FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Normal Estimation Of Very Large Point Clouds

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Stefan Marek
Matrikelnummer 0026455

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Associate Prof. Dipl.Ing. Dipl.Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.Ing. Claus Scheiblauer

Wien, 04.02.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Abstract

This diploma thesis introduces methods for external sorting and fast k nearest neighbor searching for very large point clouds. Very large point clouds are datasets that can not be processed in main memory. This leads to certain requirements for any used reconstruction technique. The most important ones are out-of-core memory management and sequential data handling algorithms. The paper "Stream-Processing Points" by Renato Pajarola shows a way to design a framework which allows to process a subset of very large point clouds. A subset is defined as a spatially continuous region holding a predefined number of points. This diploma thesis is based on the aforementioned paper and improves on the stream processing pipeline presented therein. The proposed algorithm for searching the k nearest neighbors has a complexity of $O(N * log(M))$, where N are all points in the point cloud and M are all points in main memory, which is similar to current state of the art algorithms for in-core processed data sets.

Kurzfassung

In dieser Diplomarbeit werden Methoden vorgestellt um extern sortieren und um schnell die k nächsten Nachbarn in sehr großen Punktwolken finden zu können. Wobei sehr große Punktwolken jene Datensätze sind die nicht komplett im Hauptspeicher verarbeitet werden können. Das führt zu einigen Anforderungen die von solchen Methoden erfüllt werden müssen. Die wichtigsten Anforderungen sind out-of-core Speicherverwaltung und sequentielle Datenverarbeitungsalgorithmen. Die Arbeit "Stream-Processing Points" von Renator Pajarola zeigt einen Weg, um einen Entwurf so zu gestalten, dass nur eine kleine Menge der großen Punktwolke verarbeitet wird. Eine kleine Menge ist dabei definiert als eine räumlich kontinuirliche Region die eine vordefinierte Menge an Punkten beinhält. Diese Diplomarbeit basiert auf der vorher genannten Arbeit und verbessert den darin vorgestellten Datenstrom Verarbeitungsablauf. Der vorgeschlagene Algorithmus um die k nächsten Nachbarn zu finden hat eine Komplexität von $O(N * log(M))$, wobei N alle Punkte in der Punktwolke und M alle Punkte im Hauptspeicher sind, was gleich ist zu den im Hauptspeicher verarbeitenden Algorithmen die Stand der Technik sind.

Contents

CHAPTER 1

Introduction

This diploma thesis describes ways for external sorting and fast k nearest neighbor searching in a huge unprocessed point cloud. Unprocessed point clouds in this diploma thesis are a set of points that are sampled by a scanning process. Very often objects are scanned by a laser range scanner and the result is a point cloud. Those point clouds have only spatial information together with color information from a conventional camera. The relevant datasets in this diploma thesis are point clouds that have more points than will fit in main memory. This fact will lead to the requirement that out-of-core methods have to be found in order to process the point cloud. This means that only a fraction of the whole point cloud can be processed at a time. Also those methods can only take the locality of the data into account.

Recent developments show that points as rendering primitive are enough to visualize the point cloud [WS09], [LW85]. However if a more accurate surface estimation is needed then more than just the point coordinates are required. This means that so called surface elements [HPG00], [LRZ02], [MBK05] have to be used.

Surface elements consist of the point coordinate and additional information like the normal vector that gives the orientation of the point together with a radius which defines the propagation of that point in a three dimensional space.

Those surface elements are oriented points and can be used to splat the point onto the screen space with respect to their true orientation and propagation in the object space. In order to compute normals from an unprocessed point cloud the nearest neighborhood graph has to be found first. To be able to process very large point clouds that do not fit inside the main memory of the computer, this diploma thesis uses the framework suggested by [PS04], which allows to process subsets of huge point clouds that can be processed inside the main memory. This approach divides the process in two stages. First the point cloud is sorted along the main axis. Then the point streaming operators are applied sequentially to each of the sorted points. This diploma thesis will follow this

suggestions and improves the computation of the nearest neighborhood graph and the external sorting part.

The following introduction first defines how a point is defined in context of this diploma thesis and gives a short introduction into the content creation pipeline.

1.1 Point Definition

A point has many definitions and is very much related to a given coordinate system and the n-dimensional space it exists in. Without a defined coordinate system a point would be dimensionless. Coordinate systems are defined as a system for mapping coordinates to each n-tupel of a given n-dimensional space. These coordinates are furthermore called points. A point in a geometrical coordinate system has always a spatial meaning, which describes the exact position of that point inside the given coordinate system. Points in this master thesis will be used as 3-tuples of the Euclidean space, which are defined by a three dimensional Cartesian coordinate system. The term point will also be used to specify a rendering primitive which consists of a three dimensional spatial vector. A rendering primitive shows the exact position in a Cartesian coordinate system and optionally a color vector and a normal vector.

A valid point cloud is defined as a set of points, where no point has the same position as any other point in the given set of points. One element in a point cloud is basically a sampled surface point of a three dimensional geometric object. The term point cloud stands for a set of points which are related in a geometrical way. Those datasets are mostly acquired by using 3D-Scanners or by parametric surface equations which describe the shape of the surface best.

1.2 Point Based Computer Graphics

The work by [LW85] introduces the point as rendering primitive and discusses the different challenges one has to overcome when working with point based computer graphics. Therefore the different stages of the rendering pipeline for point based computer graphics are described.

The work by [HPG00] defines the term surfel and shows a way to splat points from the object onto the screen space. The term surfel stands for surface elements. It defines oriented points that can be interpreted as surface representation at a given position in a three dimensional space. The surfel are projected onto the screen space by rendering a depth image of the points into z-buffer. This is called visibility splatting and can be seen in Figure 1.1

[LRZ02] and [MBK05] go one step further by expanding the idea with elliptical weighted average (EWA) surface splatting. The algorithm consits of two rendering

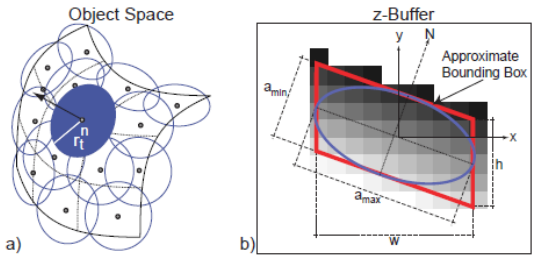

Figure 1.1: Visibility spatting [HPG00].

passes. The first pass is the same as the afore mentioned visibility splatting. The second pass applies an EWA filter on each of the splatted points. This EWA filter is a Gaussian filter that scales the point splats to the proper size.

1.3 Content creation pipeline

In order to reconstruct surface features the following content creation pipeline has to be considered. The content creation pipeline in Figure 1.2 shows which stages from the acquisition to the final surface reconstructed point cloud have to be considered.

- The first and most basic stage in the pipeline scans an environment and produces a point cloud from the acquired data. This stage specifies the quality of the point cloud and is therefore the most important stage in the pipeline.
- The second stage is the surface reconstruction stage. This stage tries to reconstruct the surface locally for any point in the point cloud.
- The third stage is the processing and modeling stage. At this stage the user can select what parts of the point cloud are outliers and what parts have to be remodelled.
- The last stage is the rendering stage where the point cloud is rendered in an graphically based editor.

3D-Acquisition

There are many ways to sample the environment for spatial information and the quality of the resulting data is very much related to the used method and the specific field the

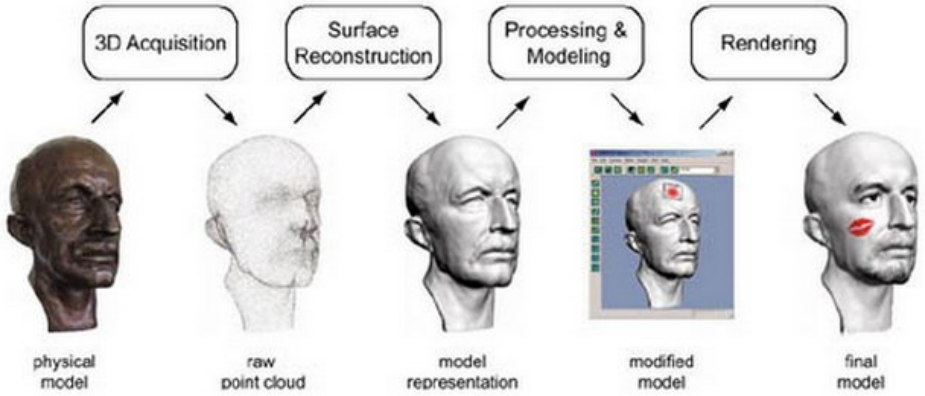

Figure 1.2: 3D content creation pipeline.

acquired data will be used. This thesis is related to datasets given by non-contact reflective optical shape acquisition methods, especially to those given by active triangulation and time of flight. The Figure 1.3 shows some of the scannable objects.

