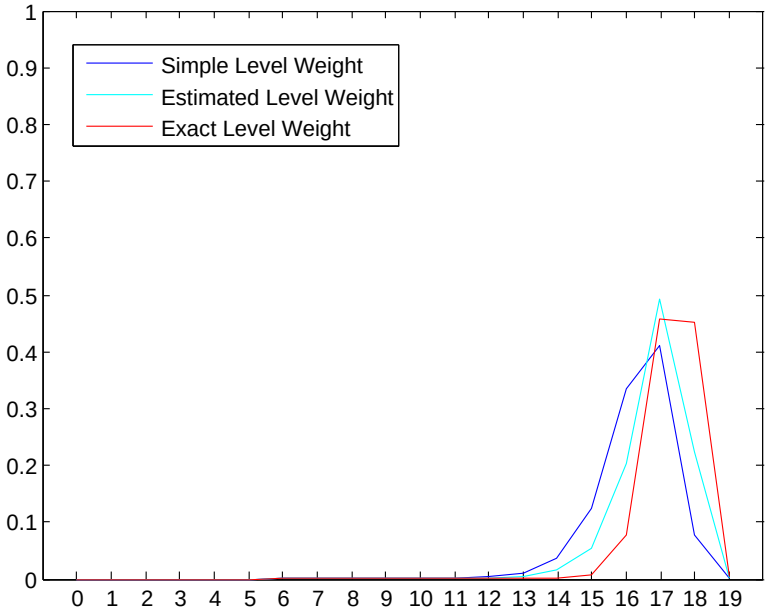


Figure 3.15: Comparison of the different level weight calculation methods for the model of the Domitilla catacomb. With the exact calculation method some 90 percent of all points are in the 2nd and 3rd lowest levels combined.

For this, the bounding box of each leaf node is intersected with the bounding boxes of all direct and indirect parent nodes, and the points of the parent nodes that are inside the leaf nodes' bounding boxes are counted as if they were residing in the leaf nodes, i.e., they are added to the level the leaf nodes are residing in. They are not added to the level they are actually residing in. The points of the leaf nodes, however, are added to the exact levels the leaf nodes resides in (the same is also done for the bounding box of each empty child node from the semi-leaf nodes, as they also act as leaf nodes). The number of points n in a leaf node q can thus be calculated as

$$n(q) = \sum_{j} \sum_{p \in pn_j} p \in bb(q) \tag{3.4}$$

where p is a point stored at node pn, and $bb(q)$ is the bounding box of node q. Index j is running over all direct and indirect parent nodes of q, starting with node q (i.e., $q = pn_0$). This equation for $n(q)$ can be plugged into Equation 3.3 as substitution for every $n(i_d)$.

This method, however, requires to load all point data from disk to memory, as otherwise it is not possible to test if the points stored at the parent nodes are inside the bounding boxes of the leaf nodes.

3.6.2.3 Level Weight based on Uniform Sampling Distribution

The last presented method for calculating the level weight estimates the number of the points in the leaf nodes by assuming a uniform sampling distribution per node. This can be achieved

Figure 3.16: The points of the different levels are blended to make a closed surface of the point model. In the closeup, points from hierarchy level 8 and 9 are shown. All levels combined result in the closed surface for the point model.

by multiplying the fraction of the volume a leaf node takes inside the volume of its (direct or indirect) parent nodes with the number of points in each of these parent nodes, and summing up the results of these products.

In contrast to the calculation of the exact level weight, where for each leaf node the points within its bounding box are actually counted, the calculation for this method is based on the number of points per node. While it is not as accurate as the previous method, it has the advantage that it is not necessary to load the points from disk.

Calculating the number of points for all leaf nodes under the uniform sampling assumption is done by traversing the octree hierarchy, and calculating for every leaf node and semi-leaf node q the value $pe(q)$, which is the percentage of empty child nodes for q (the percentage is represented by a decimal number between [0..1]). Then, the node q and all direct and indirect parent nodes of q are traversed, and the number of points at these nodes pn_j is then multiplied by the percentage that the volume of the empty nodes of q takes inside the volume of pn_j. The sum of the number of points calculated this way at each pn_j equals the number of points $n(q)$ for which q represents the leaf node. The equation for $n(q)$ is

$$n(q) = \sum_j n(pn_j) \frac{v(bb(q)) \cdot pe(q)}{v(bb(pn_j))} \tag{3.5}$$

where the sum goes over the query node q (i.e., $q = pn_0$) and all direct and indirect parent nodes of q. The term $v(bb(q))$ is the volume of the bounding box of q, and the term $v(bb(pn_j))$ is the volume of the bounding box of the currently traversed node pn_j respectively. The term $n(pn_j)$ is the number of points of node pn_j. The equation for $n(q)$ can again be plugged into Equation 3.3 as substitution for every $n(i_d)$.

An example for the level weight estimation is given in the following, using the hierarchy of Figure 3.14 (a 2D version of an MNO). The tree is traversed starting at the root node. For all

nodes in the MNO, the number of estimated points is calculated by equation 3.5. At the root node, this gives $n(q) = 1 \cdot 1/4 \cdot 1 = 0.25$, since only one child node of the possible 4 (because of 2D) is missing. At level 1, below the root node, all nodes have 2 missing child nodes, so $n(q) = 1 \cdot (2/4) \cdot 1 + 0.25 \cdot (2/4) \cdot 1 = 0.625$ for each node. The number of points at level 1 is therefore $0.625 \cdot 3 = 1.875$. The number of points at the remaining nodes in the tree can be estimated similarly. An interesting situation occurs at level 2, at node r32 (when using the node position as digit in the path to the node, and r for the root node), as it has no empty child nodes. Therefore, the estimated number of points at this node is 0. The number of points for each level are (from level 0 to level 3): 0.25, 1.875, 5.90625, and 7.96875. The total number of points is 16, which is equal to the actual number of points, but distributed differently across the levels. The (approximate) level weights in percentage (from level 0 to level 3) are therefore 0.015, 0.12, 0.37, and 0.5 respectively. When compared to the original level weights of 0.063, 0.19, 0.38, and 0.38, the weight of level 3 has increased significantly.

3.6.3 Discussion

3.6.3.1 Varying Point Sizes while Loading Points

The point sizes calculated by the point size heuristic correspond to the density of the rendered points. When moving the viewpoint through a point model, points are constantly streamed to and from the graphics card. This also means that the points used for the calculation of the point size heuristics constantly change. For example, after a fast movement, only few points will be available for rendering, and new points are thereupon streamed to the graphics card. The sequence in which the points of the nodes are fetched from disk is based on the nodes' importance for the current viewpoint, which means that the parent of a node might have to fetch the points after its child node has fetched them. While this is a desired behavior for adjusting the LOD in the areas close to the viewpoint (and in the center of the screen), it might lead to unwanted effects when using the point size heuristic while loading new points. Due to the irregular loading behavior with respect to the hierarchy levels (i.e., nodes are not loaded in level order), the virtual depth of a node can be increased or decreased several times, until all nodes required for the current viewpoint have been loaded. Therefore, the point sizes of these nodes change as well, which can lead to unwanted "flickering" in the rendered image, caused by the changing point sizes.

To alleviate those artifacts, the nodes can be loaded in different orders. For example, when loading nodes in level order, the point sizes for all rendered nodes constantly decrease. However, the points close to the viewpoint are at the lowest hierarchy level of all rendered nodes (due to the LOD), therefore, these nodes are loaded quite late when streaming in new points. This means that relatively large splats close to the viewpoint remain visible in the meantime. An improvement can be achieved by sorting the nodes of the individual levels by importance. But even then some time passes until the nodes of the lowest requested hierarchy level can be loaded to the graphics card.

Another method would be to sort the requested nodes by importance (ignoring the hierarchy level), and then only sort k % (e.g., k=95) of the least important nodes by level order, and all nodes within the individual levels by importance (as above). With this method, the nodes close to

the viewpoint are loaded first, but because not all nodes are loaded by level order, the flickering again becomes visible.

The flickering is most noticeable when rendering the points as screen-aligned square splats. When using circular splats, e.g., as used by the Gauss splats (see Section 4.4.2), the flickering becomes less obvious.

Alternatively, instead of calculating the point-size heuristic based on the currently rendered nodes, it can also be calculated based on all nodes that should be rendered for the current viewpoint (according to the minimum projected cell size). When using this approach, the point size of a node is fixed as long as the viewpoint does not move, and so flickering artifacts due to newly loaded points can be avoided. However, if nodes of some upper levels are loaded, whose child nodes will be loaded later, the nodes of the upper levels will already use the point sizes as if their child nodes were available. This can lead to somewhat undersampled areas, which will appear like holes in the point model.

While these undersampled areas are clearly visible when loading new points, the visual experience while loading the points is more comfortable with this approach, compared to the flickering artifacts that can occur when calculating the heuristic only based on the rendered points. This was also indicated by an informal user study, where the people working with point clouds at the institute were asked, which heuristic calculation approach was visually more comfortable, and the vast majority of the participants found the alternative approach (where the point sizes are calculated from all nodes that should be rendered for the current viewpoint) to be easier on the eyes.

3.6.3.2 Point Sizes for Nodes with Varying Point Densities

A hierarchy node can cover areas of different sampling densities, e.g., the points in one area of the node's volume might by sampled several times denser than the points in some other area. The branch of child nodes in the densely sampled area will be much deeper than the branch of child nodes in the less densely sampled area. The virtual depth of the parent node will therefore be mostly influenced by the child nodes in the densely sampled area. Since the virtual depth is calculated per node, this also means that the such calculated virtual depth will be too deep for the points in the less densely sampled area, i.e., the point size will be too small for these points.

Since this kind of artifacts is caused by the granularity on which the point-size heuristic is based (i.e., the heuristic is calculated per node), increasing the granularity (i.e., using smaller volumes) could alleviate or even eliminate these artifacts. This could for example be done by finding the leaf node a point would fall into, and calculate the heuristic for each point individually.

3.6.4 Results

Figure 3.15 shows a comparison of three different level weight calculation methods for the 20 levels (counted from 0 to 19) of the Domitilla catacomb model (see Section 3.11). Due to the distribution of the points in the hierarchy, up to level 13, the weight of the levels is very small. Depending on the calculation method, levels 14 (or 15) to 18 have the most points (and therefore weight), while level 19 again has only few points. When using the level weight based on the

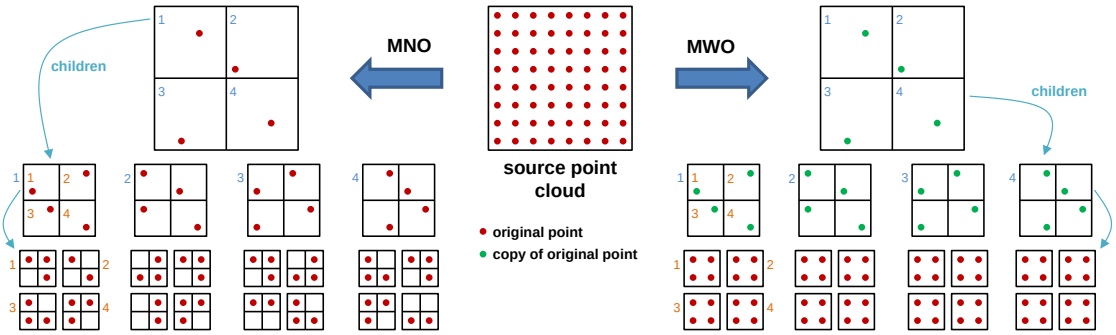


Figure 3.17: A source point cloud (center) represented as MNO (left) and MWO (right). Both point hierarchies use a 2x2 grid at the interior nodes. The MNO uses the grid also at the leaf nodes, while the MWO uses an array at the leaf nodes.

uniform sampling distribution (cyan graph), the distribution of the points is already shifted about one level. When using the exact level weight (red graph), the distribution of the points is further shifted down in the hierarchy levels, resulting in a smaller point size for rendered points of the upper levels. Also other point models that were tested have a similar distributions of points in the hierarchy.

Figure 3.16 shows the blending of the points of the different hierarchy levels. Points of level 9 are rendered smaller than points of level 8. Combining all levels results in a homogeneous representation of the zoomed out area.

3.7 Michael Wand Octree

As already described in Section 2.4, Wand et al. [107] introduced a multiresolution hierarchical data structure for rendering and manipulating huge point clouds. In this thesis, their proposed data structure, the Michael Wand octree (MWO), is compared to the MNO. Figure 3.17 shows a source point cloud in 2D, from which the 2D versions of an MNO (on the left) and of an MWO (on the right) are built. The MNO and the MWO store points at all nodes, but the MWO stores the original points only in the leaf nodes. Selecting points in an MWO is done with a selection octree (see Section 3.4).

In the following, inserting points into and deleting points from an MWO as well as rendering the points stored in an MWO will be described in more detail, as these operations will later be compared to the according operations of an MNO.

3.7.1 Inserting Points

Algorithm 3.6 shows how points are inserted into an MWO. The leaf nodes of an MWO can hold up to a predefined number of points m_{leaf}, e.g., 100,000 points. Each interior node holds a grid, and each grid cell holds a point. Furthermore, at each grid cell a 64 bit integer is maintained to count the points that fall into this cell. The grid is subdivided into 2^7 cells along each of the 3 space axes, so up to a total maximum of $(2^7)^3 = 128^3$ points can be stored at one interior node.

```
1  foreach point P in source point data do                    // Main loop
2  |  root node R->insert(P)
3  end
4  method node::insert(P):                                    // Method
5  |  if node is leaf then
6  |  |  insert P into array of points at node
7  |  |  iterate()
8  |  else
9  |  |  insert P into grid
10 |  |  if grid cell GC containing P is empty then
11 |  |  |  store a copy of P in GC
12 |  |  end
13 |  |  increment counter of GC by 1
14 |  |  appropriate child C->insert(P)
15 |  end
16 end
17 method node::iterate():                                   // Method
18 |  if num points in array > m_leaf then                   // m_leaf = max num points
19 |  |  create grid from points
20 |  |  foreach point P in array do
21 |  |  |  appropriate child C->insert(P)
22 |  |  end
23 |  |  delete array
24 |  |  convert this node to inner node
25 |  end
26 end
```

Algorithm 3.6: Inserting points into an MWO.

Usually, the number of points in an interior node is much lower for the data sets tested for this thesis. The points of the nodes are stored in one file per node. Leaf nodes store only point data, while the interior nodes store the point data and the counter for each grid cell in the file.

3.7.2 Deleting Points

Algorithm 3.7 shows how points are deleted from an MWO. The points which are going to be deleted have to be selected beforehand, which is done using a selection octree. All points in leaf nodes that are inside the volume of the selection octree are then removed from the MWO, and the hierarchy is checked for consistency.

Note that there is a subtlety involved when deleting the points. When inserting new points into the hierarchy, the representative points in the grid cells are chosen from the points inserted into a node, e.g., a copy of the first point that falls into a grid cell is used as representative point. When deleting points, it might happen that the representative point is the copy of a point that has

```
1  foreach leaf node N inside or intersecting SelOctree do
2      define array of deleted points D
3      if N is inside SelOctree then
4          put all points into D
5      else
6          put only points inside SelOctree into D
7      end
8      if all points of N are in D then
9          delete N from disk
10     end
11     foreach point P in D do
12         foreach parent node PN up to root node R do
13             decrease counter of grid cell containing P by 1
14         end
15     end
16     root node R->validate()
17 end
18 method node::validate():                                    // Method
19     foreach child C that is an inner node do
20         C->validate()
21     end
22     sum up number of points N in direct and indirect leaf node children
23     if N ≤ m_leaf then                                  // m_leaf = max num points
24         pull up all points from leaf nodes to this node
25         delete empty nodes from disk
26         convert this node to leaf node
27     end
28 end
```

Algorithm 3.7: Deleting points from an MWO.

been deleted, therefore the representative point is not covering the space of the remaining points. This can lead to problems during rendering when using certain types of hierarchy traversals. Section 3.8 proposes a method how these problems can be avoided.

3.7.3 Rendering

During rendering, a depth-first traversal chooses the nodes that will be rendered in the current frame. For this, the octree is traversed until the screen-projected size of a node is below a user defined threshold. This node and all of its siblings (inside the view frustum) are then chosen for rendering. Note that the screen-projected sizes of the siblings are not considered, as all siblings in the view frustum have to be chosen to completely replace the parent node (see also Section 3.8). Loading the nodes from disk happens asynchronously in a separate thread, so the parent

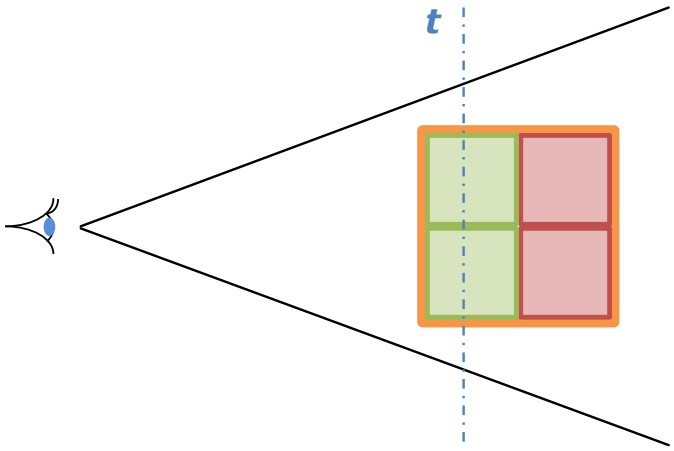


Figure 3.18: View frustum with an interior node of an MWO (orange bounding box) and its children. The front of the node has passed the threshold distance t, and the 2 green children have to be rendered. The 2 red children need not be rendered.

node is still rendered as a coarser LOD level until the children are available.

3.8 Michael Wand Octree Optimization

The MWO uses representative points in the interior nodes to enable a LOD mechanism for point clouds. During rendering, a depth-first traversal is used for choosing the nodes to render (see Section 3.7.3). This traversal tends to render more nodes than necessary for the current viewpoint. An example can be seen in Figure 3.18. The viewpoint is on the left, the orange square represents an interior node of the hierarchy, and the green and red squares represent the interior node's children. The threshold distance t is the distance at which the projected size of the orange node becomes larger than the allowed threshold (for the threshold, the projected side length of the node's bounding box is used). A child is thus rendered as soon as its projected double size is larger than the threshold (which is the size of the parent node at the child's position), and its projected size is still smaller than the threshold. In Figure 3.18, only the front half of the orange node has passed t, therefore only the 2 front-most children of the node fulfill the conditions to be rendered. The back half of the parent node is still beyond t, so the 2 back-most children need not be rendered, but in a depth-first traversal all children have to be rendered as soon as one child is in front of t.

Instead of a depth-first traversal, an importance-driven traversal can be used. The importance can be based on different parameters, e.g., the distance to the viewpoint, or how centered the node is on the screen. The importance-driven traversal stops when the maximum number of points that can be rendered is reached, or when no more nodes can be found which fulfill the conditions to be rendered. In such a traversal, no nodes will be rendered whose screen-projected size is too small. In Figure 3.18, only the 2 front-most children will be rendered together with the parent node, as its LOD representation still suffices to appropriately represent the point model

Figure 3.19: From the original point cloud (left image) a spherical volume is deleted. The center image shows the artifacts that remain when the interior nodes are not updated after deleting points. In the right image, the points in the interior nodes have been updated according to the deleted volume.

in the area of the 2 back-most children. This often reduces the number of nodes that have to be loaded during rendering, as can be seen in the results (Section 3.11).

Note that pixel overdraw occurs for both traversal strategies, depth-first and importance-driven, but for different reasons. In the depth-first traversal, a node is replaced by its child nodes as soon as one child node is close enough to the viewpoint to be rendered. This means that the points of the child nodes further away will also be rendered, even though their points are still too close together for projecting them to individual pixels, causing several points to be projected to a single pixel. In the importance-driven traversal, a parent node is rendered together with its child nodes as long as not all child nodes are chosen for the current viewpoint. While this causes pixel overdraw in the areas where the child nodes overlap with their parent node, the total number of rendered points is often smaller compared to a depth-first traversal, as the parent node contains (approximately) as many points as a child node. This means that as long as the parent node has to be rendered (this is the case as long as not all child nodes are rendered), the number of rendered points (of the parent node and the rendered child nodes) is not larger compared to rendering all child nodes.

For the importance-driven traversal, it is necessary that the points in the interior nodes of the octree actually represent the geometry of the point model, since interior nodes might be rendered at the same time as their child nodes. Otherwise, artifacts as shown in Figure 3.19 might occur. The center image is rendered with an importance-driven traversal, and the points in the interior nodes do not represent the actual geometry of the point model. This can happen, if the positions of the points in the interior nodes are not updated after deleting points from the leaf nodes. The image on the right shows no artifacts, and this can be achieved in two different ways, either by using a depth-first traversal, or by using an importance-driven traversal and updating the position of the points in the interior nodes during deleting.

During rendering, the depth-first traversal replaces a node with all its child nodes, as soon as the points in neighboring grid cells would not be projected to neighboring pixels. Therefore, the positions of the points in the interior nodes do not have to represent the actual geometry as they are replaced before the artifacts would become visible. Only for a short time, as long as the child nodes are loading, artifacts might be visible.

This means that depending on the effort spent when deleting points, different rendering

	Build Up	
Hierarchy	Time	Space
MNO	$O(N)$ to $O(Nlog(N))$	$O(N)$
MWO	$O(N)$ to $O(Nlog(N))$	$O(NC)$

Table 3.1: The time and space complexities for building MNO and MWO hierarchies.

traversals can (or have to) be used. Since an importance driven traversal is often favorable, the points in the interior nodes are replaced with points still available in the leaf nodes after deleting points from the hierarchy. For every grid cell of an interior node whose representative point is inside the selection octree (and the representative point's copy in the leaf node has therefore been deleted), a leaf node in the hierarchy is searched that intersects this grid cell, and a random point (that is also inside the grid cell's bounding box) of this leaf node is taken as new representative point. This step can be done in a separate traversal, after deleting points from the leaf nodes. The overhead due to this traversal is very low (see results in Section 3.11).

3.9 Complexity Analysis for In-Core Processing

The MNO as well as the MWO are designed for rendering and accessing large amounts of point data in short time. Search operations, on the other hand, are not as efficient compared to search operations in specialized data structures. The largest bottleneck is loading points from disk to memory, which can be alleviated to some degree by using an LRU cache. First, the behavior of the data structures will be analyzed assuming infinite memory, to exclude the effects of disk access, and afterwards their behavior including disk access is analyzed.

3.9.1 Build-Up

3.9.1.1 Modifiable Nested Octree

Building up an MNO is similar to building up a point octree, as both data structures store the points of the original data set in interior nodes as well as leaf nodes. While the point octree subdivides space at given data points, the MNO creates a hierarchy of congruent boxes, and each hierarchy level subdivides the space at the center of the previous hierarchy level. A point octree stores at each node exactly one point. The average build-up time for a point octree is proportional to $N\,log_8(N)$, where N is the total number of points used for building the octree. This estimation can be derived by calculating the total path length (TPL) [111] of a completely filled point octree. The TPL is defined to be the summed up path length when searching all nodes in a tree, and starting the search always at the root node. The path length between a node and one of its direct children is 1.

Like a point octree, an MNO stores points at interior nodes as well as leaf nodes. In an MNO, however, up to m (e.g., 128^3) points can be stored at each node. When using a hash table for storing the points, inserting a point into a node takes constant time on average [22]. The possible reduction is therefore dependent on the ratio m/N. If this ratio is larger or equal 1,

then the complexity becomes $O(N)$. For ratios approaching 0, the complexity again approaches $O(N\,log_8(N))$, which is equivalent to $O(N\,log(N))$ (see Table 3.1).

For point clouds from laser scanners, the number of points that are stored at each node is much smaller than m, since the points are distributed highly nonuniformly in space. Therefore, $m \gg N/k$, where k is the number of nodes in an MNO. Huge point clouds where $N \gg N/k$ show a complexity of roughly $O(N\,log(N))$, while smaller point clouds tend to show an $O(N)$ complexity. The space requirements for storing an MNO on disk are $O(N)$, since no additional points have to be saved.

3.9.1.2 Michael Wand Octree

Building up an MWO can be compared to building up a bucket PR octree, as both data structures store the points of the original data set only at the leaf nodes. The bucket PR octree uses the interior nodes to direct the traversal of the octree (see also Section 2.2.1), and each leaf node stores up to m_{leaf} points. The MWO stores additionally up to $m_{interior}$ points in the interior nodes for the LOD mechanism.

In an octree where the points are only stored at the leaf nodes, the time complexity for build up is $O(N\,log_8(N)) \Rightarrow O(N\,log(N))$, according to the total path length (see also Section 3.9.1.1). Using buckets at the leaf nodes, the complexity for building up an MWO changes depending on the ratio m_{leaf}/N. For ratios larger or equal to 1 the time complexity becomes $O(N)$, and for ratios approaching 0 the time complexity approaches $O(N\,log(N))$ (see Table 3.1). The points stored in the buckets at the interior nodes do not add to the complexity, as they are additionally created points that can be stored in a hash table with (on average) constant complexity. Furthermore, for every point that is inserted into the MWO, the counters in the grid cells of the interior nodes have to be updated, which can be done with constant complexity (on average) as well.

3.9.1.3 Storage Overhead of a Michael Wand Octree

The MWO has additionally created points stored at the interior nodes, which also have additional space requirements. An estimation for these additional space requirements can be useful to get an idea how much space is required to store a complete MWO. The additional space requirements do not only depend on the total number of points N in the original point cloud, but also on the distribution of the points in space, as this determines the number of necessary interior nodes of the MWO. As will be shown in the following, the number of interior nodes and subsequently the number of additionally created points can be estimated from the distribution of the points, as the distribution of the points determines the fan-out of the hierarchy nodes as well as the fill rate of the grids at the interior nodes.

For a point cloud representing a line, i.e., an object with dimensionality $d = 1$, the fan-out of the interior nodes will be similar to a binary tree, i.e., each node will have about 2 child nodes. Similar, for $d = 2$ and $d = 3$ the interior nodes will have about 4 and 8 children respectively. Assuming a grid with G cells per axis, then the grid has approximately G^d cells filled. Note that since the MWO uses point buckets at the nodes, their fan-out will rather represent the global characteristic of the data set. The granularity of the nodes is too big to capture the true dimen-

sionality. For a better estimation of the dimensionality, the points (i.e., number of occupied grid cells) within the nodes can be used.

When no assumptions can be made about the original point set, an initial estimate for d, without having previously built MWOs of similar point sets, is very difficult. On the other hand, if an MWO built from a similar point set is available, then the average number of points in its interior nodes, X, can be used as parameter for the approximation of the dimensionality. When X and G are given, then $d = log_G(X)$. From this value, an estimation for the fan-out F can be derived as $F = 2^d$.

When the total number of points N of a point set is also given, and an estimation for F and the related dimensionality d is available, then the space requirements for the interior nodes of a new MWO of a similar point set can be evaluated using the following 3 equations.

The total number of leaf nodes L can be estimated by using F as an approximation to the number of new leaf nodes that are created when a leaf node has to be split during build up. The maximum number of leaf nodes is then bounded by

$$L = \frac{N * F}{m_{leaf}} \tag{3.6}$$

and on average, each of the new leaf nodes will hold at least m_{leaf}/F points. Having an estimation for L, the total number of interior nodes I_{nodes} can be estimated with

$$I_{nodes} = \frac{L - 1}{F - 1} \tag{3.7}$$

which is the number of interior nodes for any tree with an average fan-out F. By using the dimensionality d together with G, the number of grid cells on one axis, the total number of points in the interior nodes I_{points} can be evaluted by

$$I_{points} = I_{nodes} * G^d \tag{3.8}$$

The space complexity can be written as $O(NC)$, where C is a constant depending on the estimated parameters d and F as well as on the given parameters G and m_{leaf}. For $L \gg 1$, the term $L - 1$ can be replaced by L and Equation (3.7) can thus be approximated by $L/(F - 1)$. With this simplification, C can be written as

$$C = 1 + \frac{F * G^d}{(F - 1) * m_{leaf}} \tag{3.9}$$

For example, a 2D object with $d = 2$ and $F = 4$ has a value of $C = 1 + (4/3) * (G^2/m_{leaf})$. When using the parameters suggested by [107], $G = 128$ and $m_{leaf} = 100,000$, then $C = 1.218$ for a two-dimensional object and $C = 24.97$ for a complete 3D volume. This means that for point clouds representing a single 2D area, the number of additional points I_{points} is estimated to be 21% of the number of original points, while for a point cloud representing a complete 3D volume, it is 24 times the number of original points. For the point clouds that were tested in this thesis with these parameters, the number of additional points was 40% of the original points, suggesting the dimensionality of the data sets is almost totally 2D.

	Search		
Hierarchy	Single Point	Sphere	Cuboid
MNO	$O(log(N))$	$O(R + 2^n M)$	$O(R + 4^n M)$
MWO	$O(M + log(N))$	$O(R + 2^n M)$	$O(R + 4^n M)$

Table 3.2: The time complexities for searching points in MNO and MWO hierarchies.

3.9.2 Searching

Searching for a single point in one of the two hierarchies means first searching for the node that holds the point, then searching for the point within this node, and then reporting the result. The complexities are listed in Table 3.2. The search for a point within an MNO node is of constant complexity $O(1)$, since the points in the nodes are stored inside grids (which are built once with an $O(N)$ complexity). For an MWO, the search for a point inside the leaf node is of $O(M)$ complexity, if no search data structure (like a kd-tree [9]) for the points in the leaf node has been created. Reporting a result can be done in $O(R)$ (e.g., copying the result to some memory address), where R is the number of reported points. When searching for a single point, the complexity for reporting the result can be omitted.

3.9.2.1 Modifiable Nested Octree

A range search in an MNO is similar to the range search in a PR quadtree, only in 3 dimensions. A PR quadtree subdivides the space hierarchically into congruent squares. All points are stored in its leaf nodes, one point per leaf node. A range search in a PR quadtree, where the range is a rectangular area limited by axis aligned lines, has a complexity of $O(R + 2^n)$, where R is the number of reported points and n is the maximum depth of the PR quadtree [92]. The 2^n term is derived from the maximum number of nodes that one line of the rectangle can intersect with. A PR quadtree in the 3D case becomes a PR octree. Searching all points in a PR octree that are inside an axis-aligned rectangular cuboid is proportional to $O(R + 4^n)$, where 4^n accounts for the lateral surfaces of the cuboid intersecting the octree nodes.

A range search in an MNO has to take interior nodes as well as leaf nodes into account. This means, all levels of the MNO contribute to the search algorithm complexity. This results in a performance proportional to $O(R + \sum_{i=0}^{n} 4^i) \Rightarrow O(R + 4^n)$. The complexity does therefore not increase for an MNO.

When using a sphere as the search range, it has to be discretized for counting the nodes intersecting the sphere. The number of nodes intersecting a discretized sphere is the solution to the diophantine inequation $(r - 1/2)^2 \leq x^2 + y^2 + z^2 < (r + 1/2)^2$, where $r \in \mathbb{N}$ is the radius of the discrete sphere, and $x, y, z \in \mathbb{Z}$ [5]. For increasing r it converges to $4\pi r^2$. The worst case complexity of the range search is found when using $r = 2^n/2$. This is the radius of the inscribing sphere for the axis-aligned bounding box of the root node, expressed in number of nodes at depth n. The complexity of this range search is thus proportional to $O(R + \sum_{i=0}^{n} 4\pi 2^{2i}/2^2) \Rightarrow O(R + \sum_{i=0}^{n} 2^{2i}\pi) \Rightarrow O(R + 2^n)$.

After identifying the nodes intersecting with the search range, all points within the intersecting nodes have to be tested if they are inside or outside the search range. This is of linear

	Delete Single Point		Delete Range	
Hierarchy	Leaf Node	Inner Node	Sphere	Cuboid
MNO	$O(log(N))$	$O(log(N))$	$O(2^n M log(N))$	$O(4^n M log(N))$
MWO	$O(M + log(N))$	n/a	$O(2^n M log(N))$	$O(4^n M log(N))$

Table 3.3: The time and space complexities for deleting points from MNO and MWO hierarchies. In an MWO, points in inner nodes are only representatives of the points in the leaf nodes. Therefore, points are not directly deleted from inner nodes.

complexity $O(M)$ per node, where M is the maximum number of points in a node. In total, the range search complexity for a cuboid inside an MNO is $O(R + 4^n M)$, and for a sphere it is $O(R + 2^n M)$.

3.9.2.2 Michael Wand Octree

During a range search in an MWO, points only have to be searched for in the leaf nodes, contrary to an MNO, but since the upper levels of the MNO do not contribute to the search complexity, both hierarchies have the same complexities (see Table 3.2).

3.9.3 Deleting

Deleting a single point means basically searching for it, and on the average this has a time complexity of $O(log(N))$ (see Table 3.3). In an MWO, points are always deleted from the leaf nodes, and the interior nodes are updated later. In an MNO, points are directly deleted from leaf nodes and interior nodes. Restoring the consistency of the LOD hierarchy after deleting points does not add to the complexity.

3.9.3.1 Modifiable Nested Octree

In a node, the search for the grid cell a point falls into can be done in $O(1)$. When a point is deleted from a leaf node, the LOD structure remains intact. When a point is deleted from an interior node, two more steps are required, i.e., finding a replacement point and pulling it up to the node where the other point has been deleted from.

A replacement point is searched for in a leaf node that encompasses (or is inside of) the bounds of the node's grid cell the other point was deleted from (with $O(log(N))$ complexity). This replacement point is then deleted from the leaf node and inserted into the node the other point was deleted from (with $O(1)$ complexity). Therefore, the total complexity of deleting a point from an interior node is $O(2\, log(N)) \Rightarrow O(log(N))$.

Deleting a range of points also includes first searching for the points, then deleting them, and finally restoring the LOD hierarchy. In Table 3.3, the complexities for deleting points inside a spherical and cuboid volume are given. The complexities are similar to the complexities for searching points inside such volumes, but the points do not have to be reported, so the R term can be omitted. But a $log(N)$ factor has to be added, to account for restoring the LOD hierarchy, which has to be done for every deleted point.

3.9.3.2 Michael Wand Octree

The search for a point in a leaf node is done in linear time $O(M)$, where $M = m_{leaf}$. After deleting a point from a leaf node, the interior nodes on the path from the leaf node to the root node have to update the counters of the grid cells into which the point falls (with $O(log(N))$ complexity). The total complexity for deleting a point is $O(M + 2\,log(N))$. When using the optimization of Section 3.8, the points of the interior nodes inside the selection octree have to be updated. They have to look for a replacement point in an existing leaf node. With this additional step, the total complexity for deleting a point is $O(M + 3\,log(N)) \Rightarrow O(M + log(N))$.

Since the points in the interior nodes are only representatives of the points in the leaf nodes, they are not directly deleted, but only when the according points in the leaf nodes have been deleted. Therefore, no complexity is given for deleting points from the interior nodes.

Deleting a range of points from an MWO is similar to deleting a range of points from an MNO, as in both cases the interior nodes have to be changed. The MWO has to update the counters (and also the points when using the optimization mentioned above) in the interior nodes, which are the (direct or indirect) parent nodes of the nodes where points are deleted from. Therefore, the MWO has the same complexity as the MNO for deleting a range of points.

3.10 Complexity Analysis for Out-of-Core Processing

Having only a limited amount of main memory, the point data can become larger than the available main memory, and an out-of-core management for the data becomes necessary. In this case, the points of the hierarchy nodes have to be streamed from disk during processing. Accessing the disk has a large overhead compared to memory access (see Section 2.2.2), therefore reducing the number of disk accesses is important for the efficiency of the out-of-core data management. In the worst case, e.g., if the locality of an operation on the point cloud is bad, each access to a hierarchy node is preceded by a disk access.

One strategy to reduce the number of disk accesses is to use an LRU cache, which manages nodes according to the time they were last used. If points for a new node have to be loaded from disk, the points of the node which has been accessed the longest time ago are swapped out of memory. Generally, using a memory caching method can increase performance a lot. The parameters influencing the efficiency of an LRU cache are the size of the cache and the access pattern of the operations on the nodes of the point-cloud hierarchy.

In the following, the three basic operations that were examined for in-core processing for the MNO and MWO (inserting points for build-up, searching points, and deleting points) are examined for out-of-core processing.

3.10.1 Build-Up

The complexities for building up a point-cloud hierarchy with out-of-core managed nodes are given in Table 3.4, once for the worst case and once for the average case. D is a huge constant representing disk (or more generally, external memory) access. In the worst case, every point has to be written to disk separately, which is expressed by the term ND. Note that actually all points of a node that a point belongs to have to be written to disk, not only the single point, but since

	Build Up	
Hierarchy	Worst Case	Average
MNO	$O(ND + ND\,log(N))$	$O(kD + N\,log(N))$
MWO	$O(ND + ND\,log(N))$	$O(kD + N\,log(N))$

Table 3.4: The time complexities for the out-of-core build up of MNO and MWO hierarchies.

	Search Single Point	
Hierarchy	Worst Case	Average
MNO	$O(D + D\,log(N))$	$O(D + log(N))$
MWO	$O(D + M + D\,log(N))$	$O(D + M + log(N))$

Table 3.5: The time complexities for out-of-core searching a single point in MNO and MWO hierarchies.

the points of a node are written in a continuous block, it is counted only as one external memory access. The term $ND\,log(N)$ accounts for loading the necessary nodes from disk when inserting a point into the hierarchy, because in the worst case, all traversed nodes have to be loaded from disk for every inserted point.

In the average case, the hierarchy nodes will be written do disk much less often, since during build-up usually some memory caching technique will be used, and the distribution of the inserted points will not be completely random (e.g., when using point sets coming from laser scanners). This is expressed by the term kD, where k is the number of nodes in the hierarchy, meaning that the number of disk accesses is in the order of magnitude of nodes in the hierarchy (and not in the order of magnitude of points in the data set, as in the worst case). Furthermore, the term $N\,log(N)$ accounts for traversing the hierarchy when inserting points. Since the nodes are cached in memory, and since successively inserted points often fall into the same nodes (due to a partially sorted input point set), it is most of the time not necessary to load nodes from disk when traversing the hierarchy for inserting new points. Therefore, on average, the disk access can be neglected for the hierarchy traversal.

A performance increase can be achieved by using SSDs instead of disks for the external memory, as this will reduce access times and increase the throughput for the external memory, resulting in a smaller value for the constant D.

3.10.2 Searching

The complexities for searching a single point are given in Table 3.5. The worst case occurs when all traversed nodes have to be loaded from disk during the search. This is expressed with the $D\,log(N)$ term. The MWO has to linearly search the leaf nodes if they contain the searched for point, and this is expressed, as for in-core searching, with the M term. Besides searching for the point, the result also has to be reported. Contrary to the in-core case, where the reporting only adds a small constant, reporting in the out-of-core case adds a disk access D, which cannot be omitted. In the average case, it can be assumed that all nodes that have to be traversed for the

	Search Sphere	
Hierarchy	Worst Case	Average
MNO	$O(RD + 2^n MD))$	$O(kD + R + 2^n M)$
MWO	$O(RD + 2^n MD))$	$O(k_{leaf} D + R + 2^n M)$

Table 3.6: The time complexities for out-of-core searching points in a spherical volume in MNO and MWO hierarchies.

	Delete Single Point	
Hierarchy	Worst Case	Average
MNO	$O(D\,log(N))$	$O(D + log(N))$
MWO	$O(M + D\,log(N))$	$O(D + M + log(N))$

Table 3.7: The time complexities for out-of-core deleting a single points from MNO and MWO hierarchies.

	Delete Sphere	
Hierarchy	Worst Case	Average
MNO	$O(2^n MD\,log(N))$	$O(kD + 2^n M\,log(N))$
MWO	$O(2^n MD\,log(N))$	$O(kD + 2^n M\,log(N))$

Table 3.8: The time complexities for out-of-core deleting points in a spherical volume from MNO and MWO hierarchies.

search reside in the LRU cache. Nevertheless, the result has to be reported, therefore also in this case a disk access D is added to the complexity.

For the out-of-core range search, only the complexities for the spherical search are given in Table 3.6, as the search in a cuboid can be derived analogously. In the worst case, each of the 2^n nodes that are searched has to be loaded from disk. The linear search for points intersecting the sphere is accounted for with the term M, and loading each node from disk is accounted for with the term D. Reporting each point that was found is done in the worst case with RD complexity, which includes all points inside the spherical search volume. In the average case, disk access for the range search can be described by the term kD, with k being the number of nodes in a hierarchy, which includes loading them from disk for searching, and reporting the result. Since range searches can cover large parts of the point cloud, causing points to be swept out of and into memory, the disk access cannot be omitted. For the MWO, the factor k can be reduced to the number of leaf nodes, k_{leaf}. This can be done, since the points of the interior nodes are not needed for the search.

3.10.3 Deleting

The complexities for deleting a single point are given in Table 3.7. The worst-case complexity for deleting a single point is similar to the worst-case complexity of searching a single point, as the additional steps required to replace the deleted point in the inner nodes are covered by the

$D\,log(N)$ term, and therefore do not add to the complexity. Since no result has to be reported, the single D term can be omitted. The average-case complexity for deleting a single point is the same as for searching a single point, but the term D is required for a different reason. When deleting a single point, the term D accounts for writing the changed nodes back to disk.

Deleting a range of points from one of the two hierarchies is similar to searching for a range of points, as in both cases only the nodes on the border of the search volume have to be processed exactly, i.e., each point of the nodes on the border has to be checked if it is inside or outside the search volume. In Table 3.8, the worst-case and average-case complexities for deleting points from a spherical volume are given. The term $2^n M$ accounts for checking the points on the border of the search volume (if they are inside or outside), and the term $log(N)$ for finding replacement points to keep the LOD hierarchy consistent. In the worst case, the factor D accounts for the disk accesses that are necessary before every access to a point. In the average case, the term kD accounts for all disk accesses during deleting, with k being again the number of nodes in the hierarchy.

3.11 Results

In this section, the test results for several operations on the MNO and the MWO data structures are presented. At first, the build-up performance for the MNO is analyzed, using different point clouds and attributes per point. Next, the in-core build-up performance of the MNO and the MWO are compared, followed by a comparison of the out-of-core build-up performances for these two data structures. Afterwards, the performances for deleting points from these two data structures are analyzed. Then, the effect of the MWO optimization after deleting points is investigated, and finally the build-up performance of the selection octree is tested.

Four different point models were used for benchmarking the performance of the different operations on the MNO and MWO data structures. In Figure 3.20, these four point models are depicted. At the very left is the Santa Claus model (about 16*10^6 points), followed by the Hanghaus model (about 15*10^6 points), next is the model of the Stephansdom, the largest cathedral in Vienna (about 460*10^6 points), and at the very right is the model of the Domitilla catacomb in Rome (about 1.92*10^9 points).

For the benchmarks, three different test systems were used. The first system is a desktop computer with an Intel Core2 Q6600 CPU with 2.4 GHz, 4 GB RAM, a RAID0+1 with 4 SATA hard disks running at 10,000 RPM, and a GeForce 8800 GTX graphics card with 768 MB VRAM.

The second system is a desktop computer with an Intel Core i7-2600K quad core CPU with 3.4 GHz, 16 GB RAM, a RAID0 with 2 SATA hard disks running at 10,000 RPM, a RAID0 with 4 SSDs, and a GeForce GTX 580 graphics card with 3072 MB VRAM.

The third system is a laptop computer with an Intel Core i7-2720QM CPU with 2.2 GHz, 8 GB RAM, a hard disk running at 7200 RPM, and a GeForce GT 555M graphics card with 3GB VRAM

In the following, the different test systems will be referred to as test system 1, 2, and 3 respectively.

Figure 3.20: The scans used for benchmarking the build-up algorithm and the editing operations. Left: Santa Claus model, consisting of about 16M points. It was scanned with a scan-arm. Center left: Model of the "Hanghaus" in Ephesos, consisting of about 15M points. Center right: The Stephansdom in Vienna, consisting of 193 single scans and about 460M points. Right: A view of the subterranean Domitilla catacomb in Rome, hovering below the turf. This model consists of 1828 single scans and about 1.92 Billion points.

Model	Attribs	AttsSize	Time	Build Time	Size on disk	Throughput
Santa Claus	p,c	16B	3m 25s	13.6s	255.0 MB	18.7 MB/s
Santa Claus	p,c,n	28B	4m 58s	13.9s	445.6 MB	32.0 MB/s
Ephesos	p,c	16B	34s	11.6s	235.9 MB	20.3 MB/s
Ephesos	p,c,n	28B	35s	12.1s	412.4 MB	34.0 MB/s

Table 3.9: The build-up times and sizes for the resulting models for the Santa Claus point cloud (16M points) and the Ephesos house (15M points) built with the MNO build-up algorithm. The "AttsSize" column shows the number of bytes that are used for the attributes of one point. The "Time" column shows the complete processing time, including reading the data files and converting them to a binary data stream. The "Build Time" column shows the build-up time alone. The models are built in-core. The size of the resulting models is larger than the added up size of the points alone due to padding at the disk sectors and the header file that holds the information about the octree hierarchy. The "Attribs" column shows which attributes were used for build up (p = position, c = color, n = normal).

3.11.1 Build-Up Performance MNO

The build-up performance of the MNO was benchmarked with test system 1. The 4 point models were built up with different per-point attributes to evaluate the performance characteristics of the build-up algorithm. The possible attributes were position, color, normal, point radius, point index, and scan position index. In total, these attributes use 44 bytes per point. The tests were performed with all or a subset of these attributes. When reading the point data from the source files and converting them to a binary stream, it is possible to choose which attributes the points in the binary stream should have. If an attribute is not available in the source data, it is added for each point and filled with a default value. In the resulting binary stream, all attributes that a user has chosen are available, and from this binary stream an MNO is then built. The attributes position, color, and normal are so-called "meta-attributes" in the point rendering system, because for these meta-attributes it is possible to choose which representation they should have on disk and in memory. When loading points from disk during rendering, the meta-attributes can be

Model	Attribs	AttsSize	Time	Build Time	Size on disk	Throughput
Steph	p,c	16 B	22m 35s	18m 10s	7,390 MB	6.78 MB/s
Steph	p,c,n	28 B	37m 51s	32m 52s	12,922 MB	6.55 MB/s
Steph	p,c,n,r,i,s	44 B	1h 43m 40s	1h 37m 13s	20,287 MB	3.43 MB/s
Domitilla	p,c	16 B	1h 27m 15s	1h 05m 01s	30,796 MB	7.89 MB/s
Domitilla	p,c,n	28 B	1h 48m 43s	1h 21m 24s	53,855 MB	11.03 MB/s
Domitilla	p,c,n,r,i,s	44 B	2h 25m 29s	1h 55m 23s	84,600 MB	12.22 MB/s

Table 3.10: The build-up times and sizes for the resulting models for the Stephansdom and Domitilla point clouds. Here, the "AttsSize" column shows the number of bytes that are used for all attributes of one point. The "Time" column again shows the complete processing time, including reading the data files and converting them to a binary data stream. The "Build Time" column shows the build-up time alone. "Size on disk" shows the size of the resulting model for out-of-core build up using the MNO build-up algorithm. "Build Throughput" shows the number of bytes per second that are processed during the build-up stage. The Stephansdom model has a lower throughput then the Domitilla model, which is due to the large areas that the single scans cover in the Stephansdom model. Table 3.11 shows that this results in more LRU cache misses. Finally, the "Attribs" column shows which attributes were used for build up (p = position, c = color, n = normal, r = point radius, i = point index, s = scanposition index).

Model	Attribs	LRU size	MNO nodes	Nodes to mem	Nodes to mem %
Steph	p,c	1,708 MB	43,023	26,285	61%
Steph	p,c,n	1,708 MB	43,023	49,154	114%
Steph	p,c,n,r,i,s	1,708 MB	43,023	123,283	286%
Domitilla	p,c	1,708 MB	200,159	46,085	23%
Domitilla	p,c,n	1,708 MB	200,159	65,902	33%
Domitilla	p,c,n,r,i,s	1,708 MB	200,159	100,930	50%

Table 3.11: Here the percentage of the nodes (of the final model) that are swapped back from disk to memory are given. The columns "LRU size", "MNO nodes", "Nodes to mem", and "Nodes to mem %" are showing the LRU cache sizes in megabytes, the total number of nodes in the final MNO, the number of nodes swapped back to memory during build up, and the percentage of swapped back nodes with respect to the nodes of the final MNO, respectively. The single scans in the Stephansdom model contribute to larger areas of the model and are also sampled at a higher density, which results in more cache misses in the build-up LRU cache. Due to this, more nodes have to be swapped out of and back to memory. The "Attribs" column shows which attributes were used for build up (p = position, c = color, n = normal, r = point radius, i = point index, s = scanposition index).

converted to a specific representation that is suitable for rendering, or they can be passed to the graphics card without conversion.

3.11.1.1 In-Core Performance

The Santa Claus model and the Ephesos model were used to test the in-core performance of the build-up algorithm, as they fit completely in the main memory of the test computer. The models have position, color, and normal as attributes (using 28 bytes per point). The data for the Santa Claus model was stored in a text file, while the data for the Ephesos model was stored in a binary file.

In Table 3.9, the results for the in-core build-up algorithm are shown. Reading the data from a text file takes much longer than reading the data from a binary file, therefore the complete processing time for the Santa Claus model is several times higher than for the Ephesos model, while the build times for both models are about equal. The data throughput is much higher when normals are also part of the used attributes (about 30 MB/s compared to about 20 MB/s without normals). The normals add 12 bytes (or 75% more data) to the attributes of each point, but the build times are only slightly higher, therefore the traversal of the octree structure seems to be the main limiting factor when inserting points in-core. The algorithm can insert $1.21*10^6$ points per second when using only position and color as attributes (using 16 bytes per point) and $1.17*10^6$ points per second when using a normal (requiring 12 additional bytes) per point as well (these numbers are averaged over both models).

3.11.1.2 Out-of-Core Performance

For testing the out-of-core performance, the Stephansdom model and the Domitilla model were used, together with test system 2. At most 10% (or 1.7 GB) of the main memory were used for the LRU cache. Table 3.10 shows a comparison of the build-up times of these two models with different point attributes. The first two variants of the Stephansdom model (with 16 and 28 bytes per point) show a similar build-up performance in the throughput (6.78 and 6.55 MB/s), while the third variant (with 44 bytes per point) shows only half of the throughput (3.43 MB/s). This seems to be caused by the increased memory usage of the points with the additional attributes, so that nodes have to be swept into and out of the LRU cache more often. Table 3.11 shows how many nodes relative to the total number of nodes per model have to be swapped into memory again (after they have been dropped during build up). There, the third variant of the Stephansdom model has the highest percentage of nodes (relative to the total number of nodes) that has to be swept back from disk. Genereally, all variants of the Stephansdom model show a higher percentage in this category than the variants of the Domitilla model, indicating that the point density of the Stephansdom model is higher. Another reason might be that the single scans of the Domitilla catacomb contribute to more limited areas of the model, due to the narrow hallways, while the scans of the Stephansdom are covering larger distances. This results in less LRU cache misses when building the Domitilla model, and less disk accesses during build up.

Large Build-Up Cache with 100M Points					
Sorting Type		*Sorting*	*Merging*	*Build-Up*	*Total*
No Sorting		-	-	20h 29m 20s	20h 29m 20s
Axis	HSS:	9m 46s	3m 38s	21m 06s	34m 30s
	HPS:	4m 14s	4m 13s		29m 33s
	HSP:	9m 50s	5m 59s		31m 55s
	HPP:	4m 10s	10m 10s		35m 26s
	RSS:	5m 50s	3m 36s		30m 32s
	RPS:	3m 51s	4m 04s		29m 01s
	RSP:	6m 09s	5m 58s		33m 13s
	RPP:	4m 11s	10m 55s		36m 12s
Morton Order	HSS:	19m 18s	4m 43s	15m 46s	39m 47s
	HPS:	5m 42s	6m 28s		27m 56s
	HSP:	18m 25s	6m 10s		40m 21s
	HPP:	5m 40s	11m 16s		32m 42s

Small Build-Up Cache with 10M Points					
Sorting Type		*Sorting*	*Merging*	*Build-Up*	*Total*
No Sorting		-	-	36h 43m 36s	36h 43m 36s
Axis	HSS:	9m 46s	3m 38s	> 11d 2h	> 11d 2h
	HPS:	4m 14s	4m 13s		> 11d 2h
	HSP:	9m 50s	5m 59s		> 11d 2h
	HPP:	4m 10s	10m 10s		> 11d 2h
	RSS:	5m 50s	3m 36s		> 11d 2h
	RPS:	3m 51s	4m 04s		> 11d 2h
	RSP:	6m 09s	5m 58s		> 11d 2h
	RPP:	4m 11s	10m 55s		> 11d 2h
Morton Order	HSS:	19m 18s	4m 43s	18m 55s	42m 56s
	HPS:	5m 42s	6m 28s		31m 05s
	HSP:	18m 25s	6m 10s		43m 30s
	HPP:	5m 40s	11m 16s		35m 51s

Legend	*H...Heap Sort*	*S...Serial Sort*	*S...Serial Merge*
	R...Radix Sort	*P...Parallel Sort*	*P...Parallel Merge*

Table 3.12: Sorting and build-up times for the Stephansdom model.

Figure 3.21: The Stephansdom model used for benchmarking the external sorting algorithms. It contains about 675*10^6 points.

3.11.1.3 Out-of-Core Performance with External Sorting

The performance of the external sorting variants was measured with test system 2. In the cases where parallel sorting or parallel merging were used, all 8 cores (4 cores + 4 hyperthreaded cores) of the i7-2600K CPU were used, so 8 threads could process the data in parallel. The test data set was a point model of the Stephansdom in Vienna, consisting of about 675*10^6 points, which can be seen in Figure 3.21. It uses the original samples from the laser scans and is not cleaned in any way. Most areas in the point model are sampled from several different scan positions. For example, a scan position at the entrance, inside the cathedral, will also take samples from the apsis (where the high altar is placed) at the other end of the cathedral. Such point models, where areas are sampled from several different (and far apart) scan positions, are very problematic to handle for an out-of-core build-up algorithm, since the nodes of the MNO will hold points from different scan positions, and therefore points often have to be swept out of and into memory again. When sorting the points before inserting them into an MNO, it should be possible to improve the performance of the build-up algorithm.

The total size of the chunks loaded to main memory was limited to 4GB during external sorting. After sorting, the points were inserted into a freshly built MNO. For this task, the main memory LRU cache size was once set to 10 million points and once to 100 million points respectively. The LRU cache stores the points of the nodes during the MNO build-up. If locality in the sorted point data is well exploited, a small LRU cache size could suffice for an efficient build-up performance.