The most popular 3D-scanners for optical triangulation are laser range scanners. Those types of 3D-scanner measure the time needed to send a laser beam from an emitter to a solid surface of an object and from the surface to a sensor. This information will be used to calculate the distance by using optical laser triangulation which can be formulated by a set of basic trigonometric equations as can be seen in Figure 1.4.

Another 3D laser range scanner is the time of flight scanner. A time of flight laser scanner measures the time for one round trip of a laser pulse. Figure 1.5 shows one round trip from a time of flight scanner to the object and back.

Another information which can be achieved by the scanning process is the color or texture of the sampled points. This task is done by a conventional camera which is positioned near the emitter and takes a picture from each view. The picture will be mapped as texture onto those parts of the point cloud where the view matches.

In the acquisition process there is always an error in measurement which has to be taken into account in any of the following steps of the content creation pipeline.

Surface Reconstruction

Once a point cloud has been acquired, surface features have to be reconstructed if this information is not stored with the spatial information [TWG04].

The most basic surface feature is given by the k nearest neighbor graph. It defines the direct neighborhood of a point inside the point cloud. The framework suggestions by [PS04] can be used to implement a stream processing pipeline which allows to keep

Figure 1.3: Examples of scannable objects.

just a small portion of the whole point cloud in main memory. Stream processing of points divides the reconstruction process in two two stages.

- The partitioning stage stores all points in a database ready data structure. For example a binary space partition (BSP) tree.
- The scanning stage traverses all points sequentially in a specific order and finds the neighborhood for each point.

This enables to find strategies that keep only a fraction of the whole dataset in main memory while the rest is stored out-of-core. With this information on hand reconstruction methods like [MAS01] or [MN03] can be used.

There are many more ways to reconstruct a valid surface description. But those methods are dedicated to datasets that have to be processed in main memory. One method to gather a point cloud dataset is by using parametric surfaces that fit the shape of the model best. Those ways of calculating a point cloud assumes that there is a really good mathematical description of the model, which is often not given and hard to find.

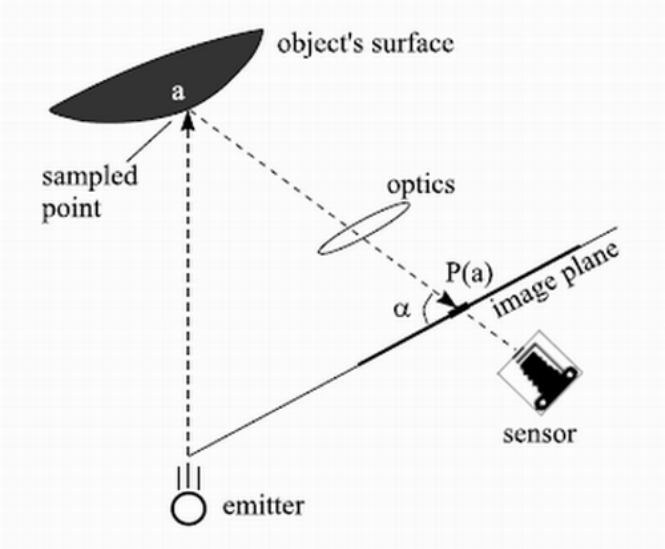

Figure 1.4: Emitter - Object - Sensor Triangulation.

There is also the possibility to approximate isosurfaces from implicit surface equations. One technique that leads to a iso-surface reconstruction is to use the marching cube algorithm [LC87]. This method needs to first voxelize the point cloud. Which means it needs to build a uniform grid around the point cloud. Then implicit surfaces can be found for each grid cell. Uniform grids are distribution depended and therefore this method is only usefull for point clouds with uniform sampling like it is given in medical data sets. The standard method for surface reconstruction in point clouds is moving least squares. The algorithm has two stages. First a local reference domain is computed. After that a local bivariate polynomial approximation to the surface in a local neighborhood has to be found [MAS03]. The reference domain can be computed by using weighted total linear least squares. The polymonial approximation of the surface can be found by computing the coefficients of a polynomial approximation so that the weighted least squares error is minimized. (see Figure 1.6)

Processing and Modeling

In this step the point cloud is altered by a user guided post processing step. User guided selection tools help the user to mark regions of interest and delete points [Sch09] or add content like textures [MZG02].

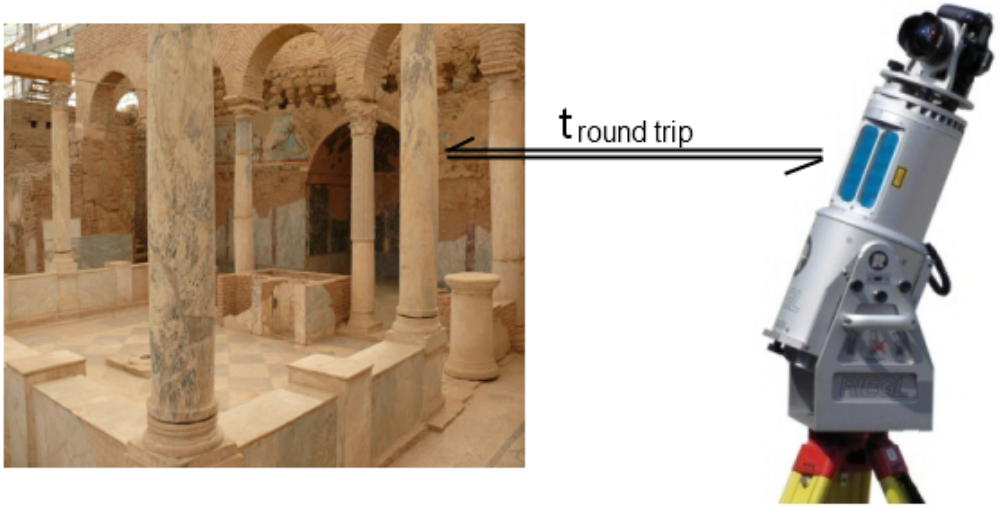

Figure 1.5: The image shows the round trip for one point of a time of flight scanner.

Rendering

The rendering step is a very well discussed topic in computer graphics. But there are just a few strategies that can be used for very large point clouds. Some of them are mentioned in the following list.

- QSplat

  [RL00] uses a bounding volume hierarchy to render very large meshes and point clouds. Therefore the bounding sphere for a model has to be computed. The bounding sphere is successivly reduced and the correspoding points are stored within the smaller bounding volume. This strategy leads to a binary tree where each node has a bounding sphere together with additional information of the child nodes like the average color of the points in the leaf nodes. Figure 1.7 shows the binary tree representation together with the memory layout of the nodes. This method allows to render only those points that are in the view frustum by testing the bounding volumes against the view frustum.

- Sequential Point Trees

  [CDS03] uses an octree to build a sequential linearized list of points. Those points are stored together with (r_{min}, r_{max}) values. These values are used to decide wether an interval in the list is drawn or not based on the visibility of the points

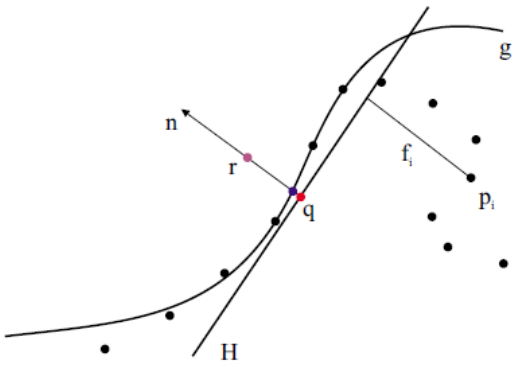

Figure 1.6: The image shows the reconstruction procedure of the moving least squares algorithm. Where H is the reference domain and g the polynomial approximation. First the reference domain H for the point r has to be found where the point q is the projection of the point r onto the reference domain H. Then a local polynomial approximation g to the heights f_i of points p_i over H is computed [MAS03].