The performance of the sorting algorithms was tested in combination with the build-up times from the sorted data sets, because a fast sorting algorithm can still result in a bad total performance, if the sorting is not adapted to the available hardware, for example to the available

memory size. This can be seen in Table 3.12, when using a size of 10 million points for the build-up LRU cache. Although sorting the points along an axis is often faster than sorting points according to Morton order, the locality of the points in the data set sorted along an axis is worse. It is even worse than the locality in the data set that uses the points coming directly from the laser scans, as in this data set subsequent points often sample the same area, which causes chunks of subsequent points often fall into the same MNO node. When sorting the points along an axis, it can happen that subsequent points are distributed quite randomly (within some distance to a plane orthogonal to the sorting axis), and therefore they might fall into nodes far apart in the MNO. This can trigger loading the nodes of a whole MNO branch from disk for a lot of the points that are inserted, which reduces the performance already a lot. An additional problem is the fragmentation occurring during the build up of the MNO. Due to the high number of necessary writes to disk, the majority of the files holding the points becomes fragmented (more than 80% during the test). Fragmentation of the files could be reduced by increasing the allocation unit size (i.e., the minimum number of bytes a file allocates on disk) of the disk. This way, more points could be stored per allocation unit, and the files would be split into a smaller number of fragments.

Due to the before mentioned issues, the build up of the MNO when using the point data set sorted along an axis actually takes longer (more than 11 days) than the build up of the raw data set when using the small LRU build-up cache, while the Morton order sorted data set can be built up with almost the same performance as when using the large build-up cache. The build-up time when sorting the points along an axis is estimated, because the build up was stopped after running for 5 days. The insertion rate of the points during build up was slowly decreasing due to the fragmentation of the files, and it was at about 708 points/second when the build up was stopped.

When using a large build-up cache, then building the MNO after sorting the points by Morton order or along an axis show approximately the same performance. Here, enough points can be stored in the LRU cache, so not a lot of nodes have to be loaded from disk, even if subsequent points are not stored in the same node, which is in contrast to the case when using the small build-up cache. In both cases, the variant with parallel sorting and serial merging is the fastest.

The sorting of the in-core chunks was tested with heap sort and (LSD) radix sort. In the merging step, the bottleneck is loading the elements from disk. When using the parallel merging variant, where several threads are accessing the external memory, the slow disk accesses reduce the performance. The worst case occurs, when parallel sorting and parallel merging are used together, as parallel sorting creates more files than serial sorting. So when parallel merging those files afterwards, the slow disk access becomes even more noticeable.

3.11.2 Rendering Performance MNO

The rendering performance of the MNO was tested with three different point models, namely the Domitilla model, the Stephansdom model, and the model of the Amphitheater 1 in Bad Deutsch Altenburg, which contains about 106 million points (see also Section 4.2). All point models used position and color as point attributes, which require 16 bytes per point for storage. Figures 3.22 to 3.24 show the FPS for moving the viewpoint along camera paths through the three models, performed on test systems 2 and 3, which are referred to as desktop and laptop respectively in the

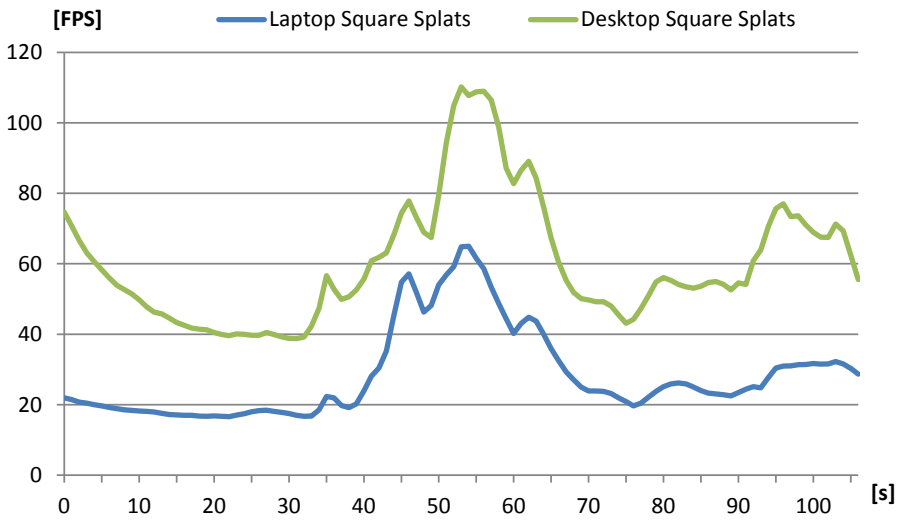


Figure 3.22: The graphs shows the frames per second for the camera path through the Domitilla model, performed on two different test systems.

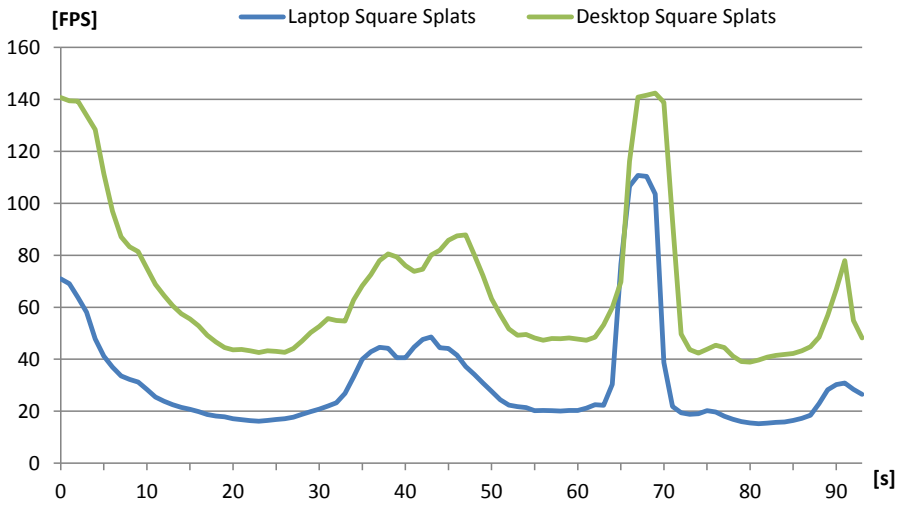


Figure 3.23: The graphs shows the frames per second for the camera path through the Stephansdom model, performed on two different test systems.

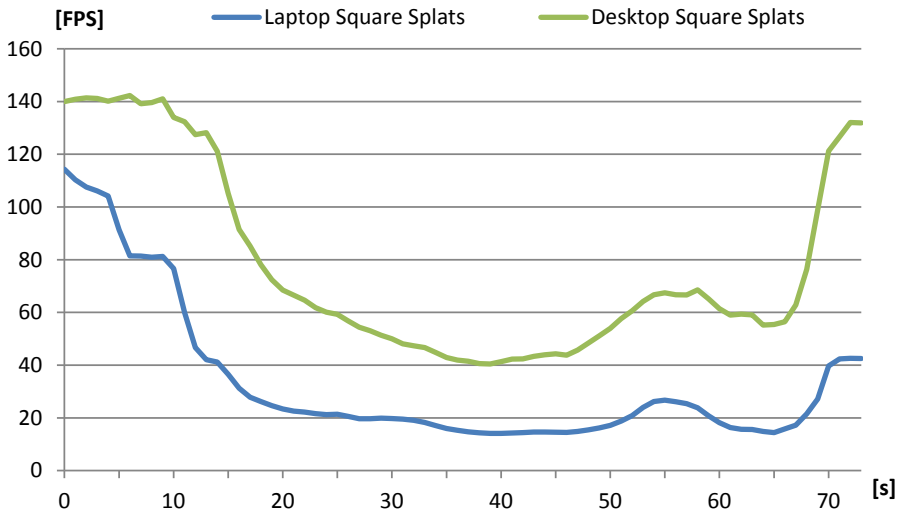


Figure 3.24: The graphs shows the frames per second for the camera path through the Amphitheater model, performed on two different test systems.

figures. The point renderer was set to draw at most 25 million points per frame. The minimum projected size of a cell (of a node's grid) was set to 1 pixel, and the weighted point size was used for calculating the splat sizes on screen. The splats were then drawn as screen-alined square splats. The viewport size was set to 1280 x 720 pixels.

As can be seen in the figures, the minimal frame rate for test system 2 is about 40 FPS, while for test system 3 it is about 15 FPS. These frame rates are achieved when rendering 25 million points. The highest frame rates (about 140 and 110 respectively) are achieved when the viewpoint moves away from the point models, to get an overview, and only a few nodes (with approximately 700,000 points in total) have to be rendered. The camera paths usually start at some distance from the point models, and then approach the models. In the last frames of the camera paths, the frame rates stay constant or even drop, which is due to the nodes that are loaded as soon as the movement stops.

3.11.3 Out-of-Core Performance MNO

In this section, the out-of-core behavior of the MNO during rendering is analyzed. First, the influence of the secondary thread on the frame rate is tested (the secondary thread loads the points from external memory). For this, the viewpoint is moved along the camera path through the Amphitheater model twice, once starting with an "empty" LRU cache (i.e., no points are in the cache, except for the ones required to render the point model from the starting position of the camera path), and once with a pre-filled LRU cache (i.e., all points that will be used for rendering during the camera movement are already available in the cache). Figure 3.25 shows a comparison of the FPS for these two cases, for which test system 2 was used. The parameters are the same as for the tests in Section 3.11.2, e.g., the minimum projected grid cell size is set to 1 pixel. The FPS are in both cases almost identical, meaning that loading the nodes from external

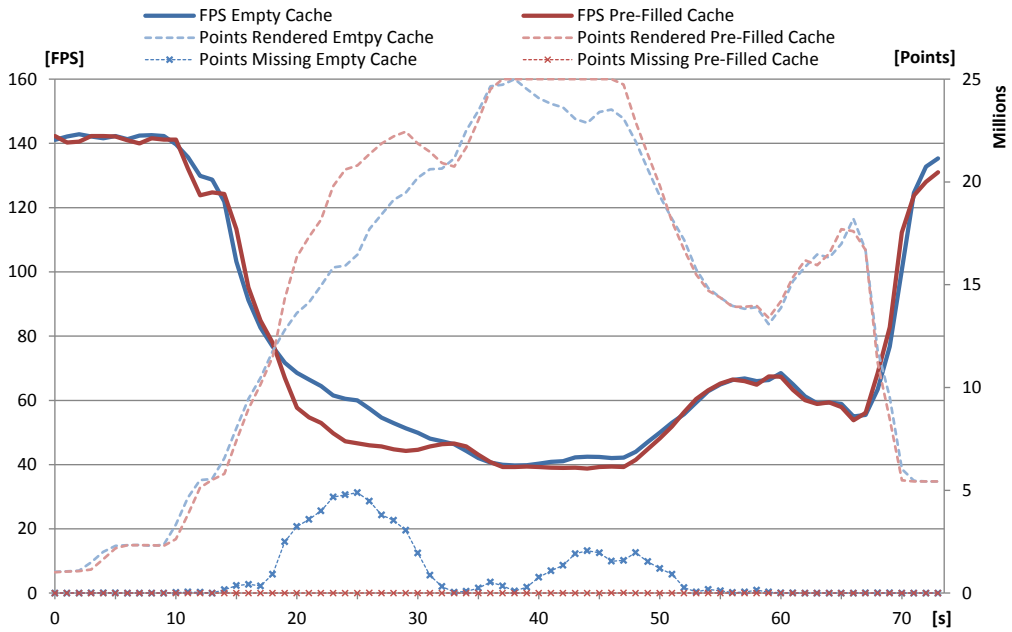


Figure 3.25: The two solid-line graphs show a comparison of the frames per second for the camera path through the Amphitheater model, once performed with an initially empty cache (blue), and once performed with a pre-filled cache (red). The two dashed graphs show the average number of points rendered per frame. At maximum, 25 million points can be rendered per frame. The two dashed graphs with cross markers show the average difference between the number of points rendered and the number of points that should be rendered (according to the LOD) per frame.

memory has no impact on the FPS during rendering. An exception is the time interval between second 20 and 30. Here, the frame rate for the walkthrough with the initially empty cache (blue graph) is even higher. In this time interval, the camera moves into a denser sampled area, so a lot of new points have to be loaded to the GPU, which can be seen at the dashed graphs, showing the number of points rendered per frame. Since loading the points to GPU takes longer when they have to be fetched from disk than from the LRU cache, the walkthrough with the initially empty cache has a higher frame rate there. Furthermore, the quality of the rendered image is influenced when loading the points from disk, as not all points are available for the appropriate LOD level, which can be seen at the dashed graphs with the cross markers. When loading the points from disk (blue graph with cross markers), several times the number of available points lags the number of requested points, so the point model has to be rendered at a coarser LOD. However, when loading the points from cache (red graph with cross markers), they are almost always available in time. In Figure 3.26, the two graphs showing the difference in the number of required and rendered points are given on a logarithmic scale. When using a pre-filled cache, the difference in the number of points is up to 3 orders of magnitude smaller (e.g., around second 25 and 45) compared to using an empty cache. Therefore, due to the pre-filled cache, only a few points are missing at any time during rendering.

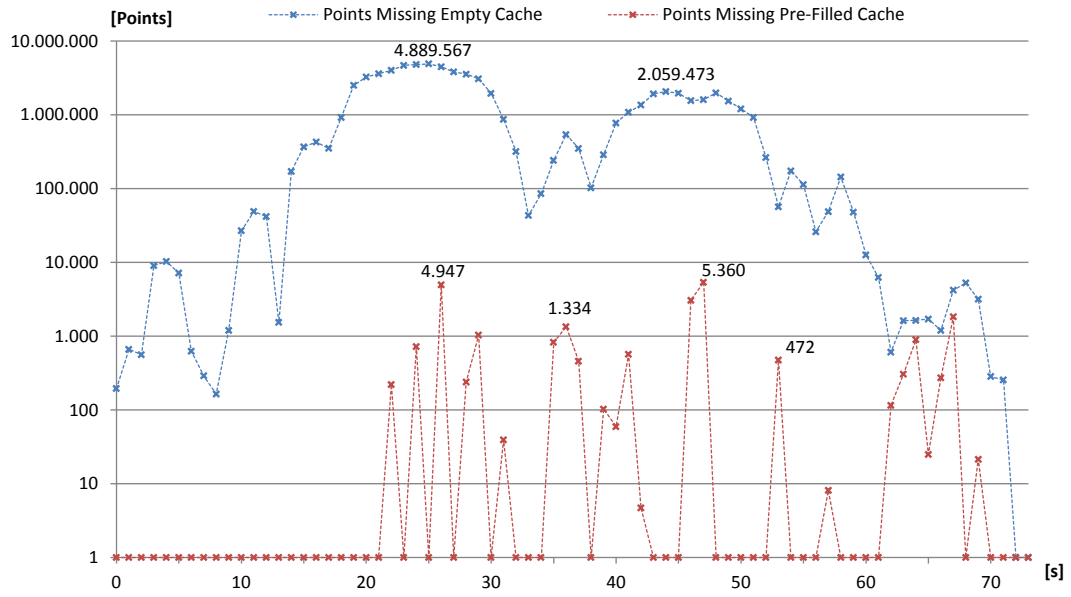


Figure 3.26: The two graphs show the same difference between the number of points rendered and the number of points that should be rendered as the two dashed graphs with cross markers in Figure 3.25, but on a logarithmic scale. In the chart, the values for some local maxima are given. When using a pre-filled cache for the walkthrough, the difference in the number of points is up to 3 orders of magnitude smaller compared to using an empty cache.

The next test deals with the time required to load the points from disk. For this test, again test system 2 was used. Figure 3.27 shows two graphs, which were recorded when loading about 1000 nodes from disk (for this, the viewport was set abruptly into the center of the Domitilla model). Before the test, the LRU cache was emptied. The blue graph shows the time required to load the points for the individual nodes, while the red graph shows the number of points per node. There is almost no correlation between the loading time and the number of points per node (the Pearson correlation coefficient for the test series is 0,18), meaning that the access to the data on disk has the most influence on the loading time. The average (median) time for loading the points of a node from disk to main memory (measured from the request for the points until they are available to be accessed from the main thread) is about 40 milliseconds. The two peaks in the blue graph are unrelated to the number of points that were requested. A possible cause for these peaks is a disk access from another process.

In Figure 3.28, the average frametimes for the camera path (using test system 2) through the Amphitheater model are shown, and again the camera path was started once with an empty cache (blue), and once with a pre-filled cache (red). Furthermore, the average times for loading the VBOs to graphics card memory for these two cases are shown by the dashed graphs. The maximum number of points loaded to graphics card memory was limited to 100,000 points. With this limitation, the time required to upload the points to graphics card memory never exceeds 1 millisecond per frame, so the impact on the frametime is very small.

The last test is concerned with the image quality after a fast movement of the viewpoint (or

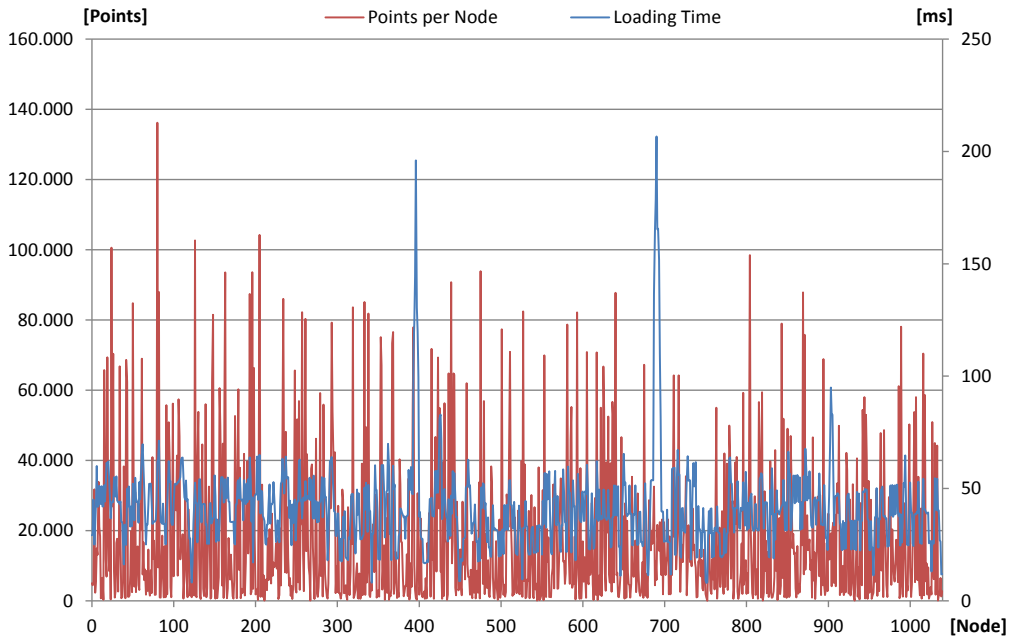


Figure 3.27: The red graph shows the number of points loaded from disk per node, and the blue graph shows the time that passed from requesting the points of a node until they are available in memory to be accessed from the main thread.

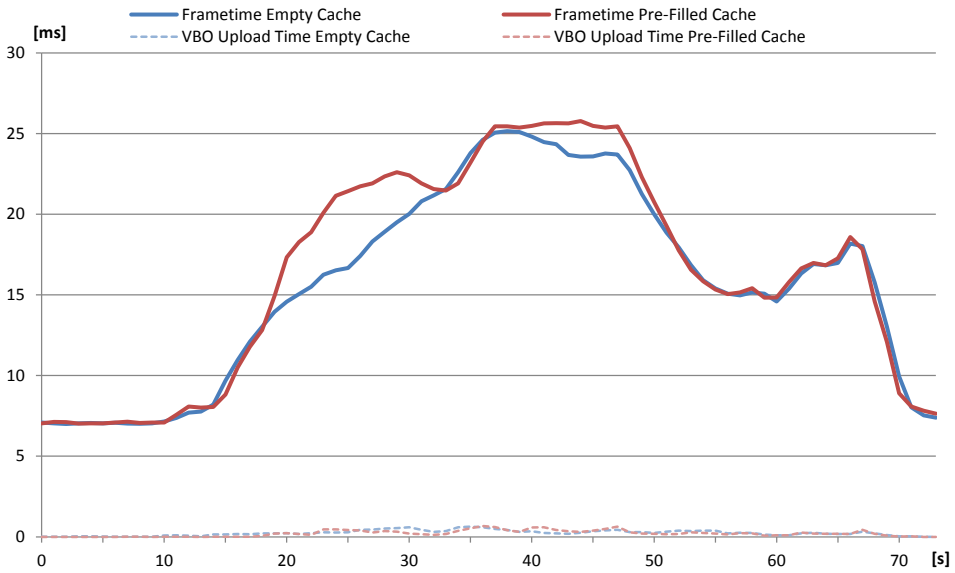


Figure 3.28: The solid-line graphs show a comparison of the average frametimes for the camera path through the Amphitheater model, once performed with an initially empty cache (blue), and once performed with a pre-filled cache (red). Furthermore, the average times for loading the VBOs to graphics card memory for these two cases are shown by the dashed graphs.

change of view direction) within a point cloud (for the test the Stephansdom model was used). In such a situation, a lot of points may have to be loaded to the graphics card, because (due to the size of available video memory or due to a densely sampled point cloud) the required points may not be available for rendering. Figure 3.29 shows a series of screenshots, where the points are successively loaded to the graphics card, after the view direction was rotated fast about 90 degrees to the right. Image (a) was captured immediately after the rotation was finished. Only about 1 million points were used for rendering, and all of them belong to nodes that were also used for rendering before the rotation occurred. The points in the left area of the image belong to nodes in lower hierarchy levels than the points in other areas, therefore the density of the points in this area is higher. Image (b) shows the result after 5 million points were loaded to the graphics card, and most of the visible surfaces are already closed. In Image (c), after 10 million points were loaded to the graphics card, all surfaces are closed, and the center area of the image becomes very detailed. In Images (d) to (f), more and more points are added to the border areas of the images, so details in those areas become gradually more pronounced.

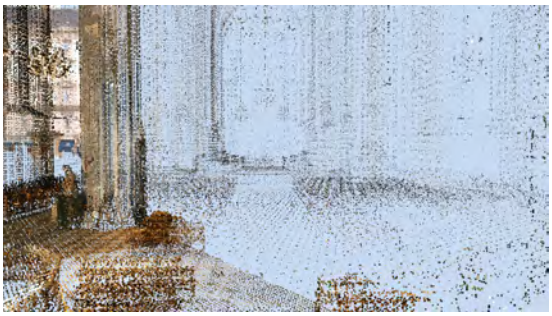
Note that the time required for loading the points to the graphics card is dependent on two factors, first on the current storage location of the points (i.e., on disk or in the LRU cache in main memory), and second on the number of points that can be loaded to graphics card memory per frame. Points stored in the LRU cache can be loaded much faster to graphics card memory than points stored on disk, but if too many points were loaded in the same frame, this would affect the frame rate. Therefore, the number of points loaded to graphics card memory in a single frame can be set as parameter, to ensure a continuous frame rate. Loading 24 million points from the LRU cache to the graphics card, without limiting the number of points loaded per frame, takes about 1 second. When limiting the number of points loaded per frame to 100,000, it takes about 4 seconds. When loading the points from disk (with or without limiting the number of points loaded per frame to 100,000), it takes about 11 seconds, indicating that disk access is the bottleneck in this situation.

3.11.4 Comparison of Build-Up Performance of MNO and MWO

In the following test series, the build-up performances of the MNO and the MWO are compared. The in-core build-up performance of the MWO was evaluated with the XGRT point processing framework, which was downloaded from their website [106]. As problems occurred when building point models out-of-core with the XGRT framework, the MWO data structure was then implemented in the point rendering system developed for the thesis. This made it possible to compare the out-of-core performance of the MNO and MWO data structures in the same point rendering system, eliminating potential side-effects of using different point rendering systems during benchmarking.

3.11.4.1 Comparison of In-Core Performance

For comparing the in-core performance of the MNO build-up algorithm with the MWO build-up algorithm, as presented by Wand et al. [107], test system 1 was used. Table 3.13 shows the parameters used for the MWO, the timings for the complete processing, and the timings for the in-core build-up alone. The results in this table for the MWO can be compared with the



(a) 1 million points.

(b) 5 million points.

(c) 10 million points.

(d) 15 million points.

(e) 20 million points.

(f) 25 million points.

Figure 3.29: Images (a) to (f) show the successive loading of points after performing a rotation about 90 degrees to the right. In Image (c), where 10 million points are rendered, the center area of the image is already quite detailed. In the subsequent images, the details in the border areas of the image become more pronounced.

Model	Attribs	Obj per leaf	Nodes mem	Time	Build Time	Size on disk
Santa Claus	p,c	131,072	512	11m 43s	1m 41s	778.6 MB
Santa Claus	p,c,n	65,536	3000	15m 21s	1m 51s	1,321.3 MB
Santa Claus	p,c,n	131,072	512	15m 07s	1m 44s	1,116.9 MB
Santa Claus	p,c,n	500,000	128	14m 47s	1m 28s	989.6 MB

Table 3.13: The build-up times and sizes for the resulting models for the Santa Claus point cloud (16M points) built with the XGRT system. Again, the "Time" column shows the complete processing time, including reading the data files. The "Build Time" column shows the build-up time alone. "Obj per leaf" means the maximum number of points per leaf node and "Nodes mem" means the maximum number of nodes in memory. The "Attribs" column shows which attributes were used for build up (p = position, c = color, n = normal).

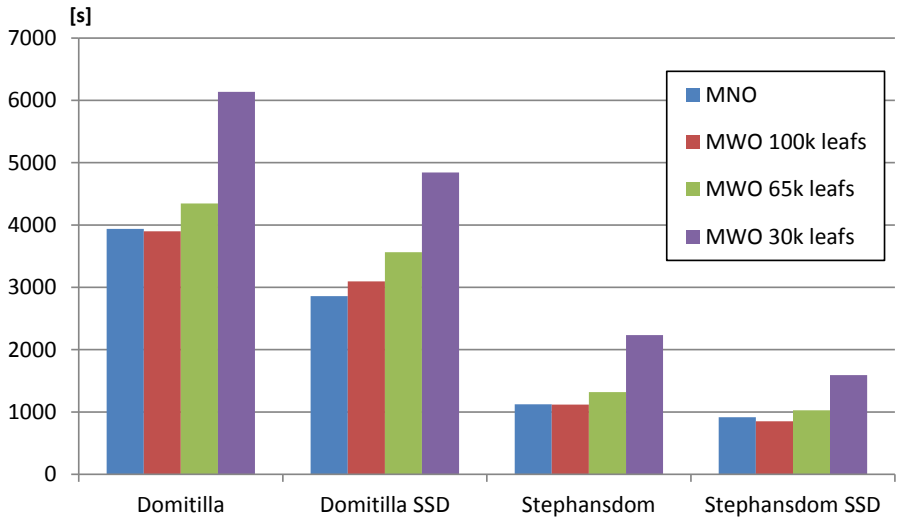


Figure 3.30: Buildup timings for the Domitilla and Stephansdom models on disk and on SSD.

results in Table 3.9 for the MNO. The build-up times of the MNO and the MWO are differing greatly, with the MNO build-up times being approximately 5 times faster. The sizes of the resulting MWO point models on disk seem to be dependent on the maximum number of objects that are allowed in the leaf nodes. The resulting models are 2 to 3 times larger than the size of the original point data and the MNO point models. These results are seemingly due to the implementation of the MWO in the XGRT framework, as the reimplemented MWO in the point rendering framework of this thesis shows a similar build-up performance as the MNO and much lower space requirements, as can be seen in the results for the out-of-core performance below.