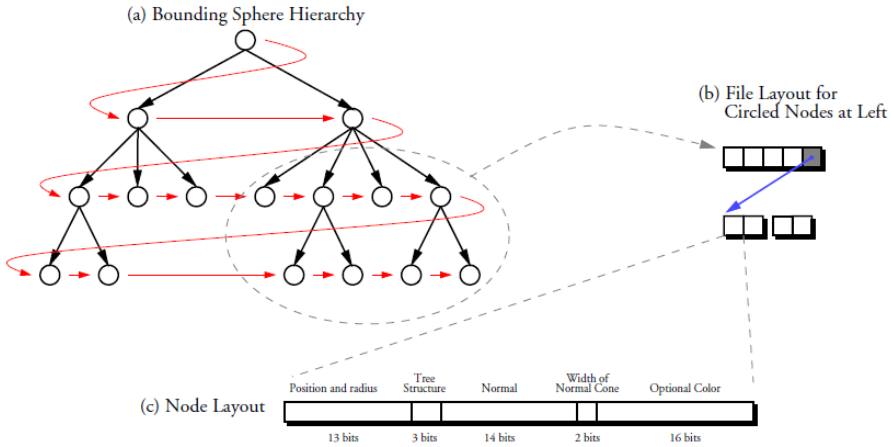

Figure 1.7: Breadth first traversal of the binary tree together with the tree node memory layout [RL00].

from a view point. This strategy allows to find the level of detail for a point cloud based on how much space the points will consume in screen space.

- Instant Points

  [WS09] concentrates on the performance of the rendering part by using a smart implementation of the sequential point tree data structure [CDS03]. The octree is

divided into inner and outer octrees. Both with a constant recusion depth. Each of the outer octree nodes manages an inner octree node. The points are managed by the inner octrees. The Figure 1.8 shows the octree representation for the outer and inner octree. The outer octree represenation takes care of the out-of-core management and the view frustum culling. The inner octree nodes are drawn whenever they are visible.

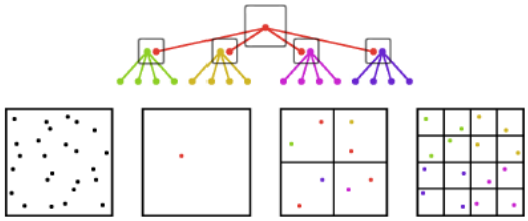

Figure 1.8: Nested octree representation [WS09].

This work was implemented in the Scanopy point rendering system [WS09] .

1.4 Contribution

In this diploma thesis an external sorting algorithm and an efficient neighborhood algorithm, which enables to reconstruct the surface normals, was developed. The external sorting algorithm is developed as tree based sorting algorithm. In order to keep only those points in memory that are necessary to distribute the points in the buckets of the leaf nodes, a least recently used (LRU) cache is used. This strategy allows to use the whole main memory as second level cache. Furthermore the file accesses are reduced, because the points are kept in the cache as long as the buckets are updated with new points. The neighborhood algorithm is implemented both memory and I/O efficient and was developed as an operator in the framework suggested by [PS04]. It improves the existing k nearest neighborhood operator by [PS04] in that it works with arbitrary sampling densities. The memory efficiency comes from the fact that only those points that contribute to the direct neighborhood are kept in main memory. The I/O efficiency comes from the fact that the read and write operations are handled sequentially on the point stream. The normal estimation operator is implemented following the linear least squares regression.

1.5 Overview

- Chapter 1 This chapter gives a brief introduction.

- Chapter 2 Previous work will show the techniques available for processing point clouds out-of-core.
- Chapter 3 External Sorting shows a way to sort point clouds externally.
- Chapter 4 Point Streaming framework introduces the point streaming framework and its local operators.
- Chapter 5 Gives a short overview into the implementation details.
- Chapter 6 Shows some results.
- Chapter 7 Summarizes all our efforts and concludes this diploma thesis.

CHAPTER 2

Previous Work

2.1 Out-Of-Core Data Structures

Most of the previous work is dedicated to in-core processing of point clouds. Out-of-core algorithms for reconstructing point clouds are an active research topic. Estimating the normals from point clouds is a well discussed topic in computer science [MAS01], [HHS92]. But those works show their results on data structures that are completely processed in main memory. Nevertheless the results of those works also apply to out-of-core algorithms. For example the work by [PS04] relies on least squares regression to compute the normals.

The most basic concern in out-of-core methods is how to find the direct neighborhood that allows to reconstruct surface features. There are several approaches that can handle this task out-of-core. Out-of-core approaches strongly depend on the locality of the underlying data and can not take global information into account. In order to find the direct neighborhood for points residing currently in memory, any database ready data structure can be used that supports range queries and stores the data distribution independent.

The approach by [MBH07] builds an out-of-core octree and solves the poisson equation per octree level by using an adaptive marching cube algorithm. Therefore the partitioned data is streamed along the x axis by traversing the octree at each level as can be seen in Figure 2.1.

The work by [JSV07] is a database approach. The point cloud is partitioned in a disk based quad-tree. The key values are the distance values between the different quadtree cells. A range query simply searches for cells with distance values inside a spherical range.

The r-tree data structure [Gut84] enables the usage of very efficient range queries. It is the preferred database data structure for geometrically based datasets because of its

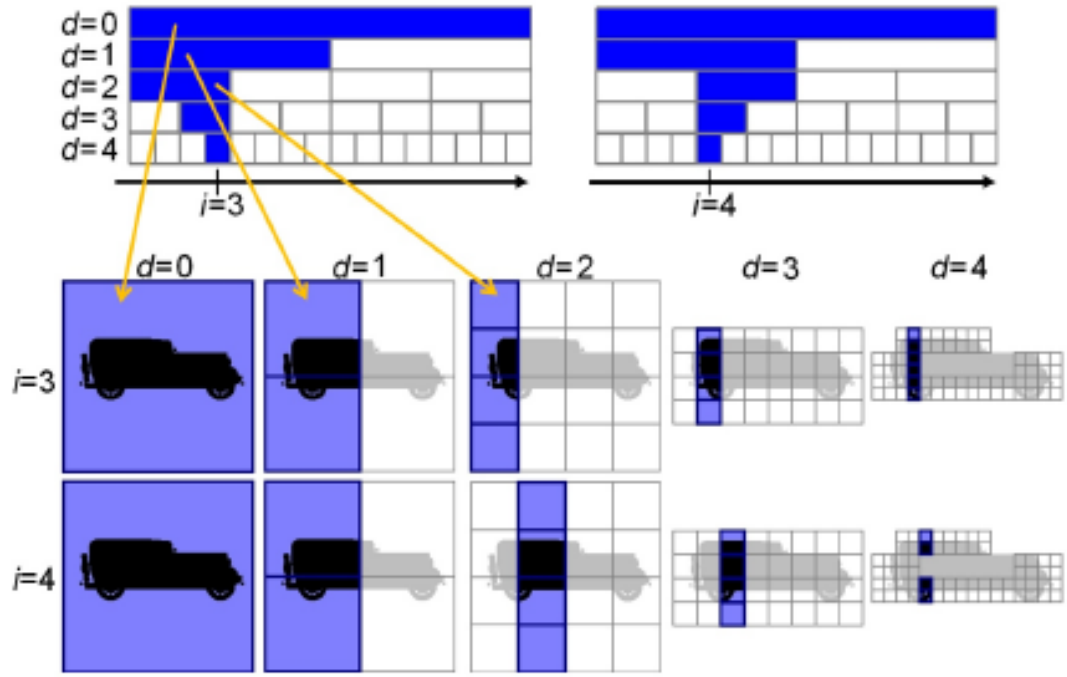

Figure 2.1: Streaming along the x axis of the octree level [MBH07]. The blue octree nodes are stored in main memory.

highly efficient space partitioning. Range queries can also be handled very efficiently. The main difference to disk based approaches like [JSV07] is that points are partitioned in rectangular regions.

The kd-tree based approach is proposed in several papers [Moo91] [Ben75] [Mou10]. The main reason for the wide usage of this data structure is that the range queries for finding the k nearest neighborhood in a point cloud is very efficient. The main disadvantage is that it has to be built in-core. A kd-trie like [Sam06] achieves similar results to the kd-tree approach. The lsd-tree [AHW89] is a variation to the kd-trie that enables to build a semi-balanced kd-trie by using the locality of the data only. It also has the ability to be used as out-of-core data structure because all points are stored in files at the leaf nodes of the tree. Figure 2.2 shows how the lsd tree manages the nodes where internal nodes are all nodes in main memory and external nodes are all nodes that are kept persistent. All inner nodes are directory nodes and the leaf nodes store the buckets with the point information.

Another efficient method that can be used to solve the problem of finding the k nearest neighbor graph is called grid file [JNS84]. This data structure partitions the points in a grid and stores the points in a file. One file can store the points of more

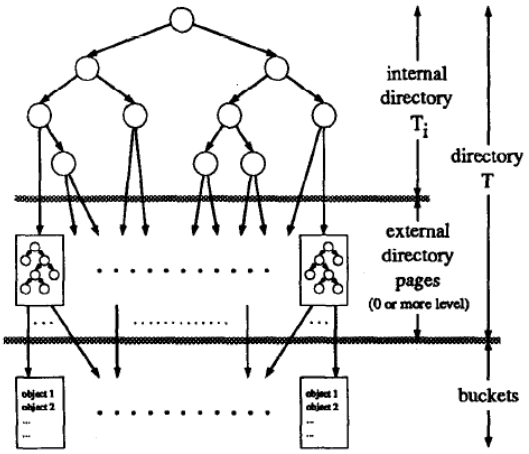

Figure 2.2: The different level of a lsd tree [AHW89].

than one grid cell. This smart usage of files allows to keep the disk with a very low fragmentation overhead. The range query to find the nearest neighborhood for a point can be achieved in constant algorithmic complexity. Nevertheless the direct access to a grid cell comes with the cost of a file access. Figure 2.3 shows how the grid file data structure partitions the three dimensional space.

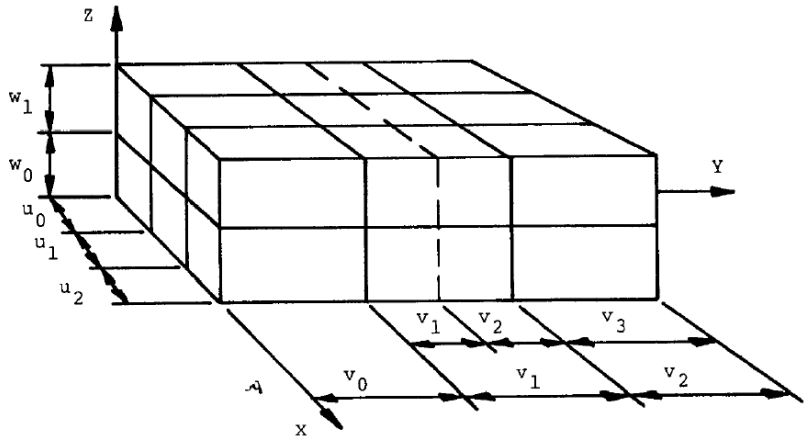

Figure 2.3: The image shows how the grid file partitions the three dimensional space [JNS84].

The work by [MC08] partitions the point cloud by sorting the points with respect to their Morton key. All points that are sorted with respect to their Morton order are also

partitioned in a quad tree (see Figure 2.4).

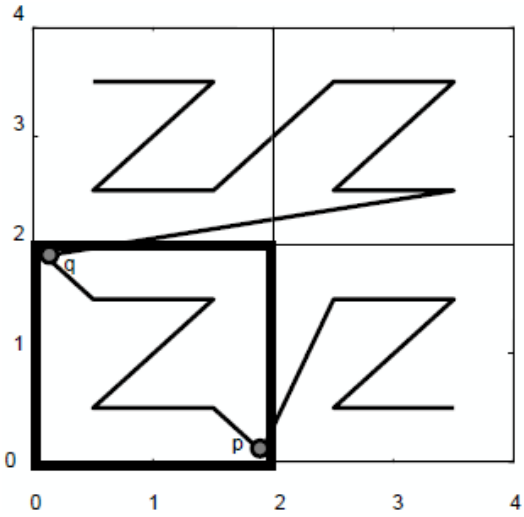

Figure 2.4: The smallest quadtree box containing two points will also contain all points lying between the two in Morton order [MC08].

The k nearest neighborhood for any point in Morton order can be found by a simple binary search. First the first k next points in morton order are take as approximation of the radius in which the nearest neighbors are situated. Then a binary search over the whole point cloud searches if points are inside the nearest neighborhood radius. Furthermore the binary search refines the nearest neighborhood radius whenever a point is inside the searching range. This strategy is an in-core approach and therefore only useful if all points can be stored in main memory.

2.2 Normal Estimation

Estimating normals from a point cloud is a well discussed topic in computer graphics. This diploma thesis uses the method of linear least squares regression analysis. The main goal in least squares regression analysis is to fit an observation X to a model function $f(X)$.

The observation $X = \{x_1, x_2...x_m\}$ consists of m n-dimensional vectors. The model function $f(X) = Y = \{Y_1...Y_m\}$ in the case of linear least squares regression is a hyper plane of the form $\sum_{i=1}^{m} \beta_i * x_i = Y_i$, where β_i are the coefficients of the hyper plane, x_i are the vector coordinate values from the model data and Y_i are the values from the model function evaluation $f(X_i)$. The main goal is to find the coefficients that minimize the squared residual errors $\min_X (Y - \beta * X)^2$. In the following sections two kinds of linear least squares fitting and their numerical solution will be discussed [Ebe09] [WHP02].

Ordinary Linear Least Squares Fitting

Ordinary linear least squares fits the model data X to a planar model function $f(x, y) = A * x + B * y + C = z$, with (A, B, C) as the plane coefficience. This means the squared residual errors will be minimized orthogonal to the z axis. This will lead to the following minimization problem $\min_X (A * x + B * y + C - z)^2$. Furthermore the first partial derivative of the system of linear equations from the minimization problem $\sum_{i=1}^{n} (A * x_i + B * y_i + C - z_i)^2 = (0, 0, 0)$ leads to a system of linear equations, $2 * \sum_{i=1}^{n} (A * x_i + B * y_i + C - z_i) = \nabla f(x, y) = (0, 0, 0)$ which are called normal equations of the least squares problem.

$$\begin{bmatrix} \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i * y_i & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i * y_i & \sum_{i=1}^{n} y_i^2 & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} 1 \end{bmatrix} * \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} x_i * z_i \\ \sum_{i=1}^{n} y_i * z_i \\ \sum_{i=1}^{n} z_i \end{bmatrix} \tag{2.1}$$

This linear system of equations can easily be solved by the inverse of the matrix $A * x = b => x = b * A^{-1}$. The inverse of an overdetermined system of linear equations can be solved by Cramer's rule. The normal vector can be easily calcluated from the plane coefficients.

Orthogonal Linear Least Squares Fitting

If the squared residual errors have to be measured orthogonal to the model function then the following normal equations can be used to solve the problem. The error function $E(A, N)$ for the least squares minimization is given by the following equation.

$$E(A,N) = \sum_{i=1}^{m}(N \cdot Y_i)^2 = N^T(\sum_{i=1}^{m} Y_i Y_i^T)N = N^T M(A) N \tag{2.2}$$

Where $Y_i = X_i - A$, X_i are the sample points and A is a point on the hyper plane. N is a unit vector on the hyper plane. $N^T M(A) N$ is a quadratic form whose minimum is the smallest eigenvalue of $M(A)$. The normal equations for a three dimensional vector space are given by $M(A) = (X_i - A) * (X_i - A)^T =$ with $A = (a, b, c) = 1/m * \sum_{i=1}^{n} X_i$.

$$M(A) = \begin{bmatrix} \sum_{i=1}^{n}(x_i - a)^2 & \sum_{i=1}^{n}(x_i - a) * (y_i - b) & \sum_{i=1}^{n}(x_i - a) * (z_i - c) \\ \sum_{i=1}^{n}(x_i - a) * (y_i - b) & \sum_{i=1}^{n}(y_i - b)^2 & \sum_{i=1}^{n}(y_i - b) * (z_i - c) \\ \sum_{i=1}^{n}(x_i - a) * (z_i - c) & \sum_{i=1}^{n}(y_i - a) * (z_i - c) & \sum_{i=1}^{n}(z_i - c)^2 \end{bmatrix} \tag{2.3}$$

These normal equations can be solved by any eigensystem solver. The normal vector is given by the eigenvector with the smallest eigenvalue. In fact the matrix is symmetric and positive semi-definite which means that all values are greater or equal zero and the upper matrix has the same values than the lower matrix. This leads to a number of possible algorithms that can solve the eigenvalue problem efficiently. An often used algorithm is the QR algorithm [WHP02]. The algorithm can solve the problem in $O(n^2)$ complexity. Therefore it is used in this diploma thesis to solve the eigenvalue problem and furthermore to find the normal of the least squares plane.

2.3 External Sorting

Problem Field

The challenge of sorting n-elements externally is not mainly given by the overall algorithmic complexity, but by the number of I/O disk accesses. Whenever the size of a given dataset is larger than the main memory, only a portion can be kept in main memory and the rest has to be stored externally. This external memory storage can be one or more hard disks. Disk I/O is the slowest possible access in any system architecture. Keeping disk I/O access as small as possible is therefore the main concern. There are many techniques to keep the I/O disk access as small as possible. The simplest and most effective is to read and write pages. The size of one page is given by the system architecture. Another way to reduce the number of I/O accesses is to use asynchronous file access techniques. The following two sections will give a brief overview of the two most important strategies to solve the problem of sorting a very large dataset out-of-core in $O(n * log(n))$ overall complexity.