3.11.4.2 Comparison of Out-of-Core Performance

For the comparison of the out-of-core performance, test system 2 was used. All operations on the data structures use an LRU cache of 100 million points.

Nodes swept back during build up				
Model	MNO	MWO 100K	MWO 65K	MWO 30K
Stephansdom	27 K	13 K	22 K	55 K
Domitilla	47 K	32 K	45 K	92 K

Table 3.14: The number of nodes swept back from disk to memory during build up. This number of nodes swept back to memory influences the build-up times of the point models.

Size of Point Models				
Model	MNO	MWO 100K	MWO 65K	MWO 30K
Stephansdom	12.9 GB	18.3 GB	20.9 GB	26.2 GB
Domitilla	30.8 GB	45.4 GB	51.6 GB	68 GB

Table 3.15: The total storage sizes for the different point models. The total storage size also influences the build-up times of the point models.

In this test series, 3 point models are used, the model of the Domitilla catacomb, the model of the Stephansdom with normals (using 28 bytes per point), and the model of the Hanghaus (see also Figure 3.20). The Hanghaus model was also used for the in-core performance benchmarking. Each point model was built-up in 4 different ways, once as an MNO with a minimum of 1000 points per node, and three times as an MWO with a different maximum number of points per leaf node (i.e., $30*10^3$, $65*10^3$, and $100*10^3$ points). The space requirements for an MWO increase with smaller leaf node size, e.g., for the Domitilla model the MWO with 30k leaf nodes is about 2.2 times larger than the original data set, which is shown in Table 3.15. This leads to higher build-up times, as can be seen in Figure 3.30. The build-up times for the Hanghaus model are always 30 seconds or less and therefore not shown in the figure.

The build-up performance of the MNO is either best or second best, and is comparable to the MWO with 100K leaf nodes. The other two MWO variants are always slower than the MNO. This is, as mentioned above, a result of the increased storage requirements for the MWO for decreasing leaf node size, but also dependent on the number of nodes that are swept back from disk to memory during build up. While the MNO requires always the least amount of storage space, it does not have to sweep back the least amount of nodes during build up. For the Domitilla model, it has to sweep back 47K nodes from disk to memory, as shown in Table 3.14. The MWO with 100K and the MWO with 65K leaf nodes have to sweep back less nodes (about 47% and 4% less than the MNO). Therefore, although they need more storage space, the build-up times are similar to the MNO. The MWO with 30K leaf nodes has to sweep back 92K nodes, and together with the largest space requirements, this results in a build-up time that is 50% to 100% longer than for the MNO. When using an SSD, the MNO can benefit from the decreased access times to the files when sweeping them back to memory, therefore it is the fastest when building the Domitilla model.

For the Stephansdom model, the MNO cannot take so much advantage from using an SSD, as the MWOs with 100K and 65K leaf nodes have to sweep back even less nodes, percentage wise, then for the Domitilla model (about 52% and 20% less than the MNO). Nevertheless, the

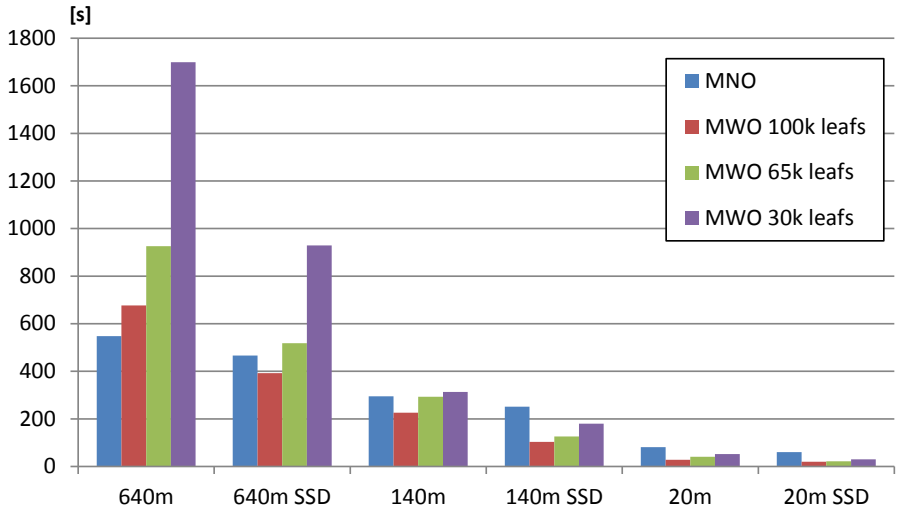


Figure 3.31: Timings for deleting differently sized selections from the Domitilla model.

build-up times for the MNO are still comparable to these two MWO variants, as the storage requirements are much smaller than for the MWOs. From all variants, again only the MWO with 30K leaf nodes performs much worse, due to the high number of nodes that are swept back to memory, and due to the high storage requirements.

3.11.5 Comparison of Editing Operations Performance

Editing operations on the MNO and MWO were tested by deleting different sized selections from the largest point cloud, the Domitilla model, with test system 2. The results are given in Figure 3.31, and show that the MNO is very efficient for deleting large areas from the point model. Interestingly, however, the MNO does not benefit from the usage of an SSD as much as the MWO variants. This means, the MNO uses mostly nodes available in the LRU cache. For smaller selections, the MNO eventually becomes the worst performing data structure, indicating that the search for the points to be deleted in the inner nodes takes some time. The MWO variants search for the points to be deleted in the leaf nodes, but then have to update the inner nodes' counters. Therefore, they have to access more nodes when deleting points. This makes them more sensitive to the access times of the external memory.

The optimization for the MWO, introduced in Section 3.8, where deleted points in inner nodes are replaced with still existing ones, causes some time overhead when deleting points. The overhead, though, is small. For the 640M points area, it causes a 1% time overhead, for the 140M area a 4% overhead, and for the 20M area a 10% overhead.

3.11.6 Comparison of Pixel Overdraw

The pixel overdraw of the 4 different data structures (one MNO, and three different versions of the MWO) was measured with different traversal strategies and three different point models. For these measurements, test system 2 was used. In Figure 3.32, the three point models, together

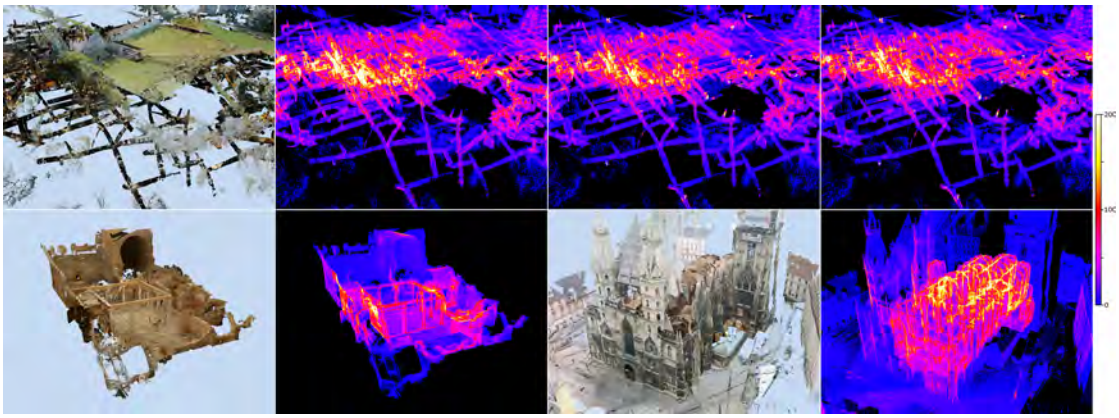


Figure 3.32: In the top row, the Domitilla model together with some overdraw heat-map visualizations is shown, from 2nd image left to right this are the MNO, the MWO with 30k leaf nodes and priority-based traversal, and the MWO with 30k leaf nodes and depth-first traversal. In the bottom row, the Hanghaus and Stephansdom models are shown, and the MNO is used for heat-map visualization. All traversals use a maximum projected grid cell size of 1 pixel, and the point clouds are rendered with 1 pixel per point.

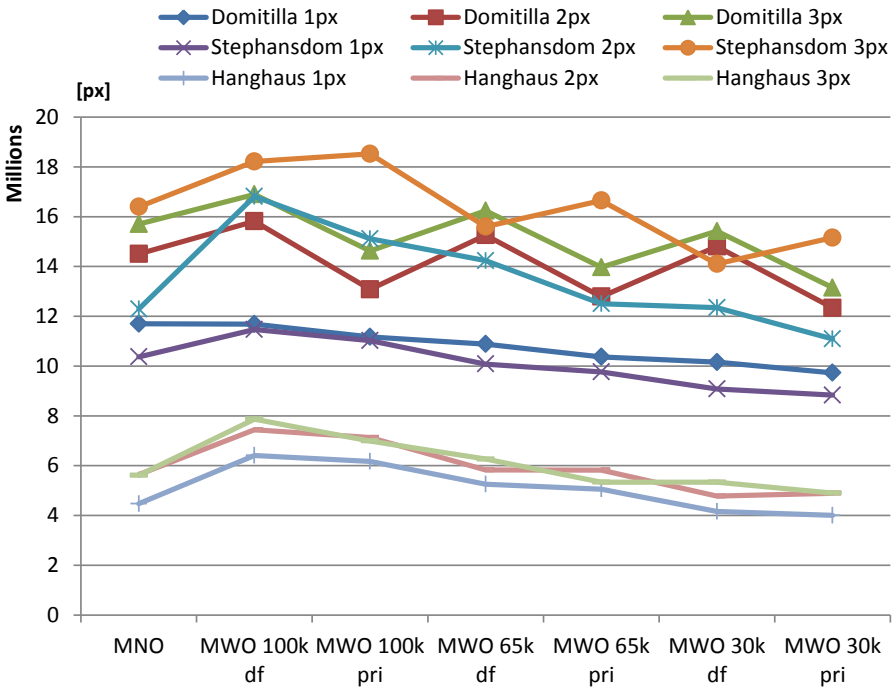


Figure 3.33: Total number of pixels drawn per frame at different settings for the projected grid cell size. The viewing position and direction is constant per point model. For the MWO variants, smaller leaf nodes cause less overdraw. Furthermore, the priority based traversal often requires less points to render than the depth-first traversal.

# Points	SelOct Build Time
313,010	0.08s
609,897	0.18s
2,521,201	0.79s
12,048,415	2.18s
13,248,293	2.52s

Table 3.16: Times needed for building a selection octree.

with some overdraw heat maps, are shown. Overall, the total number of fragments written to a 800x600 pixel viewport was measured in 21 different settings per point model. For each model, a maximum projected grid cell size of 1, 2, and 3 pixels was tested, and the models were then rendered in OpenGL with screen-aligned square splats of 1, 2, and 3 pixels side length. For each grid-cell size, the MNO model and the 3 MWO variants were rendered. Additionally, each MWO variant was rendered once with a priority-based traversal, and once with a depth first traversal. All renderings were done from one viewpoint per model. The results are given in Figure 3.33.

Although the MNO is never rendering the least number of pixels, it is only once rendering the largest number of pixels, together with the MWO with 100K leaf nodes (for the Domitilla model with 1 pixel projected grid cell size). The MNO is otherwise placed between the MWO variants. Generally, the MWO with 100K leaf nodes requires to render the most number of pixels, because to granularity of the rendered nodes is not matching the LOD levels as exactly as the MNO or the MWOs with smaller leaf nodes. The MWOs with smaller leaf nodes have the disadvantage, though, that they require much more storage space.

The traversal strategy also influences the number of rendered pixels, and in most cases the priority-based traversal requires less pixels to be rendered than the depth-first traversal. The only exceptions are the Stephansdom MWOs rendered with 3 pixels and the Hanghaus 30k MWO rendered with 2 pixels projected grid cell size, where the depth-first traversal causes less overdraw.

3.11.7 Build-Up Performance Selection Octree

The build-up times of the selection octree were also benchmarked (with test system 1), and the results can be seen in Table 3.16. Inserting points can be done at a rate of 5.26 million points per second. The deselection octree is built in the same time, the subtraction from the selection octree typically lasts 20 to 80 milliseconds.

3.12 Summary

The MNO data structure can be used to manage gigantic point-cloud data sets out-of-core. It provides an LOD structure which only uses the points from the point data set without creating additional points. For this, points are distributed to all hierarchy levels. Nevertheless, the complexity of operations performed on the MNO is comparable to the complexity of operations

performed on an octree that stores to points only at the leaf nodes. The points stored in the interior nodes of the MNO do not contribute to the overall complexity of the operations.

Sorting a point data set before inserting it into an MNO can decrease the time taken for building an MNO considerably, but the effect is dependent on the sorting order and on the memory available in the computer used for sorting the points and building the MNO. When having a lot of memory available where the points can be cached when building an MNO, compared to the size of the point data set that shall be inserted to the MNO, the sorting order is less important then when having only a small amount of memory available. Therefore, the preferred sorting method can be chosen depending on the available memory. In general, sorting a point data set before inserting it into an MNO makes the build-up performance more predictable, and often a lot faster.

The selection octree is a data structure that allows selecting points of out-of-core managed data sets. It describes the selection volume within which points are considered to be selected. Since the information about the selection is not stored at the points themselves, they do not have to be updated in external memory when they are selected or deselected. While the selection octree is generally easy to implement, there are special cases that have to be considered. The selection of single points requires to insert the points of all nodes of the MNO that are intersected by the selection tool, regardless if points inside the nodes are selected or not. This way, the resolution of the nodes in the selection octree is adequately adapted to the shape of the point model from which the points are selected.

A point size heuristic was also introduced that adapts the size of the rendered splats according to the point density inside a bounded area. In areas with a high point density, smaller splats are rendered. Additionally, to avoid rendering huge splats for points stored in the upper levels of the MNO hierarchy (in areas where points are also rendered from lower levels), the levels are assigned a weight which increases with the level depth. This way, the root node (at level 0) has the smallest weight, whereas nodes in the lowest level have the highest weight. When rendering points from different levels in the same area, then the splat size of the level with the highest weight will be preferred. Therefore, points in upper levels will be rendered with splat sizes from points in lower levels. This adjusts the rendered splat sizes of points stored in different hierarchy levels, and makes them more similar.

CHAPTER 4

Case Studies for the Use of Gigantic Point Clouds in Archaeology

In the previous chapter, the basic data structures for storing and interacting with gigantic point clouds were presented, and techniques for building, viewing, and editing such point clouds were introduced. In this chapter, the use of those techniques is demonstrated in the archaeological context, where increasingly large monuments are sampled with laser scanners for documentation purposes, and the resulting point clouds are used for further examinations. In the following, it will be shown that a number of tasks can be performed on such point clouds. For this, the point clouds of two monuments will be used.

One of the two point clouds was recorded at very large and complex Domitilla Catacomb in Rome. The left image in Figure 4.1 shows a bird's eye view of the point model. The Domitilla Catacomb is the largest of Rome's over 60 known catacombs, built by the early Christian communities between the late 2nd and the early 5th century. The soil around Rome consists mostly of tuff, which is consolidated vulcanic ash. Tuff allows for easy digging, and therefore catacombs were extended frequently and without plan. The result is a very irregular network of galleries, with a length of 12 kilometers in the case of the Domitilla Catacomb, which is challenging to document.

The other point cloud was recorded at the excavation site of the Amphitheater 1 in Bad Deutsch-Altenburg, Austria, which was part of the Roman city of Carnuntum. In the middle of the 1st century A.D., the Romans built an army camp in Carnuntum, and later on also a civilian city developed there. The right image in Figure 4.1 shows a bird's eye view of the point model. The amphitheater has a size of 97 x 76.6 meters.

Only recently, the design and deployment of moveable laser range scanning equipment made it possible to comprehensively measure the extension of these monuments in all 3 spatial dimensions. However, the sheer amount of data that is produced by scanning and photographing these monuments can be huge (about 30GB 3D data for the Domitilla catacomb, and about 1.6GB 3D data for the amphitheater model), sometimes surpassing the available memory of off-the-shelf computer hardware. Therefore, archaeologists currently only work with very small subsets of the

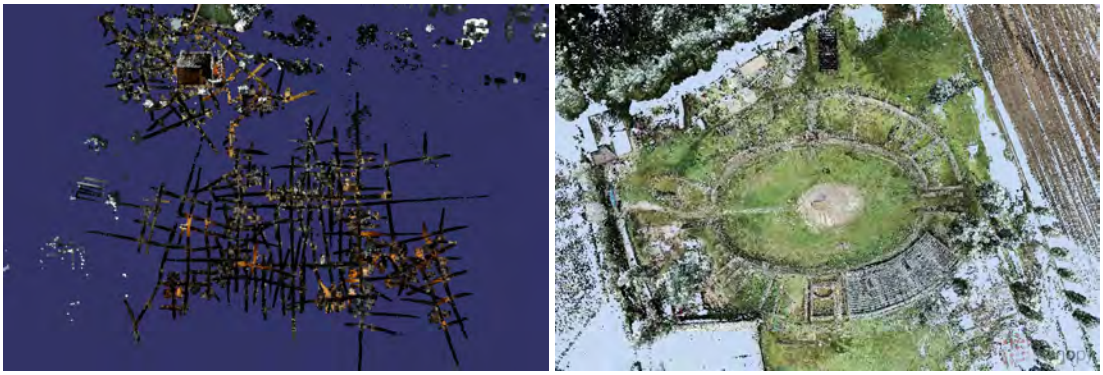


Figure 4.1: In the left image, the model of the Domitilla Catacomb is depicted, and in the right image the model of the Amphitheater 1 in Carnuntum is depicted, each from bird's eye view. The models consists of about $1.92*10^9$ and $106*10^6$ points respectively.

data, which are usually converted into 3D meshes. Mesh conversion is tedious and error-prone, especially for models with a non-manifold structure, or for areas that suffer from undersampling, both of which appear inevitably when scanning large monuments. Furthermore, a mesh is already an interpretation of the original data, which is often not desired by archaeologists.

For these reasons, mesh conversion may be suitable for small artifacts like statues, or for producing high-quality visualizations of small parts of a monument. But especially in a large monument like a catacomb, it is important to be able to interact with the whole monument at any one time. Therefore, the only resort is to work with a severely subsampled version of the original data set, for example using an in-core point renderer, which is usually part of the software that comes with the laser scanner. However, subsampling makes it impossible to inspect details, and simple point renderers use constant splat sizes, causing surfaces to "dissolve" when approached. This makes it difficult to get a correct 3D impression of the model.

While high-quality accurate reconstruction using meshes might be necessary for some tasks, one important point made in this chapter is to show that there are many tasks that can be carried out perfectly well using the raw point cloud, making direct point visualization and interaction an extremely relevant topic for archaeological work. These tasks include exploring the large-scale structure of the model, classification of findings, taking measurements, virtual anastylosis (i.e., creating hypotheses about the original structure of the monument), creating sliced views, providing visualizations limited to different building phases, and acquiring an understanding of the 3D structure of the monument in general. The point rendering system presented in this thesis (see also Chapter 7) makes all these tasks possible due to the interactivity afforded by the out-of-core MNO data structure and rendering algorithm (see also Chapter 3). Furthermore, a point rendering technique is discussed that reduces some of the artifacts that occur when rendering point models built from point clouds of several single scans, especially artifacts introduced by coloring the point clouds with photos taken during scanning.

Figure 4.2: In the left image, the viewpoint is hovering between the 1st and 2nd floor of the catacomb, showing the detailed scanning of the galleries. The right image shows a cubiculum from the inside. The registration markers are clearly visible.

4.1 Domitilla Catacomb

4.1.1 History

The Domitilla Catacomb is the largest of the more than 60 catacombs in Rome. It was built not only as a large subterranean graveyard, but also as a place of pilgrimage. It holds artifacts of the early Christian communities, like wall paintings and inscriptions, but also a subterranean basilica. The catacomb grew within a period of some 300 years, beginning in the late 2^{nd} century, from isolated tombs to a network of 12 kilometers of galleries on 4 different levels. Some galleries have a height of 5 meters, which seems to be the result of several building phases. There are also separated tombs, called cubicula, which can be accessed from the galleries. About 80 of these tombs feature artistic wall paintings. Until the 8^{th} century, the catacomb was used as a place of pilgrimage to the graves of the two saints Nereus and Achilleus. When their relics were moved to another place, the catacomb fell into oblivion until the 17^{th} century. At that time, it was rediscovered and has since been studied by archaeologists. Even after 400 years of research, a complete documentation of the findings or the structure of the catacomb is not available [114].

4.1.2 Model of the Domitilla Catacomb

In 2006, the Austrian Science Fund project "The Domitilla-Catacomb in Rome. Archaeology, Architecture and Art History of a Late Roman Cemetery" started to document the current state of the catacomb, including galleries, paintings, and finds. The 3D structure of the catacomb was recorded with a Riegl LMS-Z420i laser scanner [90], capable of scanning 12,000 points/s in a distance between 1 to 800 meters and with a depth accuracy of ± 10 millimeters (standard deviation). Each scan is colored by photographs, which were taken while scanning with a Nikon D100 digital camera mounted on top of the scanner. For lighting the photographs, a flash was used. The individual scans were registered in a global coordinate system using markers. From reduced versions of the point clouds, archaeologists also created 2D plans using standard software [34] [114]. In some particularly interesting areas, paintings were also photographed

Figure 4.3: This cubiculum in the region “Flavi Aureli” cannot be reached at-grade. The red lines denote the different height levels of the floor (a difference of more than 2 meters) and were later added to the image for visualization.

separately with a Canon EOS-1Ds. Photogrammetric methods [3] were used to get fully textured mesh models [72]. To bring the laser scan data and the mesh models into the same coordinate system, a Leica total station TCRM 1103i was used. The total station helped to measure a fix point net all over the catacomb.

Scanning a confined space like a catacomb proved difficult, because each scan position only captured a small part of the structure. Furthermore, some rooms that were abandoned could only be entered using ladders (see Figure 4.3). The catacomb was scanned in the course of 11 scanning campaigns. All areas of the catacomb, including the gallery system and the basilica, were scanned, resulting in a compound point cloud of about $1.92*10^9$ points from 1,826 individual scan positions. When only position (3 single precision floats) and color (4 bytes for RGBA) information is stored for each point, this results in a massive point model of 30.8GB data. Figure 4.2 shows two views of the model.

4.2 Amphitheater 1 in Carnuntum

The Roman Amphitheater 1 in Bad Deutsch Altenburg, Austria, is part of the excavation site of Carnuntum, a Roman city and army camp that was part of the limes pannonicus, the border between the Roman empire and the Germanic (and later also Sarmatic) tribes, who settled north and east of it. Carnuntum was at the crossways of the limes road (a road in east-west direction along the limes) and the (eastern) amber road (a trade route in north-south direction from the Baltic sea to the Adriatic sea) and was inhabited in its heydays (around the year 200 A.D.) by about 50,000 people. The Amphitheater 1 is one of the most attracting monuments of the archaeological park Carnuntum (the part of the excavation site that is publicly accessible).

Figure 4.4: The left image shows the center of the Amphitheater 1 from an entrance along the longer axis of the elliptically shaped footprint of the building. The right image shows the modern-times grandstand for spectators who attend one of the different events performed in the amphitheater (for example re-played gladiator fights).

Therefore, the department for hydrology and geo-information of the provincial government of Lower Austria carried out several 3D laserscanning data acquisitions to document this ancient site. Within a period of four years, between 2007 and 2010, about 200 scan positions were placed to acquire the whole historical building structure as well as to document the ongoing excavation process. For acquiring the data, a Riegl LMS-Z420i laser scanner with a Nikon D70 digital camera mounted on top of the scanner was used. The whole data was georeferenced in the national coordinate system by using tie points for a coarse registration and identical patches in the recorded data were used for a multi-station adjustment. An overall accuracy of 1 - 2 cm for each scan position was accomplished. Figure 4.4 shows two views of the model.

4.3 Exploration

With the point rendering system described in this thesis (see also Chapter 7), the user can explore huge point-based models like the Domitilla Catacomb interactively. One main point stressed in this thesis is that by using a raw point model and the presented point rendering system, a number of different archaeological tasks, as detailed in the following, can be completed without having to resort to mesh generation or other costly preprocessing operations.

4.3.1 Understanding the Complexity

Maybe the most immediate benefit of always having access to the whole model is that it allows users to quickly gain a thorough understanding of the object. When moving around the model fluidly, the parallax effect between close and distant areas gives important hints on the 3D structure of the object. In the Domitilla Catacomb, for example, the interconnections of the subterranean galleries changed over time. Some entrances were created while others were abandoned or became inaccessible. This development can be understood better by being able to roam the model freely and fluidly, allowing quick changes of viewpoint.

A subsampled model cannot give the same information: when the user wants to investigate an area more closely, the surface structure disintegrates, and important 3D information gets lost.

4.3.2 Area-of-Interest Extraction

Sometimes it is necessary to extract an area from the whole point cloud, for example in order to examine an object without occlusion by surrounding geometry (like galleries). This can be done by marking the area of interest with the brush, inverting the selection, and then making the selection invisible (see Section 3.4). In practical use, archaeologists also found it convenient to directly mark the surrounding geometry using the brush. The area of interest can also be exported for further processing (e.g., for creating a detailed mesh for a show case).

Using the scanner software would be much less convenient: there, the user has to know which scans influence the area of interest, merge these scans manually and extract the area of interest in the merge result. If there are more than a couple of scans involved in the area of interest, merging is not feasible anymore because it takes too long, and the extraction has to be done separately in each of the involved scans.

4.3.3 Videos for Presentations

The system also allows generating videos for presentation. This is important, because especially in large monuments like the Domitilla Catacomb, a presentation of the whole model is desired, and not only of the small parts that have been meshed in high quality. The system can generate high-quality videos by disabling the rendering budget such that rendering proceeds to the optimal LOD for the whole scene, and surfaces can be estimated for close-up views. Using the scanner software, the point cloud would have to be carefully downsampled in order to allow video generation, and close-up views would have to be avoided.

4.3.4 Cuts, Slices, Floorplans

Many archaeological tasks involve slices through the model. The box tool (see Section 3.4) allows quickly generating arbitrarily oriented slices through parts of the model. The area selected by the box can be visualized in an orthographic view to generate for example floor plans (horizontal cuts at wall height), or orthophotos of walls (for aligned vertical cuts). Again, it is very convenient to have the whole model available for these tasks and not to have to sort through individual scans.

4.3.5 Virtual Anastylosis

Archaeologists often make hypotheses about the original structure of a monument, which can be made more tangible by actually reconstructing a part of a monument using available parts. However, actual reconstruction is expensive and cannot be undone in case of errors, or is sometimes impossible because the involved parts are simply too large. Therefore, it is convenient to be able to move parts virtually. For example, columns of a temple that have been found on the ground can be selected (using the brush) and then moved around freely (by deleting them from

Figure 4.5: A close up view of a gallery in the region “Flavi Aureli”. Some areas are marked for deletion to gain a model of the initial building phase.

the point cloud and reinserting them at a new position). In the Domitilla catacomb, there are questions about the original height of the complete basilica that could be tested using this tool.

4.3.6 Measuring

Distances are easy to measure by selecting two points in the model. In the same way, it would be easy to include area and volume measurements for selections.