Merge Sort

Merge sort follows a divide and conquer strategy. First it divides the dataset into k blocks and sorts all the blocks. Secondly it merges any two sorted blocks to one larger block until there is only one block left.

In-place Merge Sort

The simplest implementation of a merge sort algorithm is the in-place merge sort. The Algorithm 1 handles the sorting procedure in two steps. First it recursivly divides the elements in two sub arrays until there is only one element left. Then it merges the sub arrays into element arrays twice as large until only one array is left.

Algorithm 1 Inplace Merge Sort

Require: A valid point stream.
Ensure: A valid sorted point stream.

1: Recursivly call Inplace Merge Sort with the left half until only one element is left.
2: Recursivly call Inplace Merge Sort with the right half until only one element is left.
3: Merge the left and right half of the point stream sorted into a temporary array (see Algorithm 2).
4: Copy the temporary array into the point stream.

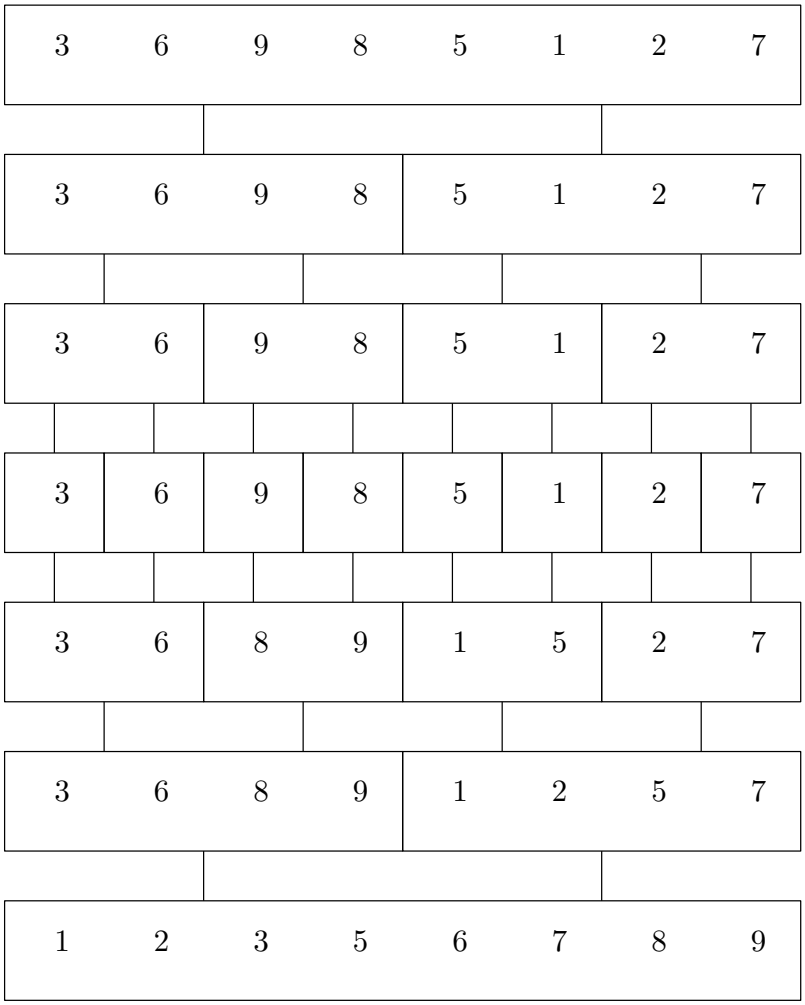

Figure 2.5: Inplace Merge Sort

External Merge Sort

External merge sort is the generalization of the in-place merge sort. The most basic implementation of external merge sort is called two way merge sort (see Algorithm 3). At the first pass N pages are read into main memory and made persistent for example by using memory mapped files. One page consists of a fixed number of elements. Each of the N pages have to be sorted according to their key values. The next pass takes two pages and merges them into a two times bigger page. The merge step has to be repeated until $N/2$ passes have been executed.

A more generalized formulation of the two way merge sort algorithm is called k way merge sort (see Algorithm 4). The main difference to the two way merge sort is that in

Algorithm 2 Merge

Require: Two arrays with at least one element.
Ensure: One merged array.

```
1:  while The first and second array is not empty. do
2:    if The first element of the first array is smaller than the first element of the second
      array. then
3:      Insert the first element from the first array into the merged array.
4:      Remove the first element from the first array.
5:    else
6:      Insert the first element from the second array into the merged array.
7:      Remove the first element from the second array.
8:    end if
9:  end while
10: while The first array is not empty. do
11:   Insert the first element from the first array into the merged array.
12:   Remove the first element from the first array.
13: end while
14: while The second array is not empty. do
15:   Insert the first element from the second array into the merged array.
16:   Remove the first element from the second array.
17: end while
```

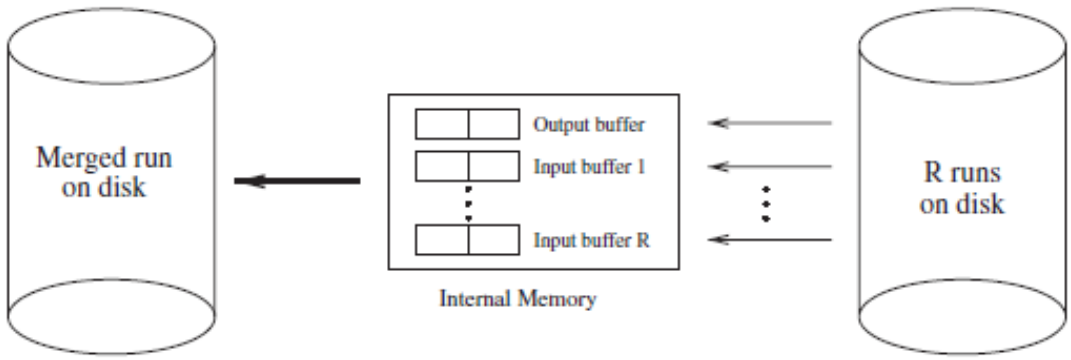

Figure 2.6: Merge Sort.

each pass B pages are read into main memory. B stands for the number of buffers that are used to merge the pages. As a consequence the merge passes are reduced to N/B.

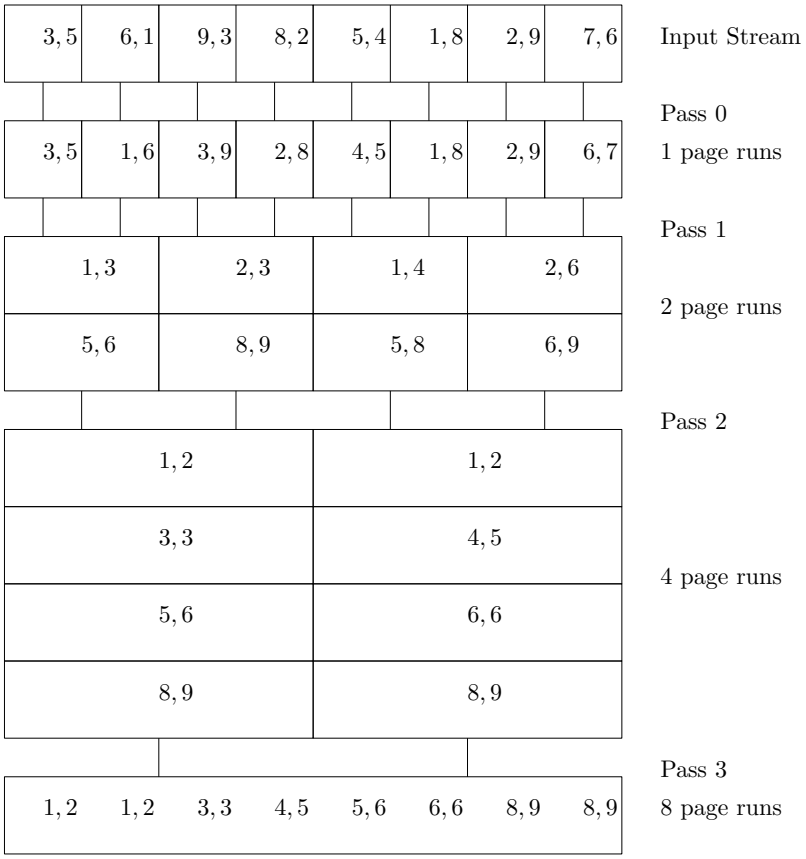

Figure 2.7: Two way merge sort with a page size of two elements.