4.3.7 Reconstruction and Visualization of Building Phases

An archaeological monument is typically constructed in several distinct building phases, often spread over decades or centuries. Reconstructing the monument at a particular building phase gives valuable insights into the building history of the monument. This can be accomplished with the system by deleting the areas belonging to other building phases. An example can be seen in Figure 4.5. Here, the building phases of the vertical high galleries are reconstructed. Initially, a gallery of the height of one man was dug out, and the cubiculum in the image was accessible at ground level. Over time, the floor of the gallery was dug deeper, and the cubiculum in the image became inaccessible, at least without auxiliary means (see also Figure 4.3).

At a larger scale, it becomes even more important to have the whole model available for this task, for example when reconstructing different phases of growth for the whole catacomb, which started from different independent centers and grew together over time.

4.3.8 Impossible Views

An interactive exploration system allows views onto parts of the model that are not possible in reality, like from inside solid ground. For example, the basilica was built in honor of two saints,

Figure 4.6: The tombs of the two saints buried below the basilica. The green border was added to the image to emphasize the tombs.

Nereus and Achilleus, and they were probably buried in the tombs in the center of the basilica, surrounded by the apsis. When looking from below the floor of the basilica towards the tombs, they are clearly visible, as can be seen in Figure 4.6. Views from below, or any other position normally not available, can provide a valuable tool for research. The shape of the graves and their exact position in relation to the grid of galleries could give hints on their origin.

4.3.9 Combining Viewing Regions

Only a virtual exploration tool allows combining model parts that are normally only visible from distinct viewing regions into one view. For example, the inside of the catacomb can be visualized together with structures that can only be seen from above ground. This allows showing which parts of the catacomb lie below which landmarks of the actual city today.

Another example is the basilica of the Domitilla Catacomb, where only the upper section of the building is visible from ground, the rest of the building is below ground level. The left image of Figure 4.7 shows the basilica cut off at ground level. The right image gives an impression of the subterranean structure. The garden surrounding the main entrance of the catacomb is just above several cubicula of the Northern area of the catacomb.

4.3.10 Scan Campaign Support

Already during a scan campaign, it is very useful to have quick access to the whole model. This way, areas that are not sampled densely enough or that are not captured at all can be identified and new scan positions can be planned.

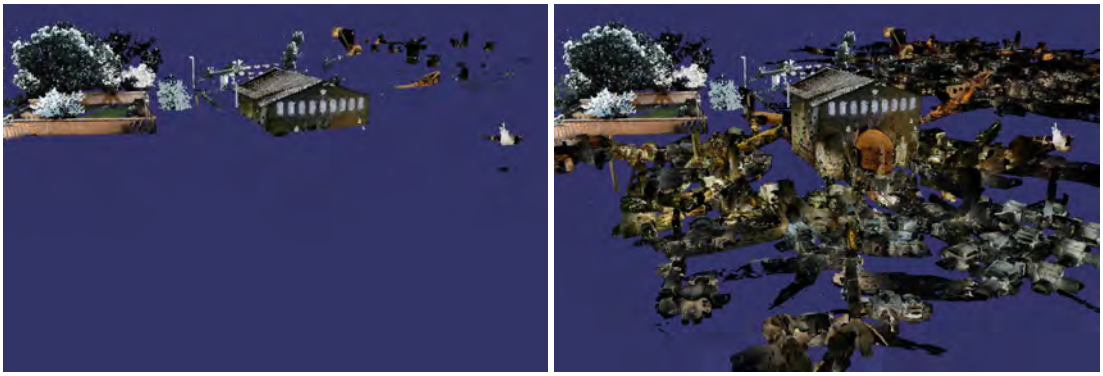


Figure 4.7: The basilica is partly above and partly below ground level. In the left image, the point cloud is cut at the ground level, so that only objects above ground remain visible. The right image shows the whole catacomb, including the subterranean areas, and the enormous expansion of the catacomb underground becomes visible.

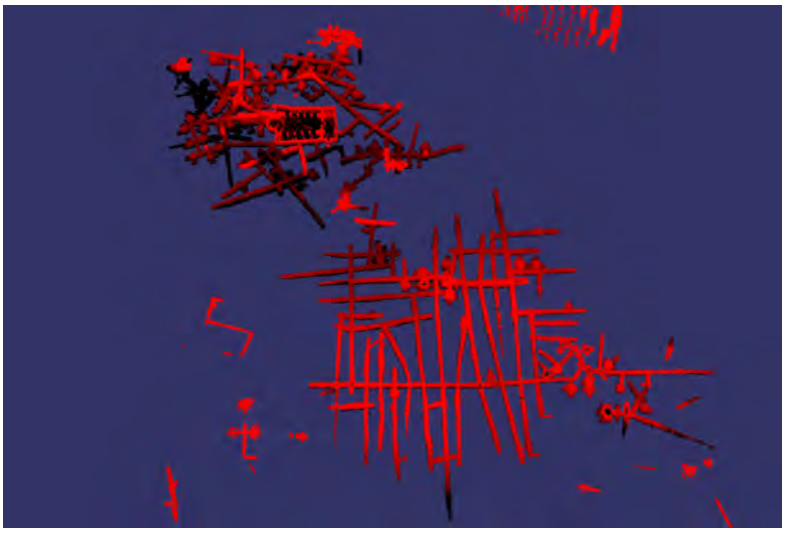


Figure 4.8: The second floor of the catacomb with elevation encoding. The height difference between pure red and black encoding is 3 meters.

4.3.11 Elevation Encoding

The model can be rendered with elevation encoding, so the user can estimate the run of the height of a gallery or an entire floor. The top and bottom height for the encoding can be chosen by the user. Figure 4.8 shows the second floor of the Domitilla Catacomb. The change of shading of the galleries in the lower part of the image means that the galleries are not at the same elevation over their entire extent. The height difference between pure red and black encoding in this image is 3 meters.

4.4 Splat Rendering

The visualization of huge point clouds coming from laser scans is a challenging problem due to several issues. First, the amount of data produced by laser scanners has to be handled, as it can easily exceed the size of the main memory available in off-the-shelf computers. This problem is even more relevant for point clouds composed from different scanning positions. Second, the local varying point density of those huge point clouds has to be accounted for. And third, after coloring the point clouds (from the different scans) with the photos taken during scanning, color artifacts might occur in areas where points are colored from different photos (due to different lighting conditions when capturing the photos).

While in this thesis methods were already proposed for the first and second of the before mentioned issues, the third one was ignored so far. In the following, rendering point clouds with simple OpenGL splats will be compared to rendering point clouds with Gaussian splats, a method which alleviates the color artifacts issue, and which also allows showing details of the scanned objects when using splats larger than 1 pixel per point. Note that for rendering, point clouds are used, which were not processed except for colorization. Therefore, the point clouds will exhibit noise in the positions of the point samples. Furthermore, no assumptions are made about the uniformity of the sampling, i.e., the point clouds can have widely varying local sampling densities. Nevertheless, when using Gaussian splats for rendering, the contributions of neighboring points will be blended at the pixels, which helps to alleviate the effects of those issues.

4.4.1 Rendering with Screen-Aligned Square Splats

The points are sorted into an MNO and rendered according to the out-of-core algorithm described in Chapter 3. After choosing the points and calculating the point sizes with the point-size heuristics (these two steps are done in every rendered frame), the points are then projected to screen space and rendered as simple screen-aligned OpenGL splats, which are drawn as square rectangles. When using color for the points, this rendering method will lead to color noise (a term which refers to the abrupt changes in color for neighboring splats). The reason for the color noise is that the splats which are used for rendering might be overlapping in screen space. Rendering with square splats and one color per splat results in images like the upper one in Figure 4.9. As can be seen, the rendered point model exhibits a lot of color noise. The point-size heuristics fills most holes in the point model, only at the staircase in the center of the image, holes in the model become visible. Improving the quality of the rendered point cloud would mean to increase the sampling of the geometry. Since the point samples are given, the sampling density cannot be increased. Therefore, a rendering method based on signal reconstruction theory was implemented, which is described below.

4.4.2 Rendering with Gauss Splats

Instead of rendering the pixels of a splats only with the color of the according point sample, the rendering technique termed Gaussian splats blends the contributions of neighboring splats overlapping in screen-space. This way, a reconstruction of the underlying signal, which consists

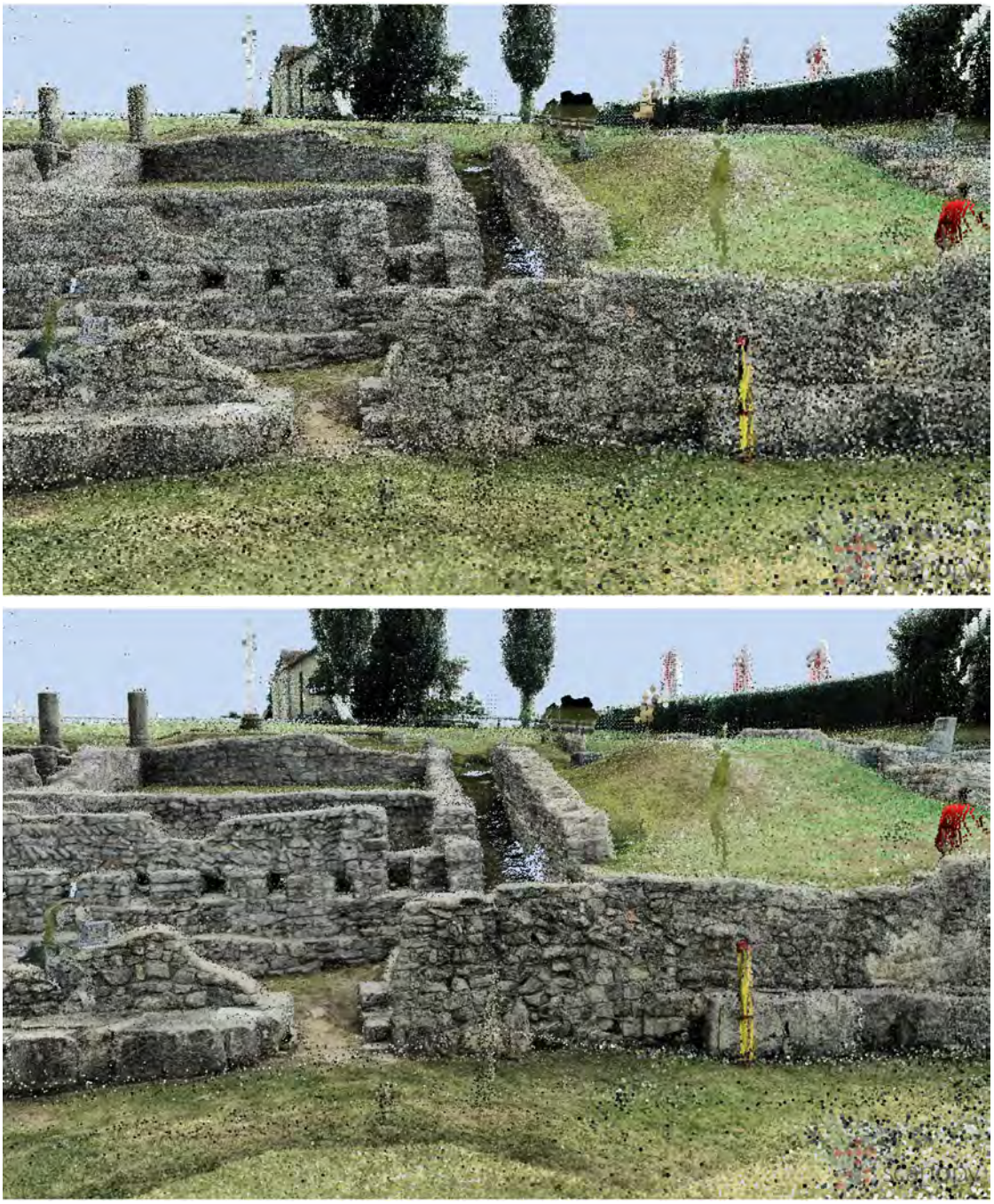


Figure 4.9: In the upper image, the points of the Amphitheater 1 model are rendered as square splats, and each splat is rendered with one color. The single stones of the wall are not clearly visible due to color noise and overlapping splats. In the bottom image, the points are rendered as Gaussian splats. The single stones of the wall are clearly visible due to the reduced color noise and blending of overlapping splats.

of the colored point samples, can be performed at the pixels in screen space. The rendering technique is based on the EWA resampling framework for point splats introduced by Zwicker et al. [116] (see also Section 2.5). With this framework, it is possible to apply textures to irregularly sampled point-based models in highest quality. It was slightly modified to make it usable by today's programmable graphics cards by Botsch et al. [12]. In this thesis, the framework of Botsch et al. is used in a simplified variant, and noisy point clouds with widely varying sampling densities are rendered with it.

The point clouds coming directly from laser scans have no normals per point available, therefore shading and orientation of the splats have to be neglected. Also the blending depth, as described later, is set to a constant user-defined value throughout the point cloud for all rendered points. In uniformly sampled point clouds, the blending depth is usually set to the average distance of points or to the radius of a single splat, but in noisy point clouds, both measures are unreliable estimates, therefore the blending depth can be set by the user.

The Gaussian splats are drawn as screen-aligned circular disks, which is achieved by shaping screen-aligned square splats (whose side length is equal to the diameter of the circular disks) in a fragment shader according to radial basis functions centered at the projected point positions in screen space. Since unprocessed point clouds are used, no exact splat radius is calculated for the points (so that neighboring splats would just slightly overlap while still creating a closed surface), but instead the point size heuristics is used.

Rendering the splats is done in three passes, which are executed in every rendered frame, similar to the rendering described by Botsch et al. [12]. First, all points that are available in the view frustum are projected to screen space, and the so-created point splats are used to fill the depth buffer of the viewport. The depth buffer is then used as input to the next step as depth image. Note that in the first step, no color information is used. In the second step, all points are projected to screen space again, but this time the previously calculated depth image is used to discard points that are invisible. The pixels of the splats produced by these hidden points would be discarded anyway, so only the color information from really visible points is blended. The blending of the splats is done according to a Gaussian weighting function, with the center of the weighting function at the projected point position. Therefore, the pixels which are close to a projected point position get the largest weight for the point's color. When blending two splats, the weights for the colors from the two splats are used to determine the contributions of the colors for the resulting pixel. In the third step, the accumulated color information for each pixel is normalized, since usually the color contributions do not form a partition of unity for each pixel, which would lead to artifacts in the blended colors.

4.5 Results

4.5.1 Exploration

The point rendering system was tested with test system 1 (see Section 3.11) and the point model of the Domitilla catacomb. For the visualization of the model, a rendering budget of 12 million points was used (taking 192MB of GPU memory, which works for all modern GPUs, including laptops), which resulted in a minimum frame rate of 20 frames per second while navigating

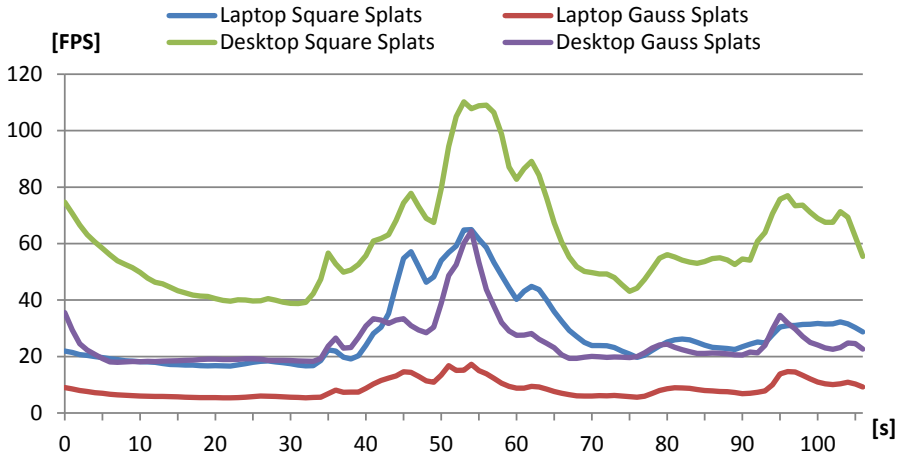


Figure 4.10: The graphs show the frames per second for camera paths through the Domitilla model, performed on two different test systems. The points are once rendered as square splats and once as Gaussian splats.

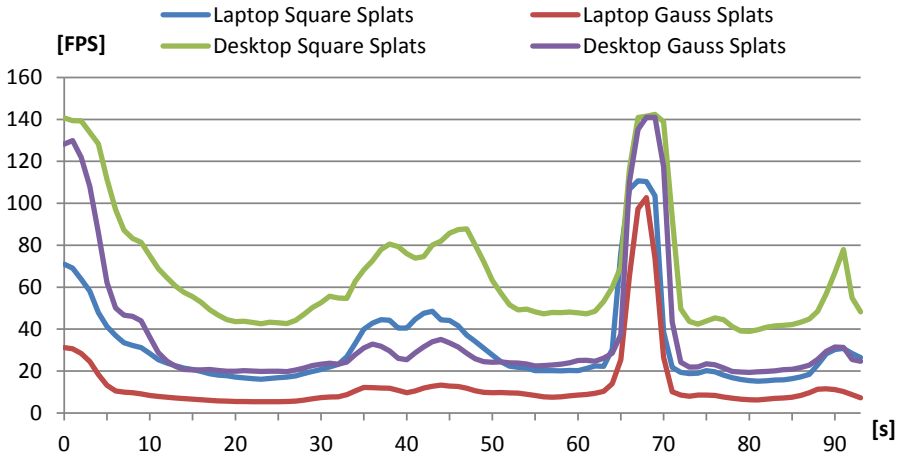


Figure 4.11: The graphs show the frames per second for camera paths through the Stephansdom model, performed on two different test systems. The points are once rendered as square splats and once as Gaussian splats.

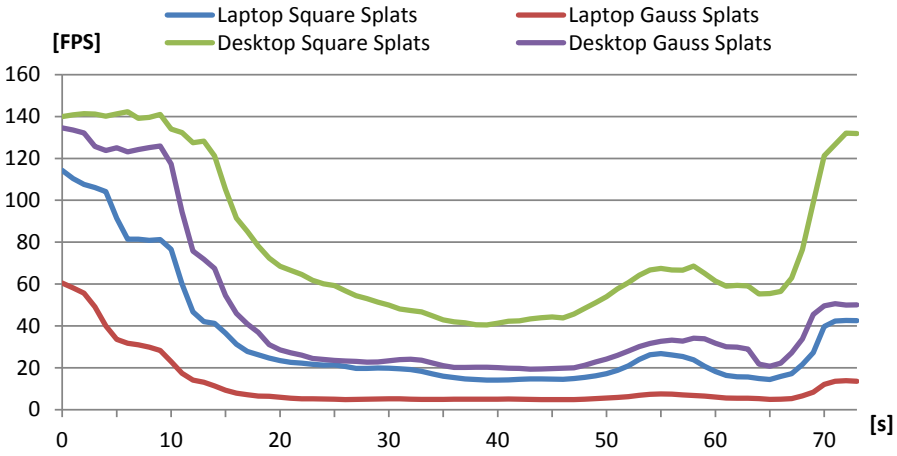


Figure 4.12: The graphs show the frames per second for camera paths through the Amphitheater 1 model, performed on two different test systems. The points are once rendered as square splats and once as Gaussian splats.

the model. In most areas, however, the whole rendering budget was not needed to achieve the optimum LOD, so the frame rate was often higher.

The actual evaluation how well the system works can only be done with end users. A formal study is difficult, though, since only the small team of archaeologists and architects involved in the documentation of this particular monument can interact with the system in a sensible way. The informal discussions with this team, consisting of one archaeologist and two architects, resulted in the feedback that the interactivity of the system and access to the whole point data set was a huge help in their work. For example, they were all able to extract a region of the model and reconstruct different building phases by removing parts from later phases, which is maybe one of the most demanding tasks described. Apart from experimenting with capabilities of the system, they also came up with several ideas for features that would support further archaeological tasks. These ideas include for example an interface for classification and categorization, and can be used to developing more tools for the interactive exploration of point clouds.

4.5.2 Gaussian Splats

An example for a point cloud rendered with Gaussian splats can be seen in the lower image of Figure 4.9. The details of the stone wall are much more visible compared to the simple rendering with square splats in the upper image of Figure 4.9, and the color noise is greatly reduced. Some noise in the positions as well as the colors of the points is still visible, which has two sources. One source are outliers, which could be removed by manually cleaning the point cloud. The second source are points, which are colored by photos that were taken under very different lighting conditions compared to the photos used for coloring their neighboring points, so that blending the splats of these points cannot compensate for the differences in the colors.

(a) 25M points, desktop 22.7 FPS, laptop 4.2 FPS

(b) 25M points, desktop 22 FPS, laptop 4.1 FPS

(c) 12.5M points, desktop 40.8 FPS, laptop 7.2 FPS

(d) 12.5M points, desktop 38.6 FPS, laptop 7 FPS

(e) 6.25M points, desktop 70 FPS, laptop 11.8 FPS

(f) 6.25M points, desktop 65.8 FPS, laptop 12.2 FPS

Figure 4.13: Two screenshot series (one on the left and one on the right), which show the effects of different point budgets on the quality of rendered images when using Gaussian splats for rendering. The screenshots in the top row have the highest point budgets and show the most details. The screenshots in the middle row still show details in the center areas, but the border areas are already somewhat blurred. In the screenshots in the bottom row, the border areas do not have enough points for rendering closed surfaces anymore. For all test cases, the frames per second were measured once for the desktop and once for the laptop test system.

4.5.2.1 Rendering Performance

Figures 4.10 to 4.12 show the rendering performance of Gaussian splats compared to screen-aligned square splats, by measuring the frame rates when moving the viewpoint along camera paths through three different point models. The test setup is the same as for the tests in Section 3.11.2, i.e., the test systems, the point models, and the test parameters are the same. It can be seen that rendering with Gaussian splats causes the frame rate to drop about 50% or even more, compared to rendering screen-aligned square splats, especially when rendering a lot of points. This frame rate drop is caused by the additional rendering passes required for Gaussian splats. For test system 3 (i.e, laptop), the number of points that have to be rendered when using Gaussian splats is almost too high, as often only about 5 FPS are achieved. In such a case, the maximum number of rendered points could be lowered. When reducing it by 50% (to 12.5 million points), the minimum frame rate increases to about 7 FPS.

In Figure 4.13, two screenshot series of the Stephansdom point model are shown that demonstrate the effects of using different point budgets when rendering the points as Gaussian splats. One series shows a view down the nave of the Stephansdom towards the high altar (at the left side of the figure), while the other series shows a close-up view of a small altar in front of a pillar between the nave and the right aisle (at the right side of the figure). For both series, also the FPS for the desktop and the laptop test systems were recorded, and they are noted in the descriptions of the screenshots. When 25 million points are used for rendering (top row of screenshots), all details in the images are visible. When using half of the points (middle row of screenshots), the center areas still remain detailed, but the border areas become somewhat blurry. By reducing the point budget 50% further to 6.25 million points (bottom row of screenshots), more artifacts occur in the border areas of the images, and some areas are not receiving enough points for drawing closed surfaces. This is due to the high density of the points in the point model, so areas in the center of the image, with a high priority for the rendered nodes, already use up most of the point budget available during rendering. For point models with a lower density, a point budget of 6 million points might still be reasonable.

4.6 Summary

The exploration of huge point models can support the work of archaeologists by providing insights which are not possible when only looking at separate parts of the data set. Only a view of the complete data set can reveal links between different areas, which are otherwise not visible. Such a view can be provided by an out-of-core managed point cloud. Additional tools can be used to extract special regions of interest, measure distances, or create cuts through a point cloud.

Rendering points with methods derived from signal reconstruction theory can reveal details which would otherwise not be visible, even in very noisy point data sets. Therefore, the presented methods are a valuable addition to the set of tools used in modern archaeology, but these tools can also be used for exploring huge point data sets in general.

CHAPTER 5

Case Study for the Use of Virtual Texturing in Archaeology

While in the previous chapter the interaction with huge point clouds was described, in this chapter the combined visualization of huge point clouds and virtually textured mesh models is examined. This combined visualization can improve the user experience when exploring huge data sets. While point clouds can be created fast with modern laser scanners, the visual quality of the resulting point models depends on the pre-processing steps performed on the data before visualization. These pre-processing steps can involve cleaning, denoising, resampling, or coloring the data, maybe even some modeling, if holes have to be filled in areas that were not sampled by the laser scanner. Since each of these pre-processing steps takes time, it is sometimes not feasible to do them due to the size or the complexity of the data set. In such cases, it can make sense to pre-process only the most relevant areas of the data set, while keeping the other areas untouched. Even a combination of models based on different data structures or rendering primitives can be an improvement to the perceived quality.

In the following, a system for combined out-of-core rendering of huge point clouds and virtually textured mesh models will be introduced. Furthermore, a workflow on how to get high-resolution textured mesh models from free-hand taken photographs of (architectural) cultural heritage sites is shown. This workflow was not developed in the course of this thesis, but it is included in the description below to get an overview of how virtually textured models are created.

5.1 Virtual Texturing

The survey of huge cultural heritage objects was in the past a very exhausting work and in many cases not feasible. From the more than 60 late Roman/early Christian catacombs in Rome, the bigger ones were never surveyed in their entirety due to lack of adequate surveying techniques. Archaeologically more interesting areas of these catacombs, like family hypogea or painted

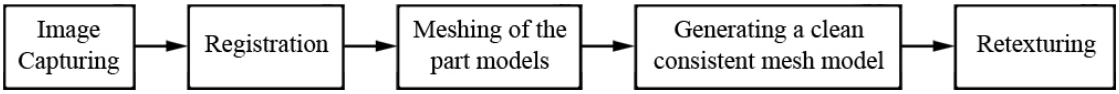


Figure 5.1: Workflow for the generation of textured mesh models.

tombs, have been surveyed already very early (beginning with the end of the 16th century) with different methods, but an overall survey was never made. With the emergence of 3D laser scanners, the geometrical survey of such objects became possible. In Section 4.1, the surveying of the Domitilla catacomb with a laser scanner is described, which resulted in a huge data set of about $1.92*10^9$ points. Additionally, more than 23,000 photos were captured, which could then be used for the generation of 3D textured mesh models.

5.1.1 Creating Mesh Models from Photographs

The condition of the late Roman / early Christian paintings in the Domitilla catacomb is in general very good. A survey by the 3D laser scanner used for sampling the geometry of the catacomb cannot adequately record these paintings, due to the limited resolution of the scanner. With a photogrammetric survey of the paintings, it is possible to generate accurate three-dimensional models with high-quality textures. The experience showed that the quality of textured mesh models emerging from laser scan data is not good enough. Especially the precision when connecting the geometry and texture in these models does not live up to the desired standards. By using photogrammetry, the geometry develops out of the photos that are also used for texturing, and discrepancies can be minimized.

A cubiculum of 2 x 2 meters with three arcosolia (a standard situation in the catacomb) and covered with paintings requires at least 18 camera positions to be captured completely. If the entrance gallery and the entrance wall should be modeled as well, at least 12 more positions are necessary. By using a so-called panoramic head, which allows rotating the camera around the focal point, photos of the entire cubiculum can be made from the same position. After capturing all images of one painting (for large paintings, 700 or more images are necessary) they are loaded into the software CalibCam [3]. Here the registration of the images can be done semi-automatically and the exterior calibration (measured tiepoints from a total station are also implemented) of the camera positions can be calculated as well as the interior calibration of the camera and the used lens. The lower the calculated error, the better the following result of the textured models.