Algorithm 3 Two-Way Merge Sort

Require: A valid point stream.
Ensure: A valid sorted point stream.

1: Divide all elements into N pages and sort each page.
2: **while** Merge pass counter $< N/2$ **do**
3: **while** There are pages left. **do**
4: Read two pages into main memory.
5: Merge the two pages sorted into a temporary array (see Algorithm 2).
6: Copy the temporary array into the point stream.
7: **end while**
8: **end while**

Distribution Sort

Distribution sort is another very effective method of sorting externally. The main idea is to distribute all elements into k buckets and sort the buckets in main memory as can

Algorithm 4 k-Way Merge Sort

Require: A valid point stream.
Ensure: A valid sorted point stream.

```
1: Divide all elements into N pages and sort each page.
2: while Merge pass counter < N/B do
3:    while There are pages left. do
4:       Read B pages into main memory.
5:       Merge the B pages sorted into a temporary array (see Algorithm 2).
6:       Copy the temporary array into the point stream.
7:    end while
8: end while
```

be seen in Algorithm 5.

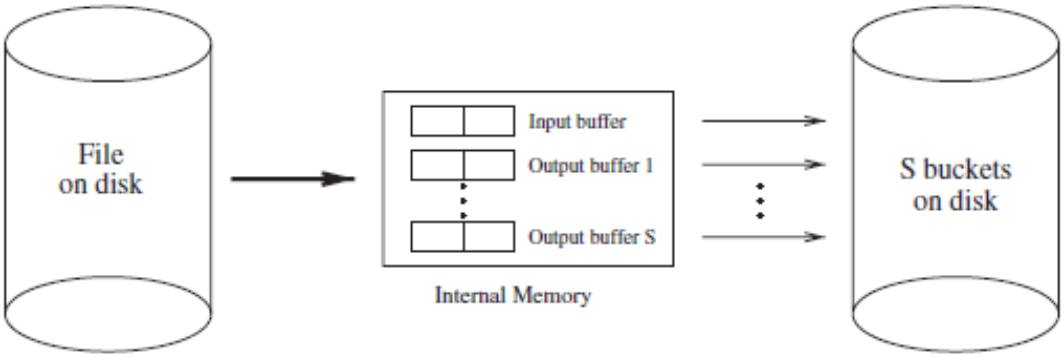

Figure 2.8: Distribution Sort.

There are many in-place variations of the bucket sort. For example radix sort [Sed92] and counting sort [Knu98] are variations of the bucket sort. One of the major drawbacks of the bucket sort is that the distribution of the points into the buckets can not be parallelized. The sorting part of the strategy can be parallelized. The distribution part in combination with a LRU cache is very efficient.

Algorithm 5 Bucket Sort

Require: A valid point stream.
Ensure: A valid sorted point stream.

```
1:  while There is a point in the point stream. do
2:      Find the bucket for the active point.
3:      if There is no bucket in the bucket list. then
4:          Create a new bucket.
5:          Insert the bucket into the bucket list.
6:      end if
7:      Insert the active point into the found bucket.
8:  end while
9:  while There is a bucket in the bucket list. do
10:     Sort the elements in the bucket.
11:     Copy the bucket into the point stream.
12: end while
```

2.4 K Nearest Neighbor Graph

Problem Field

Finding the k nearest neighborhood graph is a challenging problem. The graph itself is a directed asymmetric minimum spanning forest. Which means any vertex in the tree has k nearest neighbors which are linked asymmetrically. The edges of the nearest neighborhood graph are given by the Euclidean distance between the vertices. In order to build such a graph the Euclidean distance values between the vertices have to be calculated first. Only those vertices with the minimum Euclidean distance are the nearest neighbors to a point. This task is non trivial because finding the edges in a graph is potentially of $O(n^2 * log(n))$ complexity. The problem of finding the k nearest neighbor graph is equal to finding the nearest neighbors for any point in a point cloud. This is because the edges in the k nearest neighborhood of a point are the same edges for that point in the k nearest neighborhood graph. Therefore this chapter shows how the k nearest neighborhood graph is constructed. The following introduction into the k nearest neighbor graph shows how such a graph can be constructed in $O(n * log(n))$ complexity with minimum space requirement.

Definitions

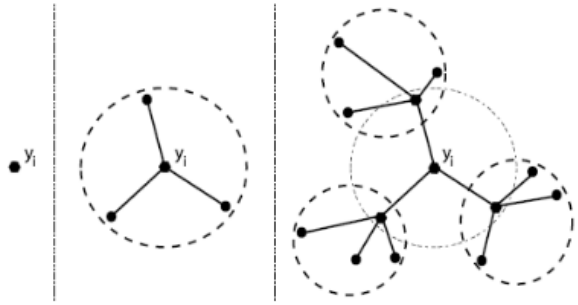

Figure 2.9: kNN - graph with four neighbors.

- The k nearest neighbor graph $KNNG(p_i), i = 1 \ldots n$ is a directed (asymmetric) minimum spanning forest for a given set of points $P = \{p_1, p_2, ..., p_n\} \in R^d$.
- The k nearest neighbors of any point p_i in the set P are all points $p_j, i \neq j$, with minimum Euclidean distance from p_i.
- k stands for the number of minimum weighted edges connecting the vertex p_i with the nearest neighbors $p_j, j = 1 \ldots n, i \neq j$.

- The weight is given by the Euclidean distance between two vertices $distance(p_i, p_j) = \sum(p_j - p_i)^2, i \neq j$.

Construction

The construction of a k nearest neighbor graph is a non trivial task. Donald Knuth [Knu98] called the problem of finding the k nearest neighbors post-office problem. The main goal is to find the nearest post-office locations for any given post-office. Typically the construction process has to overcome two stages.

- The partitioning stage stores all points in a database ready data structure. For example a binary space partition (BSP) tree.

- The scanning stage traverses all points sequentially in a specific order and computes surface features like the normal vector.

Brute Force

The greedy way to solve this problem is shown in the following Algorithm 6. It divides the problem in two subproblems. The first step calculates the Euclidean distances between all points. The second step finds the k nearest neighbors by sorting the Euclidean distance values in ascending order and store the first k elements. The algorithm has $O(n^2 * log(n))$ complexity and needs $O(n^2)$ space.

Algorithm 6 Brute Force

Require: A valid point cloud.
Ensure: Any point has k nearest neighbors.

1: **for all** $p_i \in P, i = 1 \dots n$ **do**
2: **for all** $p_j \in P, j = 1 \dots n$ **do**
3: Compute the Euclidean distance $d_i(p_i, p_j), i \neq j$.
4: **end for**
5: Sort the Euclidean distance values in increasing order.
6: Store the first k points of the sorted Euclidean distance values.
7: **end for**

Brute Force With Heap

The sorting step can be avoided by using a maximum heap that stores the k nearest neighbors immediatly and rejects all points with Euclidean distance greater than the already found maximum Euclidean distance value (see Algorithm 7). The maximum heap is a binary tree with the property that all elements at any level of the binary tree have to be greater than the elements at all lower binary tree level. The heap enhancement of the brute force algorithm leads to a complexity of $O(n^2 * log(n))$ and needs $O(n * k)$ additional space.

Algorithm 7 Brute Force With Heap

Require: A valid point cloud.
Ensure: Any point has k nearest neighbors.

1: **for all** $p_i \in P, i = 1 \dots n$ **do**
2: **for all** $p_j \in P, j = 1 \dots n$ **do**
3: Update the maximum heap with p_j (see Algorithm 8).
4: **end for**
5: **end for**

Algorithm 8 Heap Update

Require: Two valid points p_i and p_j, $i \neq j$. The point p_i is the center point and p_j is the new point to be updated.
Ensure: The k nearest neighbor points for the point p_i are in heap order.

1: **if** The heap is full **then**
2: **if** $distance(p_i, p_j) \leq distance(p_i, p_{max}), i \neq j$ **then**
3: Remove the maximum value of the maximum heap (see Algorithm 9)
4: Insert p_j in the maximum heap (see Algorithm 10).
5: **end if**
6: **else**
7: Insert p_j in the maximum heap (see Algorithm 10).
8: **end if**

The heap data structure is very powerful when it comes to keep values in a specific order. One major advantage of the heap data structure is that it can be implemented as an implicit data structure (see Figure 2.10). This allows to use any container with random access to its elements. Because of that the access to parent and child elements is accomplished in constant complexity $O(1)$.

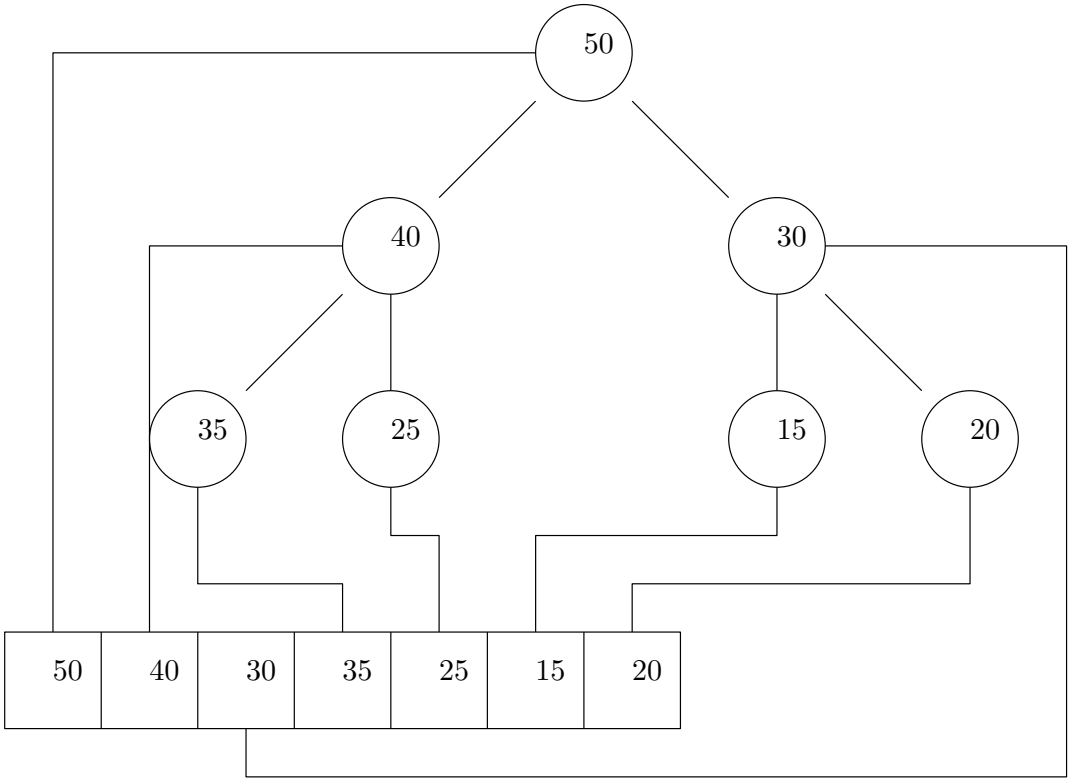

Figure 2.10: A heap and its corresponding implicit representation as array.

The following two Algorithms 9 and 10 will ensure that the afore mentioned binary heap property is intact at all time. The heap data structure is always balanced and therefore the following heap remove and heap insert algorithms are accomplished with $O(log(n))$ complexity.

The heap remove Algorithm 9 ensures that the elements are in heap order after the first element is removed. The first element will be replaced by the right most leaf node. Afterwards the new root element will test if the right or leaft child node is greater than the root node. If the right or left child node is greater than the parent node then the corresponding child node will be swapped with the parent node. This procedure will repeat until there is no more child node or if the parent node is greater than the child nodes (see Figure 2.11).

Algorithm 9 Heap Remove

Require: A valid heap.
Ensure: First heap element is removed.

1: Remove the first heap element.
2: Replace the first element with last element in heap order, which is the right most leaf node.
3: Start with first heap element.
4: **while** The element has a child. **do**
5: **if** The left and right child element is greater than the element. **then**
6: Swap the element with its child element.
7: Find the new child element.
8: **else**
9: The heap is in order (return).
10: **end if**
11: **end for**

The heap insert Algorithm 10 ensures that the elements are in heap order after the element is inserted at the left most leaf node position. After the element is inserted it will be tested against the parent node. If the parent node is less than the child node then the two nodes will swap their position. This procedure will be continued until the parent node is greater then the child node or the root node is reached (see Figure 2.12).

Algorithm 10 Heap Insert

Require: A valid element.
Ensure: The element is inserted in heap order.

1: Insert element at last leaf element.
2: **while** The element has a parent. **do**
3: **if** The element is greater than the parent element. **then**
4: Swap the element with its parent element.
5: Find the new parent element.
6: **else**
7: The heap is in order (return).
8: **end if**
9: **end for**

Uniform Grid

A uniform grid is the simplest data structure that stores the values in an n-dimensional array. It enables the fastest way of nearest neighbor searching possible (see Algorithm 11). The direct neighborhood of a grid cell can be found in constant complexity $O(1)$ by a simple lookup in the n-dimensional array (see Figure 2.13). The search for the neighborhood inside the grid cell has $O(M)$ complexity where M is the number of points in the grid cell. Each point stores a maximum heap that manages the k nearest neighborhood for that point.

Algorithm 11 Uniform Grid Search For Nearest Neighborhood

Require: A valid point cloud.
Ensure: Any point has k nearest neighbors.

for all $p_i \in P, i = 1 \dots n$ **do**
  Search in the grid cell where the point is situated (see Algorithm 12).
  if The point p_i does not have k nearest neighbors. **then**
    Increase the search range from the grid cell where the point is situated in each direction by one.
    while The grid borders have not reached and the point p_i does not have k nearest neighbors. **do**
      Search in all grid cells up to the grid cell search range (see Algorithm 12).
      Increase the search range by one cell in each direction.
    end while
  end if
end for

Algorithm 12 Uniform Grid Cell Search For Nearest Neighborhood

Require: A valid point p_i.
Ensure: All points in the cell are tested against the given point.

for all Points p_j in the grid cell. **do**
  if There are more than k elements in the maximum heap. **then**
    Remove the maximum point from the heap (see Algorithm 9).
  end if
  Insert the current point p_j into the maximum heap (see Algorithm 10).
end for

N Dimensional Range Searching

There are many data structures that allow to setup range queries in a n-dimensional space. In fact any binary space partitioning (BSP) tree data structure can be used. One of the most important data structures when it comes to searching for the nearest neighborhood of a point is the kd-tree [Ben75] [Sam06]. The kd-tree is a binary space partitioning tree which allows to partition a point cloud according to its distribution in the n-dimensional space where it is situated in (see Figure 2.14). The insertion method (see Algorithm 13) is the same as in any binary space partition tree. The space is subdivided into hyper planes and the points are located either left or right from the hyper plane. The complexity of the insertion method is logarithmic $O(log(n))$ if the kd-tree is balanced.

Algorithm 13 Insert Into KD-Tree

Require: A valid point.
Ensure: The point is inserted into the kd-tree.

```
1: Start with tree node as root node.
2: while The tree node is not a leaf node. do
3:    if The point is left from the tree node. then
4:       Set tree node as left node.
5:    else
6:       if The point is right from the tree node. then
7:          Set tree node as right node.
8:       end if
9:    end if
10: end while
11: Insert point p_i into the kd-tree at the found tree node.
```

Another way to insert elements in a kd-tree is to completely build the kd-tree instead of sequentially inserting elements into the kd-tree. In fact the build Algorithm 14 is the prefered strategy because it guarantees a balanced binary tree which is important for any search strategy. The build kd-tree Algorithm 14 has an overall complexity of $O(n * log(n))$ if the method for finding the splitting node is to find the median at the current splitting axis. The median can be found with linear complexity $O(n)$, if a selection algorithm (e.g. STL nth_element) is used.

Algorithm 14 Build KD-Tree

Require: A valid point cloud.
Ensure: A valid kd-tree.

1: Find the splitting node at the current axis.
2: Set the splitting value for the median node.
3: Build KD-Tree using the left node of the found median node.
4: Build KD-Tree using the right node of the found median node.

The Algorithm 15 shows how the goal of searching for the k nearest neighborhood can be achieved by using a kd-tree.

Algorithm 15 KD-Tree Search For Nearest Neighborhood

Require: A valid point cloud.
Ensure: Any point has k nearest neighbors.

1: Build a valid kd-tree. (see Algorithm 14)
2: **for all** $p_i \in P, i = 1 \dots n$ **do**
3: Search KD-Tree for point p_i starting from the kd-tree root. (see Algorithm 16 or 17)
4: **end for**

The search kd-tree Algorithm 16 inspects only those subtrees that are inside the spherical range query. In order to keep the spherical range distance as small as possible the algorithm starts with a value that is set to infinity. As soon as k elements are stored in the heap, the spherical range distance will be set to the maximum distance the heap currently stores. The spherical range distance is given by the distance between the point p_i and the maximum neighborhood point p_{max}. This simple but effective strategy allows to traverse only those values that are inside the current maximum distance.