To create a mesh model, one image pair showing the same area of the cubiculum respectively the painting is loaded into the software 3DM Analyst [3] and a textured mesh model is created. To cover a whole cubiculum, many partial models have to be generated. Since these models overlap in many areas, it is important to create a consistent and closed surface model. By exporting the vertices of each partial model from 3DM Analyst and importing them into Geomagic, a closed and, even more important, clean mesh model can be created [72] [2]. To retexture the model (color information is lost during the export) the images taken for the modeling process together with their orientation matrices are used. This work is done by the 3DImage Software application [1].

Although for the capturing of the images, special light panels were used, it is necessary after the re-texturing process to adjust the single images in their color and brightness to hide the seams where images are stitched together. After finishing this work, the meshed and textured models can be imported into the rendering system without any further changes. It will be in the same coordinate system as the point cloud of the Domitilla catacomb, therefore it will be automatically placed at the right position. Figure 5.1 shows the sequence of the single steps to generate textured mesh models.

There are more than 80 paintings spread all over the catacomb, and for some textured models, more than 60 images are used. They will be rendered with virtual texturing, which is described in more detail below.

5.1.2 Rendering Virtually Textured Models

The basic idea of virtual texturing was already described in Section 2.6. It enables to use a texture that is too large to fit completely into the memory of the graphics card. The complete virtual texture resides on disk. During rendering, a virtual texturing algorithm has to execute 3 main parts for every rendered frame:

- Determination of virtual texture tiles that are needed for rendering from the current viewpoint.
- Streaming these texture tiles into graphics memory.
- Mapping the texture from the loaded tiles to the geometry.

The texture coordinates of the geometry have to be adjusted so that they point to the right areas in the virtual texture. The buildup of the virtual texture and the offsetting of the geometry's texture coordinates are done offline in a pre-process. When the tiles are streamed to the graphics card, they are stored in the physical texture. The physical texture has a certain amount of free cells into which tiles can be loaded. During rendering, this texture is bound when the fragment shaders are called and is used for all texture lookups of the geometry. Since the texture coordinates of the loaded tiles are not the same in the physical texture as they are in the virtual texture, a mapping between texture coordinates is necessary, and this is provided by the pagetable texture. In the pagetable texture, each texel corresponds to exactly one tile of the virtual texture, including the complete mipmap chain. The pagetable texture stores the physical texture coordinates of the corresponding virtual texture tile. If a tile is not available in the physical texture, a fallback entry can be stored instead, e.g., the coordinates of the next available lower-resolution mipmap level. It is possible to hold the tiles of the highest levels (e.g., levels n, n-1, and n-2) always in the physical texture, so when loading higher-resolution tiles, these tiles can be used for rendering until the appropriate tiles have arrived. Note that the physical texture is not mipmapped, it only stores tiles of different mipmap levels of the virtual texture.

5.1.3 Determination of Virtual Texture Tiles

The determination of the virtual texture tiles is done per pixel in a pre-rendering pass. This exact tile determination in view space is possible by rendering the scene into the frame buffer,

and storing in each pixel the information which tile coordinates to use for texturing this pixel, including the mipmap level. In the OpenGL API the mipmap level selection is not defined in the specification, it is left to the actual OpenGL implementation. Therefore, one cannot always rely on OpenGL to calculate the mipmap level. Instead, a separate texture lookup can be performed in the fragment shader that samples a mipmapped texture, which has stored the mipmap level in its texels. The result of this pre-rendering pass is an "image" in the frame buffer that contains in its pixels the addresses of all tiles that are necessary for texturing the currently visible geometry. Note that the tile addresses might not (and often will not) be unique, as a tile usually covers more than just one pixel in the scene. On the other hand, this is even necessary for the virtual texturing algorithm to work, as it relies on spatial coherency for texturing the geometry.

When this pre-rendering pass is finished, the frame buffer is read back to the CPU. There, the tiles which shall be loaded from disk are extracted from the buffer, and the corresponding requests are sent to the tiles-loading thread.

5.1.4 Streaming Tiles into Memory and Updating the Physical Texture

Streaming the tiles from disk into memory happens after determining which tiles are visible in the current frame. Streaming is done in a background thread, which loads the tiles asynchronously to the rendering thread. After loading, the tiles have to be decompressed, as they are stored in a compressed image format like JPEG or PNG. If desired, the tiles can then be re-compressed to a format that is readable by the GPU, like the DXT compression formats. This reduces the memory requirements in graphics memory, but it also reduces the visual quality of the texture tiles. Decompression and re-compression can be done in a third thread.

Finally, the pagetable texture has to be updated to reflect the changes in the physical texture. When a tile becomes available in the physical texture, its coordinates are stored in the corresponding texel in the pagetable texture. The fallback entries in the pagetable texture also have to be updated. Updates are done from the CPU, as the information which tiles are available in graphics memory is also managed by the CPU.

The next step is to upload the newly loaded tiles to the physical texture on the graphics card. The physical texture is stored in a way that hardware filtering can occur when sampling the texture, i.e., as a 2D texture or as a texture array. Since current graphics drivers do not support enough layers for texture arrays to store all tiles of a physical texture, a 2D texture is used per default, where the tiles are tightly packed in a grid layout. When there is no free position in the physical texture for the newly loaded tiles, then tiles can be replaced according to the LRU paradigm [25], where those tiles which were not used for the longest time are replaced by the new tiles.

5.1.5 Map Tiles onto Geometry

During rendering, the virtual texture shader, which is a fragment shader, transforms the virtual texture coordinates of the geometry to coordinates in the physical texture, in which the tiles are stored on the graphics card. It does so by first sampling the pagetable texture (to get the texel that holds the coordinates of the corresponding tile in the physical texture). All virtual texture coordinates that refer to one tile fetch the same texel from the pagetable texture. When the

Figure 5.2: The cubiculum of the Fossor Diogenes [113] in the context of the adjacent hallways. The hallways are rendered as point cloud, while the photos of the paintings in the cubiculum are stored in a virtual texture, which is mapped onto a mesh model. In the left image, the virtually textured mesh is blended into the point cloud, while in the right image, the point cloud is marked, so the virtually textured mesh can be easily distinguished.

coordinates of the tile are known, the correct offset within this tile is calculated and added to obtain the final physical texture coordinates, and this texture is then sampled to color the current pixel. Note that if the entry in the pagetable texture points to a fallback entry, the tile of the fallback entry will be used for texturing instead.

5.1.6 Further Considerations about Virtual Texturing

Although the size of a virtual texture is not limited per se, there are currently size limitations due to restrictions of the graphics card hardware. One limiting factor is the length of the mipmap chain. Each additional level increases the size of the pagetable texture by a factor of 4. A mipmap chain of length 11 results in a pagetable texture size of already 1024^2 texels. For larger pagetable texture sizes, a considerable performance hit occurs when updating the pagetable. Other limiting factors are the tile size and physical texture size. Since the physical texture stores all tiles that are needed for texturing the scene from the current viewpoint, the physical texture has to be large enough to provide space for the currently visible tiles. For a complex scene with many different sub-textures visible at the same time, it is advantageous to have many small tiles rather than only a few large tiles. The side length of a tile is always an integer and a power of 2. The tile size cannot be arbitrarily small, as decreasing the tile size one step increases the number of I/O operations when loading the tiles from disk, and also increases the size of the pagetable texture by a factor of 4. Therefore, reasonable tile sizes are from 64^2 to 256^2 texels. The physical texture can be chosen as large as the maximum texture size that can be handled by the GPU, which is currently about 8192^2 texels. From these considerations results a maximum virtual texture size of $262{,}144^2$ texels for the highest resolution level with a mipmap chain length of 11.

Creating virtual textures is a problem for common image manipulation programs like Photoshop [85] or Gimp [35], since these do not support images with a size of $64{,}000^2$ pixels or more, therefore such images usually cannot even be imported or exported. Because of this, the

Figure 5.3: An inside view of the cubiculum of the Fossor Diogenes [113]. The texture consists of the original photos, which were captured to create the mesh model by photogrammetric reconstruction techniques.

virtual texture has to be created in a distinct application, and cannot be edited interactively. Also, viewing virtually textured models in common 3D-suites like Blender [11] or Maya [69] is not possible, since these programs do not support virtual texturing out of the box. Instead, a set of Python scripts is used to arrange the original images into a texture atlas, which is then divided into tiles, and all mipmap levels for these tiles are created.

5.2 Accelerating the Tile Determination

In this section, an enhancement to the tile-determination step of the virtual texturing algorithm is proposed, which compresses the list of tiles visible from the current viewpoint, thus removing all redundancy. This accelerates the tile-determination step by up to a factor of four in the tested scenarios. This enhancement was developed in the course of a master's thesis by Mayer [70]. It is not peer-reviewed, because similar techniques, which were developed almost simultaneously, had already been published.

The performances for the figures in this section were measured with a Mac Pro Quad-core 2006 with 2 Intel Xeon DP 5150 CPUs, 7 GB main memory, a GeForce 8800GT 512 MB graphics card, and a 7200 RPM hard disk.

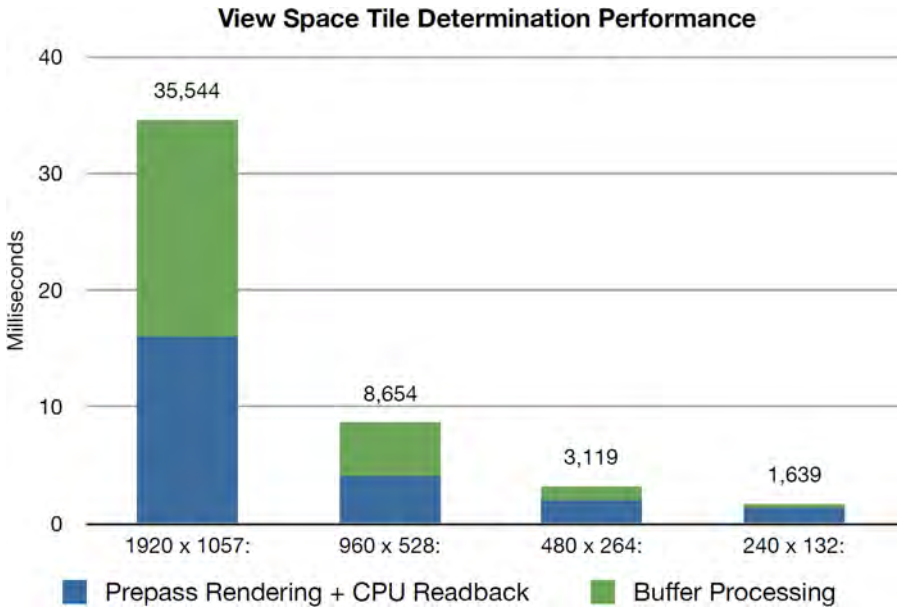


Figure 5.4: Timings for the tile-determination rendering pass and the following read-back of the buffer to main memory at four different frame buffer resolutions. From left to right, these are full (1920 x 1057), half, quarter, and eighth resolution.

5.2.1 Exact Tile Determination

The tile-determination step is the first of the three steps that a virtual-texturing algorithm has to perform every frame during rendering (see also Section 5.1.2). It finds the tiles that are visible in the current frame. Depending on the set of tiles found in this step, the subsequent tile-streaming step requests the tiles that are not available on the graphics card from external memory.

There are different overall strategies that can be used for the tile determination. Possible strategies include aggressive, conservative, exact, and approximate ones. An aggressive strategy does not find all required tiles for a frame, for the sake of increased performance. Because of this, artifacts can occur, as not always the best mipmap level of a tile is available during rendering. A conservative strategy, on the other hand, will choose more tiles than necessary to load from external memory, i.e., also tiles currently not visible will be loaded. While this can have a positive effect when moving through a virtually textured model, as tiles currently not visible might become visible in a later frame, more tiles have to be loaded to the physical texture, so the frequency at which tiles are changed in the physical texture increases as well. Depending on the access time to external memory, this can have a significant impact on loading times. An exact strategy will find exactly those tiles that are currently visible in the frame. An approximate strategy does not use all information available to choose the tiles for rendering, but instead uses an approximation of the visibility calculations, for example the distance from the viewpoint to the polygons that shall be textured. This can be a viable strategy in terrain rendering systems.

In implementations where the occlusion between objects is considered, tile determination might become a complex problem. When aiming for an exact tile determination, the occlusion

between objects can be resolved by using a z-buffer.

The virtual texturing system in this thesis uses the exact tile-determination strategy. It is implemented by doing an additional pre-rendering pass, where a fragment shader calculates the virtual texture coordinates and mipmap levels of the tiles that are used for texturing the fragments. The fragment shader stores this information in a buffer on the graphics card. After the pre-rendering pass, this buffer holds a list of tiles, but it also contains a lot of redundancy. This is on the one hand desirable, as more redundancy means less tiles have to be used for texturing the fragments of the current frame (see also Section 5.1.3). On the other hand, as the buffer has to be transferred from the graphics card to main memory (where the list of unique tiles is extracted from the buffer), more data then necessary has to be moved. Transferring the buffer is nevertheless required, so the tiles missing on the graphics card can be requested from external memory. Depending on the resolution of the frame buffer, moving the result buffer can cause a considerable performance hit.

Figure 5.4 shows the performance of the tile-determination rendering pass and the following read-back of the result buffer at four different resolutions. At full resolution (1920 x 1057), about 35 milliseconds are required, while at half resolution only about 8 milliseconds are required. According to this result, using a frame buffer with half the resolution will increase the performance notably, at the cost of a somewhat reduced accuracy. Tiles that are only visible in very small areas might be missed and tiles of a coarser mipmap level have to be used for texturing those areas, but on the other hand, less tiles have to be loaded to the graphics card. Another advantage is that after the tile-determination pass, a smaller buffer has to be moved from the graphics card to main memory. So for practical reasons, a half-resolution frame buffer (or an even smaller one but with increased artifacts) should be used for the tile-determination pass.

5.2.2 GPGPU Buffer Compression

Instead of moving the whole buffer created in the tile-determination pass to main memory, the list of unique tiles can also be extracted from the buffer directly on the graphics card, by using a GPGPU approach. The GPGPU capabilities of a graphics card can be accessed by an API like OpenCL [55] or CUDA [78]. In this thesis, the OpenCL API is used to process the buffer on GPU. The list of unique tiles is small, with roughly only few hundred entries, therefore the amount of data that has to be moved to main memory is reduced significantly, and the transfer time becomes negligible.

The creation of the list is divided into two parts. First, a uniquification of the tiles stored in the tile-determination buffer is performed, and second, the unique tiles are sorted into the list. Each of the two steps is implemented as an OpenCL kernel. The uniquification of the tiles is done by iterating over the elements in the buffer, where the elements store the coordinates and mipmap level of a tile per fragment. The frame numbers, which indicate when each tile was last accessed, are stored in a data structure that resembles the virtual texturing page table. It has the same number of mipmap levels, and the same number of elements per level, and the tile coordinates correspond to the virtual texturing page table as well, but instead of storing the physical texture coordinates of the tiles, it stores the frame numbers the tiles were last accessed. So when iterating over the buffer, the frame numbers in this data structure are updated for the tiles visible in the current frame. After this unification, the second kernel traverses this “frame

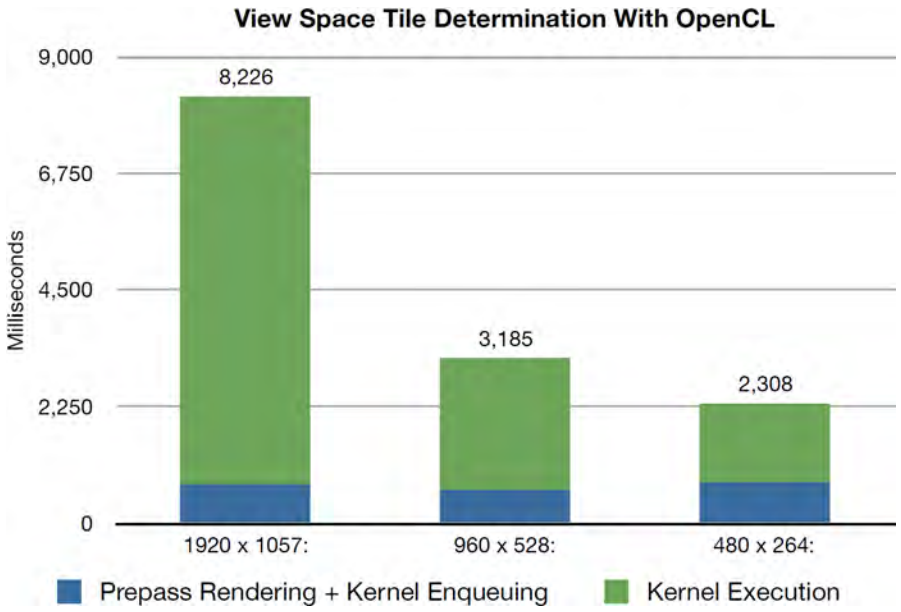


Figure 5.5: Timings for the tile-determination rendering pass and the following read-back of the buffer to main memory at four different frame buffer resolutions with GPGPU buffer reduction. From left to right, these are full (1920 x 1057), half, and quarter resolution.

numbers page table", and sorts the indexes of the tiles visible in the current frame into a list. This list is then transferred to main memory. Note that if using a byte per tile to store the frame numbers, the frame numbers have to be wrapped every 255 frames. This requires to reset all elements in the frame numbers page table to 0, and start counting the frames from 1 again.

Hollemeersch et al. [50] presented a similar approach, implemented in CUDA. While the first kernel was developed independently of their work, the second kernel is directly based on their sample code.

Figure 5.5 shows the performance of the GPGPU buffer-compression method. It can be seen that the time needed for the full resolution is about 4 times faster compared to the method where the buffer is processed by the CPU and it is as fast as using only half resolution with the previous method (see also Figure 5.4). Therefore, GPGPU buffer compression can be used to improve the accuracy of the tile-determination step while still maintaining the same performance. Nevertheless, the information about the number of visible fragments per tile is lost. This can be accounted for by storing the number of visible fragments for each tile at the cost of increased memory usage.

Instead of using the first kernel to update the frame numbers page table, the new OpenGL extension `ARB_shader_image_load_store` could be used in the fragment shader during rendering, as it allows random write accesses to textures. This would save memory and probably also time, but was not tested.

5.3 Results

The virtual-texturing algorithm was implemented in a separate library [71] in the course of the master's thesis by Mayer [70]. The library was integrated into the existing point rendering framework. The virtually textured models are separate nodes in the scene graph [80], and are rendered as part of the scene graph rendering traversal.

All textures of the mesh models were stored in a virtual texture of 131,072x131,072 pixels resolution. For testing, the test system 1 described in Section 3.11 was used, except for the graphics card, which was replaced by a GeForce 285GTX with 1 GB VRAM. In a walkthrough through the Domitilla catacomb model with $1.92*10^9$ points and 29 virtually textured mesh models, the frame rate never dropped below 10 frames-per-second, although 2 separate threads for loading were used, one for loading points and one for loading textures. The complete point-cloud data uses 30GB on disk (in about 260,000 files), and the virtual texture uses 6GB on disk (in about 1.4 million JPEG files). Figure 5.2 shows a virtually textured model as it is rendered with the combined point and mesh rendering framework. The left image shows the point cloud of the Domitilla catacomb and the virtually textured model displayed at the same time. In the right image, the points are marked, so that the position and the boundary of the virtually textured model becomes easily distinguishable. When changing the position of the viewpoint such that it is "inside" the virtually textured model, as shown in Figure 5.3, the fine details of the paintings on the wall become visible.

5.4 Summary

For regions of an object where the surface displays a lot of details, it can be preferable to capture these details with a camera, and reconstruct the geometry of the object from the taken images. The same images can then be used to texture the reconstructed object. For providing high-resolution textures, the texture data can be managed out-of-core by a virtual texturing algorithm. It is furthermore possible to combine virtually textured models with out-of-core managed point clouds in a single virtual scene, which results in a display of areas with high surface details that are embedded in a point model of huge extent.

From the three steps that are performed by a virtual texturing algorithm every frame, the first step, which is the tiles determination step, has to extract the list of unique tiles visible in the current frame from the information stored in the frame buffer created in a pre-rendering pass. The list can be extracted by calculations on the CPU, which requires to move the whole frame buffer from graphics memory to main memory. The extraction of the list can be sped up by doing the calculations directly on the GPU, thus avoiding the transfer of the complete frame buffer to main memory. Instead, only the list of unique tiles has to be transferred. This way, the accuracy of the tile-determination step can be improved while maintaining the same performance.

CHAPTER 6

Navigation

In the previous chapters, the technical foundations for handling large point models were laid, and also a combined display of large point models with virtually textured meshes was shown. These techniques are well suited for the visualization of data sets representing physically large structures or objects. In this chapter, a tool for navigating inside such data sets is proposed. As scanning objects of large extent becomes increasingly common, and since there are almost no limitations to the storage size of point models when using out-of-core managed data structures for rendering (except for the size of the external memory), those point models can represent areas of tremendous physical size, and the issue of orienting oneself inside those point models becomes relevant. When navigating such areas, a tool which supports orientation can be very helpful.

6.1 Navigation Graphs

For the purpose of navigating within huge virtual scenes, a graph-based guidance system was developed. It serves two purposes: first, it makes the exploration of a virtual scene less intimidating, which might be the case when exploring a large site, and second, it connects the most interesting landmarks and guides the user to these places. The virtual scene which the user explores is rendered in real time, so he can look around freely, but the navigation path is tied to a predefined 3D graph representing the possible directions.

6.1.1 Graph-based Guidance System

The problem that is tackled with a graph-based guidance system can best be described by an example. In Figure 6.1, two images of a point model of the Domitilla catacomb in Rome are shown. This point model was described in Section 4.1, and consists of about $1.92*10^9$ points. In the upper image, the complete point model is shown. The maze-like structure of the galleries is easily visible, but this structure is normally covered by the ground level, as most of the catacomb is dug into subsoil. Only some parts are also visible above the surface, like the upper part of the

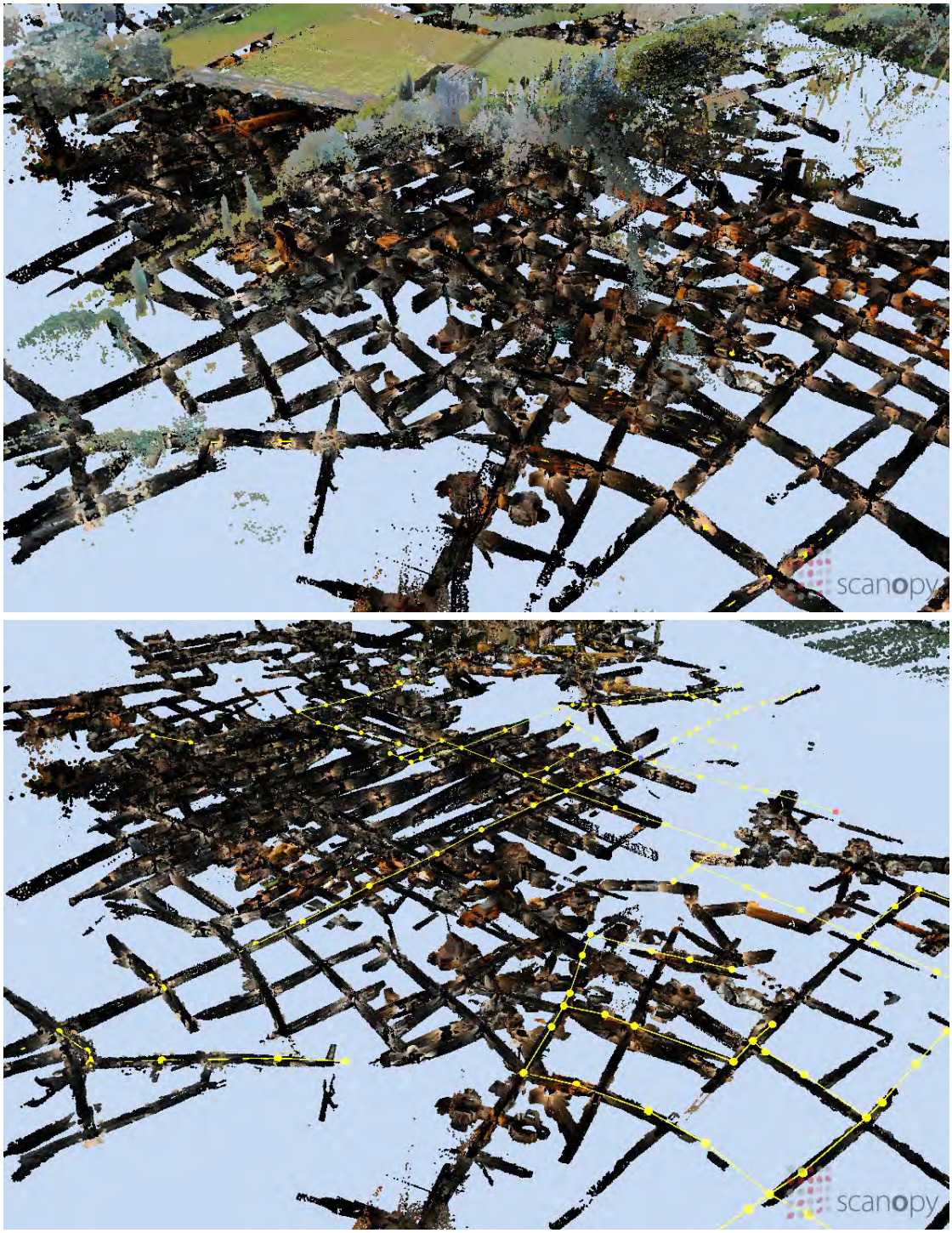


Figure 6.1: The upper image shows an overview of the point model of the Domitilla catacomb in Rome. The lower image shows a cut-away view of the point model from the same viewpoint and with a yellow navigation graph inside.

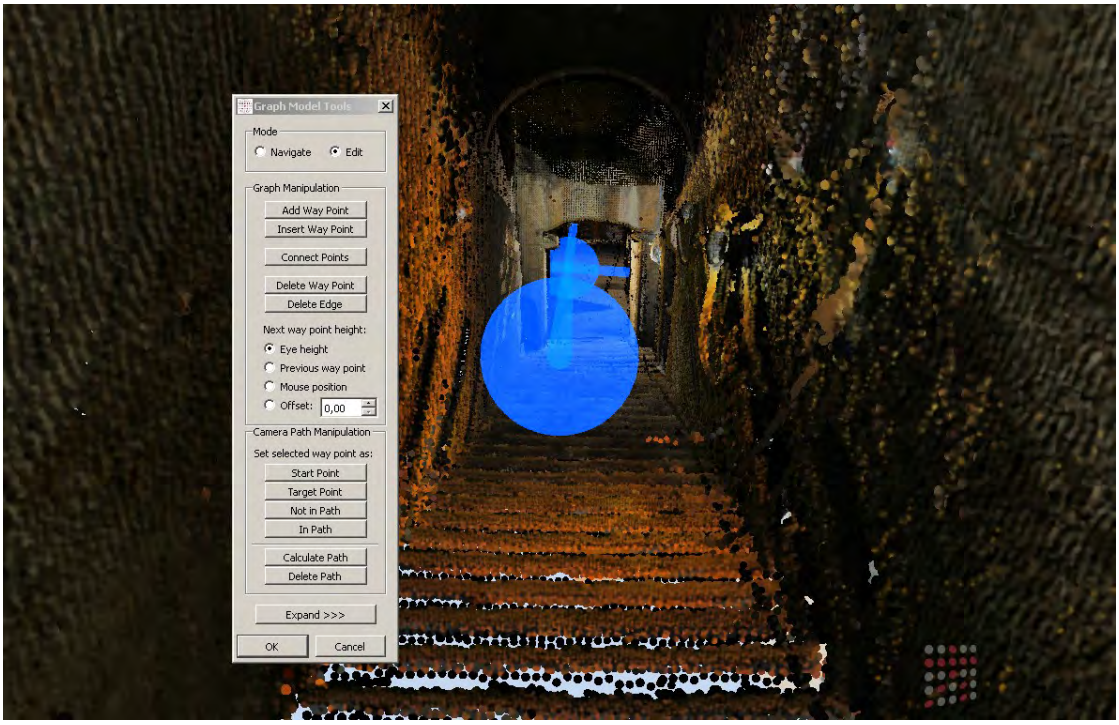


Figure 6.2: Editing options for a navigation graph.

basilica. Around the main entrance to the catacomb, also the ground level was scanned, and it is visible in the upper image (the green surface). Due to the huge size of the point model, one gets easily lost when trying to find a way through the catacomb. This is further complicated by the uniformity of the galleries, which often show no decorations. The problem of navigation is true in real life as well as when exploring the point model. What would be helpful when navigating this point model is a path the user can cling to, so he can avoid parts of these maze-like galleries which are not so interesting and he is instead guided to the places of interest. To help exploring the point model, the use of a 3D graph representation of the point model is proposed.

Note that the navigation graph is not a camera path. A camera path has the movement speed and the viewing direction for each position on the camera path predetermined, so following a camera path is not an interactive experience at all. The user is tightly bound to the pre-defined angles of view along the camera path, and he cannot explore the world around him. With the graph-based guidance, the user can look around freely and also choose, at least to some degree, which way he wants to go. The graph representation acts as a kind of rail system on which the user can move forward and backward, but which he cannot leave.

6.1.2 Target Audience

Before designing a guidance system, it is necessary to define the target audience which should be supported. This can be done by looking at what kind of user types are actually interacting with huge point models.

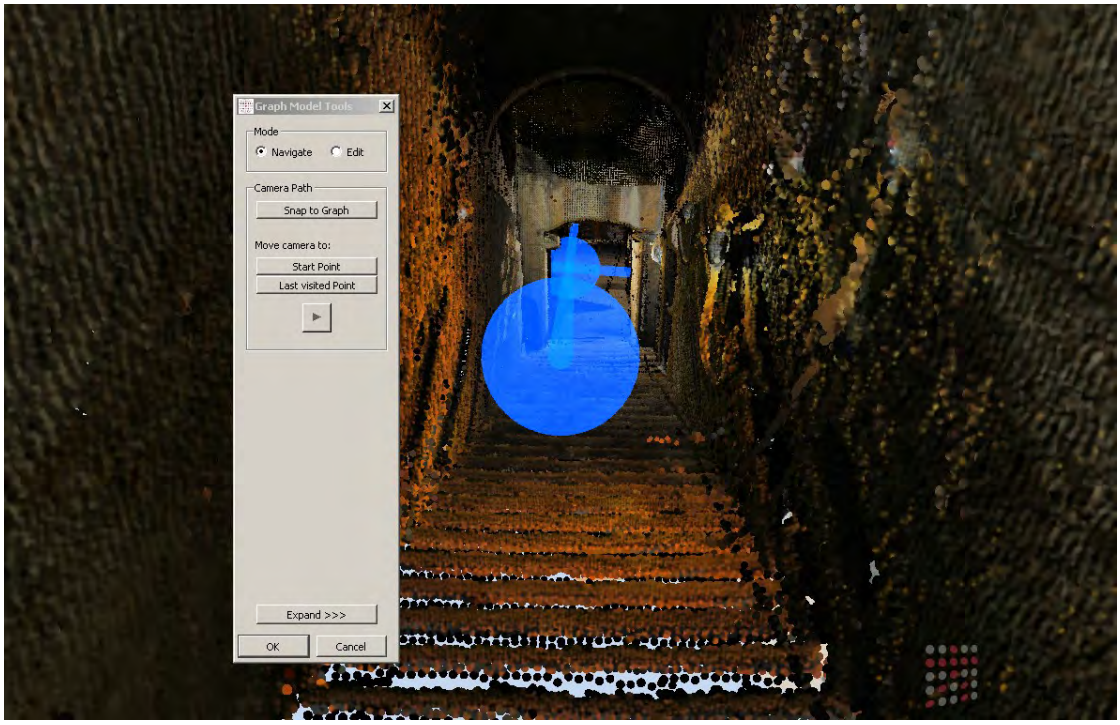


Figure 6.3: Navigation options for a navigation graph.

On the one hand, there is the expert user. This is the person who knows all about the scanned object, he is maybe the employer who ordered a scan from a specific site, or he is the person who scanned the site. He interacts often with the data to inspect it, to judge the quality of the scans, and so on. He knows the point model, the layout of the geometry, and he also knows the program controls, how to move around and interact with the data.

On the other hand, there is the interested layman, a person who can be described as a visitor in an exhibition. He will interact with the data for the first time, so it is probably more difficult for him to orient himself inside the model, and he also does not know where to search for the places of interest in the model. The graph-based guidance is targeted to aid the interested layman in exploring huge point models by presenting the most interesting ways through the data.

6.1.3 Preprocessing

In a preprocessing step, a 3D graph is created manually (see also Section 6.1.4). The graph consists of 2 sets, a set of nodes and a set of edges. Each node is connected to one or more other nodes via a respective edge. A node represents a junction or a change of direction, whereas an edge represents a straight gallery. The graph is not limited to lie in a plane, but can also have nodes at higher or lower levels. This gives enough flexibility to model the graph according to the layout of the point model. In the lower image of Figure 6.1 a cut-away view of the Domitilla catacomb model is shown. The top of the model is hidden to enable a view into the first floor of galleries. The yellow navigation graph imitates the galleries of the catacomb. Not all galleries

are part of the navigation graph, as the navigation graph only connects the places of interest. It currently consists of about 180 nodes, and covers about 15% of the galleries in the catacomb.

6.1.4 Guidance

The point model of the Domitilla catacomb is especially well suited for a guidance system, as it exhibits 12 kilometers of maze-like galleries distributed over 4 floors, where one can get easily lost. Finding the points of interest in this point cloud is a challenge. A navigation graph can provide the required help to keep the orientation when moving inside the point cloud.

The graph-based guidance system is integrated into the point rendering system developed for this thesis (see also Chapter 7). The user interface for the editing options for a navigation graph is shown in Figure 6.2. When editing a navigation graph, the user moves freely around the point model and sets so-called "way points". The way points are the nodes of the graph and represent either a junction or a change of direction. Way points can be added (i.e., a new edge connects to the new way point), inserted into an existing edge, or deleted. Furthermore, two arbitrary way points can be connected, and edges can be deleted. One problem that occurs during the creation of the navigation graph is assigning an appropriate position to the way point in 3D space. During the creation of the graph, the user can control a sphere-shaped cursor that moves along the surface of the point model. While this gives a good perception of the position of the cursor, there is a problem with this approach, as usually the way points are not set at the surfaces of the point model but in the center of the galleries in mid-air. Therefore, the vertical position of a way point can be set in four different ways. It can be set to the eye height (i.e., the center of the screen), to the height of the previous way point, to the height of the mouse pointer, or to the height of the mouse pointer with an additional offset.

In the editing options, a path can also be defined between two arbitrary nodes of the navigation graph, and this path is thought to give an automated camera movement along the navigation graph between a user defined start and end point. The path between the start and end point is calculated automatically, and always the shortest path is chosen.

When traveling on a navigation graph, the camera moves along the edges of the graph, i.e., the camera follows a linearly interpolated path between the nodes of the graph. Alternatively, splines [6] could be fitted to the nodes of the graph, i.e., the camera would move on splines between the nodes. This would require to layout the navigation graph according to the restrictions that are imposed by the splines. This was not tested, as the creation and editing of the navigation graphs should be kept as simple as possible. Therefore, in the current implementation, the camera travels exactly along the edges and nodes of the graph.

In Figure 6.3, the user interface for moving along a navigation graph is visible. The button "Snap to Graph" causes the camera to smoothly move to the closest position on the navigation graph that can be reached. The other two options move the camera to the start of the camera path and to the last visited way point, respectively. When pressing the "Play" button, the camera starts moving along the navigation graph, while the user can control the speed of the camera movement and the viewing direction by moving the mouse. It is also possible to reverse the movement of the camera by pressing the "UP" and "DOWN" arrow keys on the keyboard.

In Figure 6.4, the user interface is shown that appears when reaching a junction on the navigation graph. The movement of the camera stops, and arrows on the screen indicate the

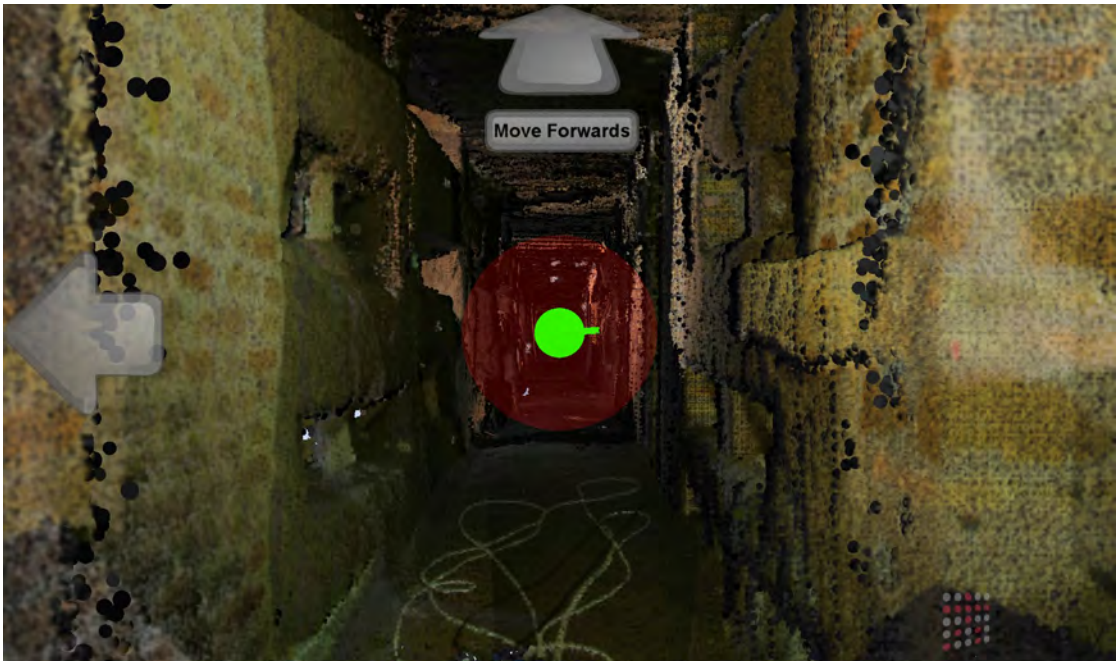


Figure 6.4: Choices of direction at a junction when travelling along a navigation graph.

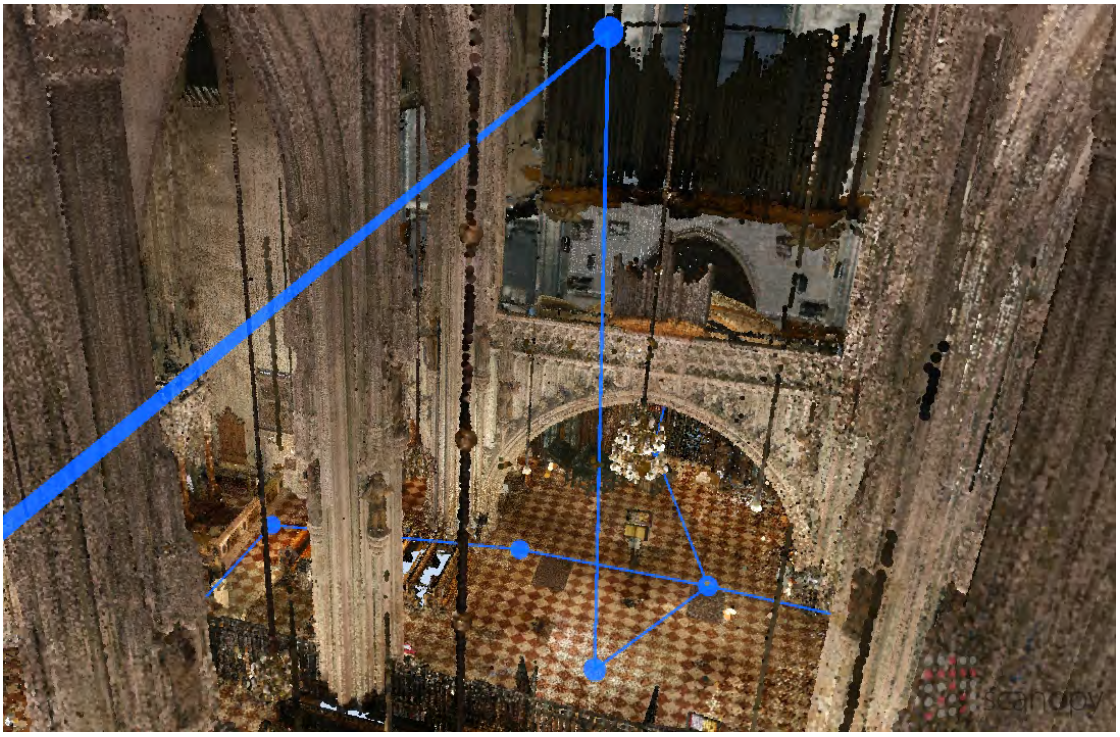


Figure 6.5: Interior of the Stephansdom point model with a blue navigation graph inside.

possible movement directions. The user can choose with the “LEFT” and “RIGHT” arrow keys, or by clicking the arrows with the mouse cursor, which path to follow. The view direction is then turned in the according direction to the next possible path. Having chosen the desired direction, the user can then press the “UP” key or click the “Move Forward” arrow on the screen with the mouse to start moving forward. The edge of the navigation graph in this particular example is shown in semi-transparent red, but this can be changed to a different color or transparency. The diameter of the navigation graph’s edges can also be changed, even down to zero, which causes them to disappear. These display options are also available for the nodes, so if nodes and edges are set to size zero, the user will not see the navigation graph at all.

Figure 6.5 shows a point model of the Stephansdom in Vienna, with the viewpoint hovering below the ceiling. The navigation graph inside the Stephansdom is not only laid out at the height inside which the visitors usually move, but it is also extended upwards. This is an example of how navigation graphs can be used to direct tourists along unusual paths, which might reveal never before seen perspectives.

6.2 Summary

For navigating virtual scenes of huge extent, navigation graphs can be used, which represent possible paths through such scenes by reducing the layout of the scene to a graph representation. The camera position can then only follow the edges and junctions of this navigation graph, while the camera speed and orientation can be freely chosen by the user. At the junctions of the graph, the user can choose which direction to follow. With the help of navigation graphs, users can keep their orientation, even if they are not familiar with the scene.

While currently the graphs are modeled manually, the modeling could be done automatically. In the point cloud of the Domitilla catacomb, this could be done by finding the medial axis of the galleries. Since the success of the automatic modeling is also dependent on the data quality of the point cloud, the manual modeling process can be used as a back-up in regions where the quality is not sufficient, e.g., in highly undersampled regions.

CHAPTER 7

Software Application

During the course of this thesis, a software application was developed that incorporates all presented algorithms, data structures, and interaction methods. Large parts of the application were developed from scratch, and care was taken to employ well-known design patterns [32] to handle the data and parameter flow between the different components of the application. The size of the complete source code grew to more than 100,000 lines of code, excluding 3rd party libraries, comments, and empty lines. In total, 12 people contributed to the development of the application, and they used it as a testbed for their praktika, bachelor's theses, master's theses, or PhD theses. While adding new algorithms to the application is only one part, the subsequent source-code refactoring to include the new algorithms seamlessly into the whole software system is an effort which takes time as well. While the ultimate purpose of developing an application for a thesis is testing new algorithms or data structures, the maintenance of the existing code base is important as well, as this way the results from previous stages can be used in upcoming developments. Therefore, well implemented algorithms and data structures are of great importance. In the following, a more detailed description of the software application, named Scanopy, will be given.

7.1 Development

The application is developed mainly in C++ and with OpenGL [56] as the graphics library. The OpenGL viewports (it is possible to show up to 4 viewports simultaneously) are encapsulated by OpenSceneGraph [80], a scene-graph library based on OpenGL, which is also used to organize the objects in the 3D scene, manage the view frustum, and determine the rendering order of the scene objects. For the user interface (UI) the Qt library [26] is used, which provides a UI framework for C++. The OpenSceneGraph viewports are wrapped as Qt widgets in Qt windows. The source code is developed in the Microsoft Visual Studio IDE, and can be compiled for 32-bit and 64-bit Microsoft Windows platforms. The source code revisions are managed in a Git [37] repository.

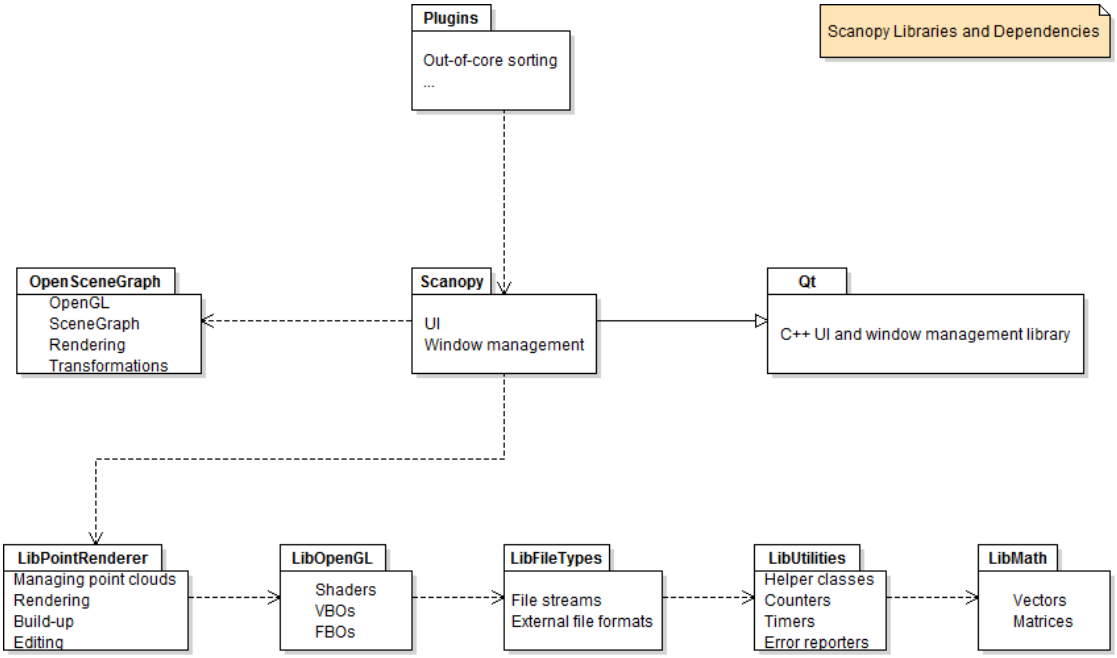


Figure 7.1: The main libraries and dependencies of Scanopy as packages in a class diagram. OpenSceneGraph and Qt are external dependencies, while the application itself, the plugins, and the other libraries were developed directly for Scanopy.

The development period of the application spans several years (about 10 when including preceding projects), and over the years, several insights were gained that made it possible to scale the development process, as more contributors began working on different parts of the source code. Most importantly, the source code became more and more modularized, either by implementing new algorithms as plugins, or by restructuring the class hierarchy, because initially only two large modules existed, the UI and the point rendering library. A modularized design requires more code for initialization, but the advantages gained in code maintenance outbalance these additional efforts by far.

7.2 Design

In Figure 7.1, the main modules and dependencies of Scanopy are shown in an UML class diagram [79] as packages. The main application Scanopy consists of a main window that is derived from a Qt class called QMainWindow. All other windows of the application are registered as children of the main window. Scanopy is dependent on OpenSceneGraph for the management of the different objects in the 3D scene. An OpenSceneGraph class called CompositeViewer is registered with Qt, which is responsible for displaying the different viewports in the Qt windows. It also provides an entry point for the scene graph traversal, which renders the objects of the scene. The algorithms and data structures for point rendering are collected in a separate library, called LibPointRenderer. The point clouds are all part of a single OpenSceneGraph Drawable node,

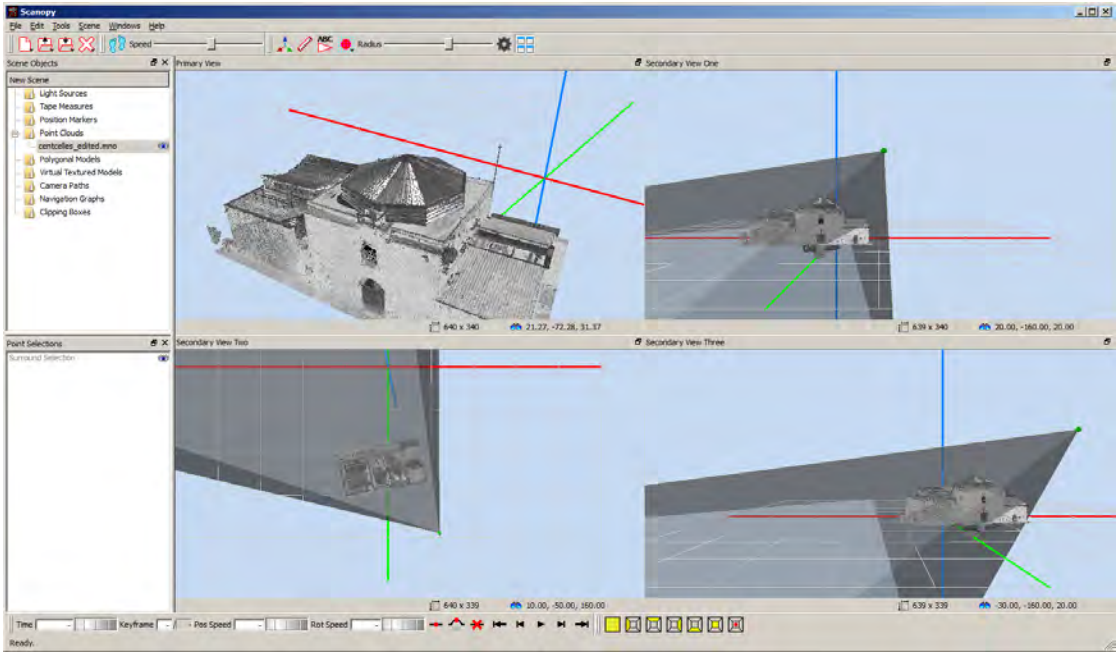


Figure 7.2: The user interface of Scanopy with 4 viewports. The primary view is in the upper left. The three secondary views show the camera and the view frustum of the primary view.

which is intended to render a single scene graph object. The LibPointRenderer, however, manages all point clouds on its own and does the point rendering with direct OpenGL calls, so it does not use OpenSceneGraph methods for rendering. Some OpenGL functions that require often-used initialization code are collected in the LibOpenGL, like shader loading, shader parameter management, VBO management, and so on. Other functionality, like file type classes, error reporting classes, and mathematics classes are collected in additional libraries. The plugins are implemented as Qt plugins, and they can access all Qt classes, as well as the LibPointRenderer classes and the classes of all the other libraries.

7.3 User Interface

Figure 7.2 shows the user interface of Scanopy with 4 viewports. A point model of the villa Centcelles, Spain, is rendered. It is colored with the reflection values of the recorded samples. Each viewport can have a different viewing position and direction, and the camera and viewing frustum of the primary view (which is the upper left viewport) is visualized in the other views.

In the upper area of the application window, a toolbar with buttons for several often used operations is attached. The buttons for loading and saving scene objects are on the very left, followed by the buttons for walk mode and speed (in walk mode, the mouse controls the camera, otherwise the mouse controls the currently active tool, for example the brush), and on the right, buttons for several interaction tools are placed, like the object movement tool, the measuring tool, the position marker tool, the brushing tool, and the slider for the size of the brush. The last

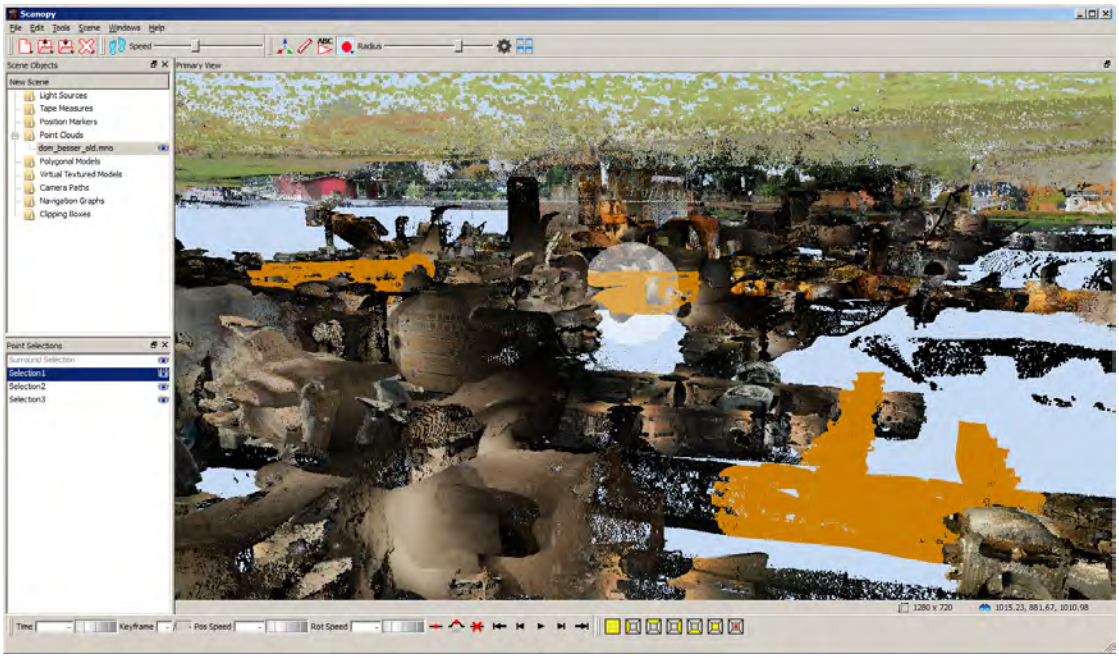


Figure 7.3: The user interface of Scanopy with one loaded point cloud. The Point Selection window in the lower left area shows 4 available selections. Selection 1 is the currently active selection, and the points belonging to this selection are rendered orange.

two buttons open the settings dialog and switch between 1 and 4 viewports respectively. The toolbar at the lower edge of the application window has on the left the elements for managing a camera path and on the right buttons for predefined viewing positions (which are dependent on the selected scene object).

In the upper left area of the application window, the so-called Scene Objects window is visible, where all objects of the current scene are listed. This includes point clouds, virtually textured meshes, normal (i.e., not virtually textured) meshes, camera paths, clipping boxes, navigation graphs, and more. The objects are listed in their according categories, and each object can be made visible or invisible by clicking the eye symbol right to its name. A scene can also be saved to disk, referencing all objects contained in the scene by their file path.

Figure 7.3 shows areas with selected points (orange) in the Domitilla catacomb point model. The view position is slightly below the turf, and in the center of the viewport the brushing sphere is visible.

In the lower left area of the application window, inside the so-called Point Selections window, a list of 4 selections is visible. This list indicates the current state of the available selections. Selection 1 is the currently active selection, and the visible points that are part of this selection are rendered orange. This coloring happens during rendering, so only the visible points are colored according to their selection state. When changing the active selection, the points of the then active selection will be rendered orange. The eye on the right side of the selection name can be clicked, and this toggles the visibility state of the points of this selection. An open eye means

that the points of the selection are visible (i.e., they are rendered), and a closed eye means the points are invisible (i.e., they are not rendered). This way, a preview without the selected points can be done, and it does not require to actually delete the points in the selection. The surround selection (the first item in the Point Selections window) contains all points that are not part of any other selection. It is always available. When activating the surround selection, all points not in any other selection will be rendered orange.

CHAPTER 8

Conclusion

Gigantic point clouds are increasingly used in industry, archaeology, museums, and other areas. These applications require visualizations of complex objects that support their comprehension. Point clouds allow views of these objects from angles which are not possible when exploring them in the real world. The source of gigantic point clouds are nowadays often laser range scanners, which measure the distances to their surrounding environment at discrete points, and thus create a cloud of point samples. Since the speed of acquisition and the density of these point clouds has increased a lot in recent years, point clouds consisting of 10^9 points and more are not uncommon.

In this thesis, a hierarchical data structure, called modifiable nested octree (MNO), was presented, which can handle gigantic point clouds in an external memory setting. This means that only the parts of the point clouds that are required for the current operation have to be loaded into main memory, while the complete point-cloud data is stored on external memory and can be orders of magnitude larger than the main memory. For rendering, the MNO provides an level-of-detail mechanism that does not use additionally created points, but distributes the original points throughout the hierarchy. It is also possible to add new points or delete existing ones from the hierarchy. Sorting a point data set before inserting it into an MNO often provides considerable time savings, depending on the sorting order and the available main memory during build up of the MNO. Build-up times for an MNO using sorted point clouds are an order of magnitude faster compared to build-up times from unordered point sets.

For selecting points, another data structure was presented, called selection octree. It describes the volume inside which all points of a point cloud are considered to be selected. The information about the volume is not stored at the points, therefore a selection octree can be used with point clouds in an out-of-core setting. The implementation of the selection algorithm when using a selection octree has to be designed with the possibility of selecting single points from a point cloud as well. When not taking care of this special case, the selection octree might be built-up with inaccurate borders, and wrong areas of the point cloud might be covered by the selection octree.

Furthermore, a point-size heuristic was presented, which adjusts the size of the rendered point splats depending on the density of the rendered points, and depending on the hierarchy levels of the rendered points in the MNO data structure.

The exploration of point clouds can help to understand the scanned objects, for example how they were created or which purpose they serve. For this task, several tools were presented. While viewing a complete point data set is already an enhancement over viewing only parts of a point data set at one time, which is often not possible without an out-of-core point-cloud viewer, other interaction methods like area-of-interest extraction or cuts through the point cloud provide even better understanding during exploration. Additionally, an improved rendering method which reconstructs the color of scanned objects from noisy point clouds reveals details of surfaces which are not visible otherwise. All these methods are available for out-of-core managed point clouds.

With virtual texturing it is possible to use very large and detailed images for texturing polygonal meshes. When combining virtually textured meshes with gigantic point clouds in a single virtual scene, it is possible to represent areas with colorful or very detailed surface features as textured meshes and the remaining areas as point clouds. This way, the extent of a large object can be made visible by the point cloud, whereas the most interesting areas are represented as textured meshes. Navigation through such large objects can be supported by predefined paths on a navigation graph, which connects places of interest. Inexperienced users can follow the paths on the navigation graph, so they do not get lost inside those large objects, as the movement will be restricted to the predefined paths.

With the algorithms, data structures, and interaction methods presented in this thesis, it is possible to render gigantic point clouds with off-the-shelf computers. Additionally, it is possible to edit them, explore them, and interact with them. The size of point data sets is ever growing, since the technology for creating point models from real-world objects matures and produces point models with higher density in shorter time. Therefore, the contributions of this thesis can help to conquer the point data sets of the future.

Bibliography

[1] Ahmed Abdelhafiz. 3DImage Software. https://sites.google.com/site/abdelhafizcv/cces/3DImage/ (accessed 2014-03-07), 2014.

[2] Ahmed Abdelhafiz, Irmengard Mayer, Gerold Eßer, and Norbert Zimmermann. Generating a Photo Realistic Virtual Model for the large Domitilla-Catacomb in Rome. In *Proceedings of the 9th Conference on Optical 3-D Measurement Techniques*, July 2009.

[3] ADAM Technology. Website. http://www.adamtech.com.au/, 2014.

[4] American Society for Photogrammetry and Remote Sensing. LASer (LAS) File Format Exchange Activities. http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html, 2014.

[5] Eric Andres. Discrete circles, rings, and spheres. *Computers and Graphics*, 18(5):695–706, 1994. ISSN 0097-8493.

[6] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to Splines for Use in Computer Graphics &Amp; Geometric Modeling*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[7] R. Bayer and E.M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

[8] Rudolf Bayer. B-tree and UB-tree. *Scholarpedia*, 3(11):7742, 2008.

[9] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[10] J.-Angelo Beraldin, Francois Blais, Luc Cournoyer, Michel Picard, Daniel Gamache, Virginia Valzano, Adriana Bandiera, and M. A. Gorgoglione. Multi-resolution digital 3d imaging system applied to the recording of grotto sites: the case of the grotta dei cervi. In Marinos Ioannides, David Arnold, Franco Niccolucci, and Katerina Mania, editors, *VAST06: The 7th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 45–52, Nicosia, Cyprus, 2006. Eurographics Association.

[11] Blender. Website. http://www.blender.org/, 2014.

[12] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-Quality Surface Splatting on Today's GPUs. In Marc Alexa, Szymon Rusinkiewicz, Mark Pauly, and Matthias Zwicker, editors, *Symposium on Point-Based Graphics*, pages 17–24, Stony Brook, NY, 2005. Eurographics Association.

[13] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In Markus Gross, Hanspeter Pfister, Marc Alexa, and Szymon Rusinkiewicz, editors, *Symposium on Point-Based Graphics*, pages 25–32, Zürich, Switzerland, 2004. Eurographics Association.

[14] Tamy Boubekeur, Florent Duguet, and Christophe Schlick. Rapid visualization of large point-based surfaces. In Mark Mudge, Nick Ryan, and Roberto Scopigno, editors, *VAST05: The 6th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 75–82, Pisa, Italy, 2005. Eurographics Association.

[15] Tamy Boubekeur and Christophe Schlick. Interactive out-of-core texturing with point-sampled textures. In Mario Botsch, Baoquan Chen, Mark Pauly, and Matthias Zwicker, editors, *Symposium on Point-Based Graphics*, pages 67–73, Boston, Massachusetts, USA, 2006. Eurographics Association.

[16] Stefan Bruckner and M. Eduard Gröller. Volumeshop: An interactive system for direct volume illustration. In *IEEE Visualization*, page 85. IEEE Computer Society, 2005.

[17] Kai Bürger, Jens Krüger, and Rüdiger Westermann. Direct volume editing. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1388–1395, 2008.

[18] Edwin Eearl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. thesis, Dept. of CS, University of Utah, December 1974.

[19] Timothy M. Chan. Closest-point Problems Simplified on the RAM. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 472–473, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[20] David Cline and Parris K. Egbert. Interactive display of very large textures. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 343–350. IEEE, 1998.

[21] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):599–608, July 2010.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[23] M. Cosman. Global Terrain Texture: Lowering the Cost. In Eric G. Monroe, editor, *Proceedings of IMAGE VII Conference*, pages 53–64. The IMAGE Society, June 1994.

[24] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential Point Trees. In *Proceedings of ACM SIGGRAPH 2003*, volume 22(3) of *ACM Transactions on Graphics*, pages 657–662. ACM Press, 2003.

[25] Peter J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM (CACM)*, 11(5):323–333, May 1968.

[26] Digia. Qt project. `http://qt-project.org/`, 2014.

[27] Florent Duguet, George Drettakis, Daniel Girardeau-Montaut, Jean-Luc Martinez, and Francis Schmitt. A point-based approach for capture, display and illustration of very complex archeological artefacts. In Y. Chrysanthou, K. Cain, N. Silberman, and F. Niccolucci, editors, *VAST04: The 5th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 105–114, Brussels, Belgium, 2004. Eurographics Association.

[28] Niklas Elmqvist, M. Eduard Tudoreanu, and Philippas Tsigas. Tour Generation for Exploration of 3D Virtual Environments. In *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*, VRST '07, pages 207–210, New York, NY, USA, 2007. ACM.

[29] FARO Technologies, Inc. Website. `http://www.faro.com/`, 2014.

[30] Raphael Ari Finkel and Jon Louis Bentley. Quad Trees A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4:1–9, 1974.

[31] Sarah F. Frisken and Ronald N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools*, 7(3):3–14, 2002.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[33] Vijay Gandhi, James M. Kang, and Shashi Shekhar. Spatial databases. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, USA, 2007.

[34] Geomagic. Website. `http://www.geomagic.com/`, 2014.

[35] Gimp. Website. `http://www.gimp.org/`, 2014.

[36] Matt Ginzton and Kari Pulli. Scanalyze a system for aligning and merging range data, 2002.

[37] Git-SCM. Website. `http://git-scm.com/`, 2014.

[38] 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.

[39] Google. Sparsehash. `https://code.google.com/p/sparsehash/`, 2014.

[40] Prashant Goswami, Fatih Erol, Rahul Mukhi, Renato Pajarola, and Enrico Gobbetti. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer*, 29:69–83, 2013.

[41] Prashant Goswami, Yanci Zhang, Renato Pajarola, and Enrico Gobbetti. High quality interactive rendering of massive point models using multi-way kd-trees. In *18th Pacific Conference on Computer Graphics and Applications (PG)*, pages 93–100, 2010.

[42] Ned Greene and Paul S. Heckbert. Creating Raster Omnimax Images From Multiple Perspective Views Using the Elliptical Weighted Average Filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, June 1986.

[43] Markus Gross and Hanspeter Pfister. *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[44] J. P. Grossman and William J. Dally. Point sample rendering. In George Drettakis and Nelson L. Max, editors, *Rendering Techniques '98, Proceedings of the Eurographics Workshop in Vienna, Austria, June 29 - July 1, 1998*, pages 181–192. Springer, 1998.

[45] Gabriele Guidi, Fabio Remondino, Michele Russo, Fabio Menna, and Alessandro Rizzi. 3d modeling of large and complex site using multi-sensor integration and multi-resolution data. In Michael Ashley, Sorin Hermon, Alberto Proença, and Karina Rodriguez-Echavarria, editors, *VAST08: The 9th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 85–92, Braga, Portugal, 2008. Eurographics Association.

[46] S. Havemann, D. W. Fellner, A. M. Day, and D. B. Arnold. New approaches to efficient rendering of complex reconstructed environments. In *International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 185–194, Brighton, United Kingdom, 2003. Eurographics Association.

[47] Paul S. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986. revised from Graphics Interface '86 version.

[48] Paul S. Heckbert. Fundamentals of Texture Mapping and Image Warping. M.sc. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June 1989.

[49] Michael Herf. Radix Tricks. `http://stereopsis.com/radix.html` (accessed 2014-03-07), December 2001.

[50] Charles Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle. Accelerating Virtual Texturing Using CUDA. *GPU Pro: Advanced Rendering Techniques*, pages 623–641, 2010.

[51] Herman Hollerith. Patent US395781 (A) - Art of Compiling Statistics. http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=395781&KC=&FT=E&locale=en_EP (accessed 2014-03-07), 1889.

[52] J.M.P. van Waveren and id Software. id Tech 5 Challenges - From Texture Virtualization to Massive Parallelization. http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf (accessed 2014-03-07), 2009.

[53] Aravind Kalaiah and Amitabh Varshney. Statistical geometry representation for efficient transmission and rendering. *ACM Transactions on Graphics*, 24(2):348–373, April 2005.

[54] Christian Kehl, Tim Tutenel, and Elmar Eisemann. Smooth, Interactive Rendering Techniques on Large-Scale, Geospatial Data in Flood Visualisations. In *Proceedings of ICT Open 2013*, 2013.

[55] Khronos Group. Opencl website. https://www.khronos.org/opencl/ (accessed 2014-04-11), 2014.

[56] Khronos Group. Opengl website. http://www.opengl.org/ (accessed 2014-04-11), 2014.

[57] Donald Ervin Knuth. Big Omicron and Big Omega and Big Theta. *SIGACT News*, 8(2):18–24, April 1976.

[58] Donald Ervin Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, volume 3. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2 edition, 1998.

[59] Boštjan Kovač and Borut Žalik. Visualization of LIDAR datasets using point-based rendering technique. *Computers & Geosciences*, 36(11):1443–1450, 2010.

[60] Oliver Kreylos, Gerald W Bawden, and Louise H Kellogg. Immersive Visualization and Analysis of LiDAR Data. In *Advances in Visual Computing*, pages 846–855. Springer, 2008.

[61] Jens Krüger, Jens Schneider, and Rüdiger Westermann. DUODECIM - A Structure for Point Scan Compression and Rendering. In Marc Alexa, Szymon Rusinkiewicz, Mark Pauly, and Matthias Zwicker, editors, *Symposium on Point-Based Graphics*, pages 99–107, Stony Brook, NY, 2005. Eurographics Association.

[62] Sylvain Lefebvre, Jérome Darbon, and Fabrice Neyret. Unified texture management for arbitrary meshes. Research Report 5210, INRIA, Rhône-Alpes, May 2004.

[63] Kurt Leimer. External Sorting of Point Clouds. Bachelor's thesis, September 2013.

[64] Marc Levoy. The Digital Michelangelo Project. In *3DIM99*, pages 2–11, 1999.

[65] Marc Levoy and Turner Whitted. The Use of Points as a Display Primitive. Technical Report 85-022, University of North Carolina at Chapel Hill, Computer Science Department, January 1985.

[66] Dani Lischinski and Ari Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In George Drettakis and Nelson L. Max, editors, *Rendering Techniques '98, Proceedings of the Eurographics Workshop in Vienna, Austria, June 29 - July 1, 1998*, pages 301–314. Springer, 1998.

[67] Luca Chittaro and Lucio Ieronutti and Roberto Ranon and Eliana Siotto and Domenico Visintini. A High-Level Tool for Curators of 3D Virtual Visits and ist Application to a Virtual Exhibition of Renaissance Frescoes. In Alessandro Artusi, Morwena Joly, Genevieve Lucet, Denis Pitzalis, and Alejandro Ribes, editors, *VAST10: The 11th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 147–154, Paris, France, 2010. Eurographics Association.

[68] Making Art Studios. Lightning Engine - Virtual Texturing. `http://blog.makingartstudios.com/?tag=virtual_textures` (accessed 2014-03-07), 2008.

[69] Maya. Website. `http://www.autodesk.com/products/autodesk-maya/`, 2014.

[70] Albert Julian Mayer. Virtual Texturing. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, October 2010.

[71] Albert Julian Mayer. LibVT. `http://sourceforge.net/projects/libvt/` (accessed 2014-03-07), 2013.

[72] Irmengard Mayer. 2D-Texture builds 3D-Geometry. New Approaches in Documenting Early-Christian Mural Paintings at the Domitilla Catacomb in Rome. In *Proceedings of the 14th International Congress on Cultural Heritage and New Technologies*, 11 2008.

[73] Irmengard Mayer, Claus Scheiblauer, and Albert Julian Mayer. Virtual texturing in the documentation of cultural heritage. *Geoinformatics FCE CTU*, September 2011.

[74] Microsoft. TechNet Webpage - How NTFS works. `http://technet.microsoft.com/en-us/library/cc781134(v=WS.10).aspx` (accessed 2014-03-07), 2003.

[75] Martin Mittring and Crytek GmbH. Advanced Virtual Texture Topics. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 23–51, New York, NY, USA, 2008. ACM.

[76] G.M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. Technical report, International Business Machines Ltd., Ottawa, Canada, 1966.

[77] NVIDIA. Game & Graphics Development Webpage - Texture Atlas Whitepaper. https://developer.nvidia.com/content/texture-atlas-whitepaper (accessed 2014-03-07), 2011.

[78] NVIDIA. Cuda website. http://www.nvidia.com/object/cuda_home_new.html (accessed 2014-04-11), 2014.

[79] Object Management Group. Unified modeling language. http://www.uml.org/, 2014.

[80] OpenSceneGraph. Website. http://www.openscenegraph.org/ (accessed 2014-03-07), 2014.

[81] Oxford Dictionaries. Oxford University Press. lidar. http://www.oxforddictionaries.com/definition/english/lidar (accessed 2014-03-07), January 2014.

[82] Renato Pajarola, Miguel Sainz, and Roberto Lario. Xsplat: External memory multiresolution point visualization. In *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing*, pages 628–633, 2005.

[83] PassMark Software. Memory Benchmarks. http://www.memorybenchmark.net/, 2014.

[84] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[85] Photoshop. Website. http://www.photoshop.com/, 2013.

[86] Alice Pignatel, Fausto Brevi, and Sebastiano Ercoli. Real-time Visualization of the Forum of Pompeii. In *Making history interactive: Computer Applications and Quantitative Methods in Archaeology (CAA); Online proceedings of the 37th international conference*, March 2009.

[87] Ruggero Pintus, Enrico Gobbetti, and Marco Agus. Real-time Rendering of Massive Unstructured Raw Point Clouds using Screen-space Operators. In Franco Niccolucci, Matteo Dellepiane, Sebastián Peña Serna, Holly E. Rushmeier, and Luc J. Van Gool, editors, *VAST 2011: The 12th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage, Prato, Italy, 2011. Proceedings*, pages 105–112. Eurographics Association, 2011.

[88] Fabio Remondino, Stefano Girardi, Alessandro Rizzi, and Lorenzo Gonzo. 3D Modeling of Complex and Detailed Cultural Heritage Using Multi-resolution Data. *Journal on Computing and Cultural Heritage (JOCCH)*, 2(1):2:1–2:20, July 2009.

[89] Rico Richter and Jürgen Döllner. Out-of-Core Real-Time Visualization of Massive 3D Point Clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '10, pages 121–128, New York, NY, USA, 2010. ACM.

[90] Riegl Laser Measurement Systems. Website. `http://www.riegl.com/`, 2014.

[91] Szymon Rusinkiewicz and Marc Levoy. Qsplat a multiresolution point rendering system for large meshes. In *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)*, pages 343–352, New York, July 23–28 2000. ACMPress.

[92] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Elsevier, 2006.

[93] 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, November 2011.

[94] Claus Scheiblauer, Norbert Zimmermann, and Michael Wimmer. Interactive Domitilla Catacomb Exploration. In Kurt Debattista, Cinzia Perlingieri, Denis Pitzalis, and Sandro Spina, editors, *VAST09: The 10th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 65–72, St. Julians, Malta, 2009. Eurographics Association.

[95] Ruwen Schnabel, Sebastian Moeser, and Reinhard Klein. A Parallelly Decodeable Compression Scheme for Efficient Point-Cloud Rendering. In M. Botsch, R. Pajarola, B. Chen, and M. Zwicker, editors, *Symposium on Point Based Graphics*, pages 119–128, Prague, Czech Republic, 2007. Eurographics Association.

[96] Harold H. Seward. Digital Computer Laboratory Report R-232. Master's thesis, Massachusetts Institute of Technology, 1954.

[97] Jonathan Shade, Steven J. Gortler, Li wei He, and Richard Szeliski. Layered Depth Images. In *SIGGRAPH*, pages 231–242, 1998.

[98] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic, Revision 2008. `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4610933` (accessed 2014-03-07), 2008.

[99] Stanford Computer Graphics Laboratory. The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/` (accessed 2014-03-07), 2014.

[100] storagereview.com. Intel SSD 530 Review. `http://www.storagereview.com/western_digital_velociraptor_1tb_review` (accessed 2014-03-07), 2012.

[101] storagereview.com. Western Digital VelociRaptor 1TB Review. `http://www.storagereview.com/intel_ssd_530_review` (accessed 2014-03-07), 2013.

[102] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The Clipmap: A Virtual Mipmap. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 151–158. ACM SIGGRAPH, Addison Wesley, July 1998.

[103] Pierre Terdiman. Radix Sort Revisited. `http://codercorner.com/RadixSortRevisited.htm` (accessed 2014-03-07), 2000.

[104] Markus Tragust. Integrating annotations into a point-based rendering system. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, January 2014.

[105] Jeffrey Scott Vitter. Algorithms and Data Structures for External Memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008.

[106] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Roman Parys. Xgrt extensible graphics toolkit, 2009.

[107] 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 *Symposium on Point Based Graphics*, pages 37–45, Prague, Czech Republic, 2007. Eurographics Association.

[108] T. Weyrich, M. Pauly, R. Keiser, S. Heinzle, S. Scandella, and M. Gross. Post-processing of Scanned 3D Surface Data. In Markus Gross, Hanspeter Pfister, Marc Alexa, and Szymon Rusinkiewicz, editors, *Symposium on Point-Based Graphics*, pages 85–94, Zürich, Switzerland, 2004.

[109] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[110] Michael Wimmer and Claus Scheiblauer. Instant Points. In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136. Eurographics, Eurographics Association, July 2006.

[111] Chi Kuen Wong and Jurg Nievergelt. Upper Bounds for the Total Path Length of Binary Trees. *J. ACM*, 20(1):1–6, January 1973.

[112] Hui Xu and Baoquan Chen. ActivePoints Acquisition, Processing and Navigation of Large Outdoor Environments. Technical report, Department of Computer Science and Engineering University of Minnesota at Twin Cities, 2002.

[113] Norbert Zimmermann. *Zur Wiederentdeckung des Fossor Diogenes*. Tipografia Vaticana, Vatican, 2011.

[114] Norbert Zimmermann and Gerold Eßer. Showing the Invisible - Documentation and Research on the Roman Domitilla Catacomb Based on Image Laser Scanning and 3D Modelling. In *Proceedings of the 35th International Conference on Computer Applications and Quantitative Methods in Archaeology (CAA)*, pages 56–64, Berlin, Germany, 2007. Dr. Rudolf Habelt GmbH.

[115] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus H. Gross. Pointshop 3D an Interactive System for Point-based Surface Editing. *ACM Trans. Graph*, 21(3):322–329, 2002.

[116] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface Splatting. In ACM, editor, *SIGGRAPH 2001 Conference Proceedings, August 12–17, 2001, Los Angeles, CA*, pages 371–378, pub-ACM:adr, 2001. ACM Press.

[117] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective Accurate Splatting. In Wolfgang Heidrich and Ravin Balakrishnan, editors, *Proceedings of the Graphics Interface Conference, London, Ontario, Canada*, volume 62 of *ACM International Conference Proceeding Series*, pages 247–254. Canadian Human-Computer Communications Society, May 2004.

Curriculum Vitae

Personal Details

Name	Claus Scheiblauer
Date of Birth	January 30th, 1973
Place of Birth	St.Pölten, Austria
Languages	German, English
Nationality	Austria

Education

Vienna University of Technology

2007 – present	Doctoral Student, Informatics Department Dissertation: *Interactions with Gigantic Point Clouds* Advisor: Associate Professor Dr. Michael Wimmer
2006	Master of Science in Computer Sciences Master's Thesis: *Hardware-Accelerated Rendering of Unprocessed Point Clouds* Advisor: Associate Professor Dr. Michael Wimmer
1992 – 2006	Study in Computer Sciences
1991 – 1992	Study in Electrical Engineering

Bundesgymnasium und Bundesrealgymnasium St.Pölten

1983 – 1991	Secondary School

Grillparzer Volksschule St.Pölten

1979 – 1983	Primary School

Employment History

Vienna University of Technology

Jul. 2013 – present	Project Assistant in the EU FP7 Project Harvest 4D
Jan. 2010 – Jun. 2013	Project Assistant in the FFG FIT-IT Project TERAPOINTS
Jan. 2007 – Dec. 2009	Project Assistant in the FFG Bridge Project SCANOPY

Riegl Laser Measurement Systems GmbH

Aug. 2005	Internship

Reviewing

Conferences	Eurographics (EG) Computer Graphics International (CGI) Virtual Reality, Archaeology and Cultural Heritage (VAST) Computer Graphics Theory and Applications (GRAPP) Computer Graphics, Visualization and Computer Vision (WSCG) 3D Web Technology (WEB3D) Spring Conference on Computer Graphics (SCCG)
Journals	Transactions on Visualization and Computer Graphics (TVCG) Computer Graphics Forum (CGF) Journal on Computing and Cultural Heritage (JOCCH) The Visual Computer Journal (TVCJ)

Publications

M. Arikan, R. Preiner, C. Scheiblauer, S. Jeschke, M. Wimmer, Large-Scale Point-Cloud Visualization through Localized Textured Surface Reconstruction, to appear in *IEEE Transactions on Visualization and Computer Graphics*, 2014

C. Scheiblauer and M. Wimmer, Analysis of Interactive Editing Operations for Out-of-Core Point-Cloud Hierarchies, *WSCG 2013 Full Paper Proceedings*, pages 123-132, June 2013
`http://wscg.zcu.cz/DL/wscg_DL.htm`

C. Scheiblauer and M. Wimmer, Graph-based Guidance in Huge Point Clouds, *Proceedings of the 17th International Conference on Cultural Heritage and New Technologies*, ISBN 978-3-200-03281-1, November 2012
`http://www.chnt.at/proceedings-chnt-2012-post/`

C. Scheiblauer and M. Pregesbauer, Consolidated Visualization of Enormous 3D Scan Point Clouds with Scanopy, *Proceedings of the 16th International Conference on Cultural Heritage and New Technologies*, ISBN 978-3-200-02740-4, pages 242-247, November 2011
http://www.chnt.at/chnt-16-2011-proceedings/

I. Mayer, C. Scheiblauer, and A.J. Mayer, Virtual Texturing in the Documentation of Cultural Heritage, to appear in *Online Proceedings of the XXIII CIPA Symposium*, September 2011
http://cipa.icomos.org/index.php?id=69

C. Scheiblauer and M. Wimmer, Out-of-Core Selection and Editing of Huge Point Clouds, *Computers & Graphics*, 35(2), pages 342-351, April 2011
http://dx.doi.org/10.1016/j.cag.2011.01.004

C. Scheiblauer, N. Zimmermann, and M. Wimmer, Interactive Domitilla Catacomb Exploration, *Proceedings of the 10th VAST International Symposium on Virtual Reality, Archaeology and Cultural Heritage*, pages 65-72, September 2009
http://dx.doi.org/10.2312/VAST/VAST09/065-072

M. Wimmer and C. Scheiblauer, Instant Points, *Symposium on Point-Based Graphics 2006*, pages 129-136, July 2006
http://dx.doi.org/10.2312/SPBG/SPBG06/129-136