Algorithm 16 KD-Tree Search

Require: A valid kd tree and a valid point p_i. Start with point p_j as root node.
Ensure: k nearest neighbors.

if The point p_j is on the right side from the point p_i. **then**
  Call search kd-tree with left child from p_j.
  if The point p_j is inside the given spherical range. **then**
    Call search kd-tree with the right child from p_j.
  end if
else
  Call search kd-tree with the right child from p_j.
  if The point p_j inside the given spherical range. **then**
    Call search kd-tree with left child from p_j.
  end if
end if
if The current point p_j is inside the spherical range. **then**
  if There are more than k elements in the maximum heap. **then**
    Remove the maximum point from the heap (see Algorithm 9).
    Insert the current point p_j into the maximum heap (see Algorithm 10).
    Set the search radius to the maximum distance between the point p_i and the maximum point in the heap.
  else
    Insert the current point p_j into the maximum heap (see Algorithm 10).
  end if
end if

Another effective way to search for the k nearest neighborhood is shown in the work by [Mou10]. This search strategy concentrates on finding the nearst neighbor and than give an estimate of the range radius from the distance of the point to the nearest neighbor (see Algorithm 17). Therefore the following inequality will be used $distance(p, q_i) \leq k * (1 + \epsilon) * distance(p, q_j), i \neq j, \epsilon \geq 0$. This means that the range radius for the k nearest neighborhood will be k times one plus epsilon times the distance from the point p to the nearest neighbor q_j. The value ϵ has to be greater or equal zero. Any point

q_i whose distance to the point p is less or equal the estimated radius will be a nearest neighbor. The main disadvantage in this solution is that the estimation of the range radius is distribution dependent.

Algorithm 17 KD-Tree Approximated Nearest Neighbor Search

Require: A valid kd tree and a valid point p_i. Start with point p_j as root node.
Ensure: k nearest neighbors.

if The point p_j is on the right side from the point p_i. **then**
  Call search kd-tree with left child from p_j.
  if The point p_j is inside the given spherical range. **then**
    Call search kd-tree with the right child from p_j.
  end if
else
  Call search kd-tree with the right child from p_j.
  if The point p_j inside the given spherical range. **then**
    Call search kd-tree with left child from p_j.
  end if
end if
if The current point p_j is inside the spherical range. **then**
  Update the heap from the point p_i with the new point p_j (see Algorithm 8).
  Compute the new range radius $r_{new} = k * (1 + \epsilon) * distance(p_i, p_j)$.
  if $r_{new} < rangeradius$ **then**
    Set the range radius to r_{new}.
  end if
end if

One Dimensional Range Searching

Another efficient way of dealing with the problem of finding the k-nearest neighborhood is to use one dimensional hash keys. In order to get one dimensional hash key values from n-dimensional coordinates the coordinates have to be mapped to unique one dimensional values. This can be achieved by building a binary space partitioning tree like a quad tree or by bit interlacing the corrdinate values.

The unique key in binary space partitioning trees is given by the binary decision paths from the root to the node where the point is stored (see Figure 2.15). The code given by the binary paths decision codes is often called morton code or z-order. The morton order can be used as a unique hash key.

The unique key can also be achieved by bit interlacing the coordinate values. For example by storing the keys in a two times bigger value. The odd bits in the key stores the bits from the x coordinate bits and the even bits store the y coordinate bits.

In the work by Timothy M. Chan [Cha06] the partitioning stage is reduced to sorting the points by a locality preserving hash code between two points. The comparison operator used in the sorting and binary search procedure uses another way of calculating the unique hash code. This approach computes the Morton code by XORing the coordinate values between two points. After that it finds the axis with the minimum of all XORed values. This found axis gives the position value where the coordinates will be compared.

While this approach works only for integer values, the work by [MC08] shows a way how to implement the strategy parallelized for floating point values.

The interesting part in the binary search is that the first k elements before and after any point in the sorted stream give an estimation of the radius in which all nearest neighbors exist. This estimated radius is refined by a simple binary search on all Morton ordered elements.

Algorithm 18 Parallel Construction of k-Nearest Neighbor Graphs

Require: A valid point cloud.
Ensure: Any point has k nearest neighbors.

1: Sort in morton order.
2: **for all** $p_i \in P, i = 1 \dots n$ **do**
3: Binary search k nearest neighborhood for point p_i (estimate and refine radius).
4: Update the maximum heap of the point p_i with the new elements (see Algorithm 8).
5: **end for**

2.5 Summary

The algorithm 8 is the most efficient way to store the nearest neighborhood. The algorithm 16 provides the prefered searching strategies for point clouds with arbitrary sampling density. The algorithms 17 and 11 can be used in order to find the nearest neighborhood in uniform sampled point clouds.

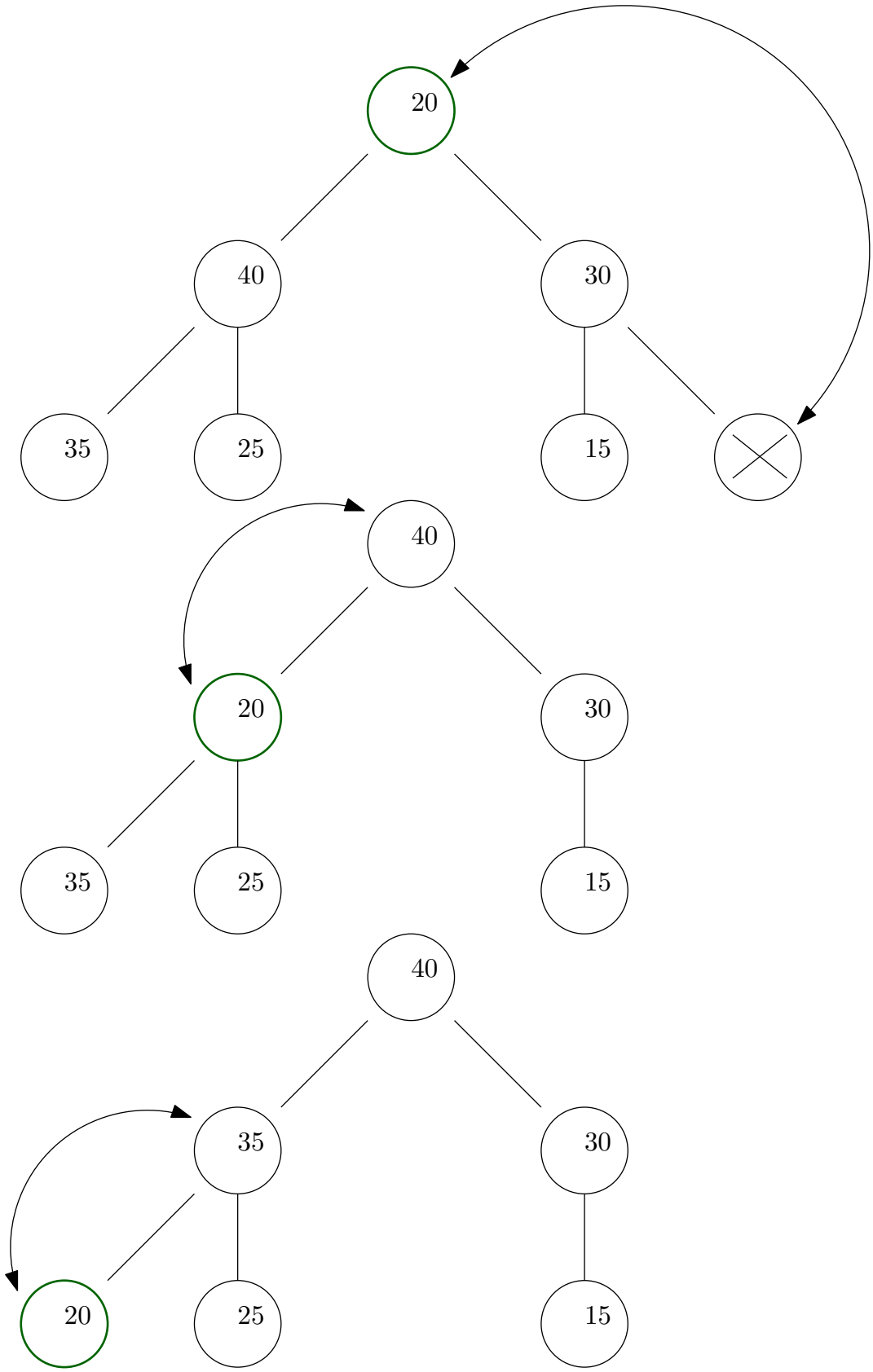

Figure 2.11: Heap Remove

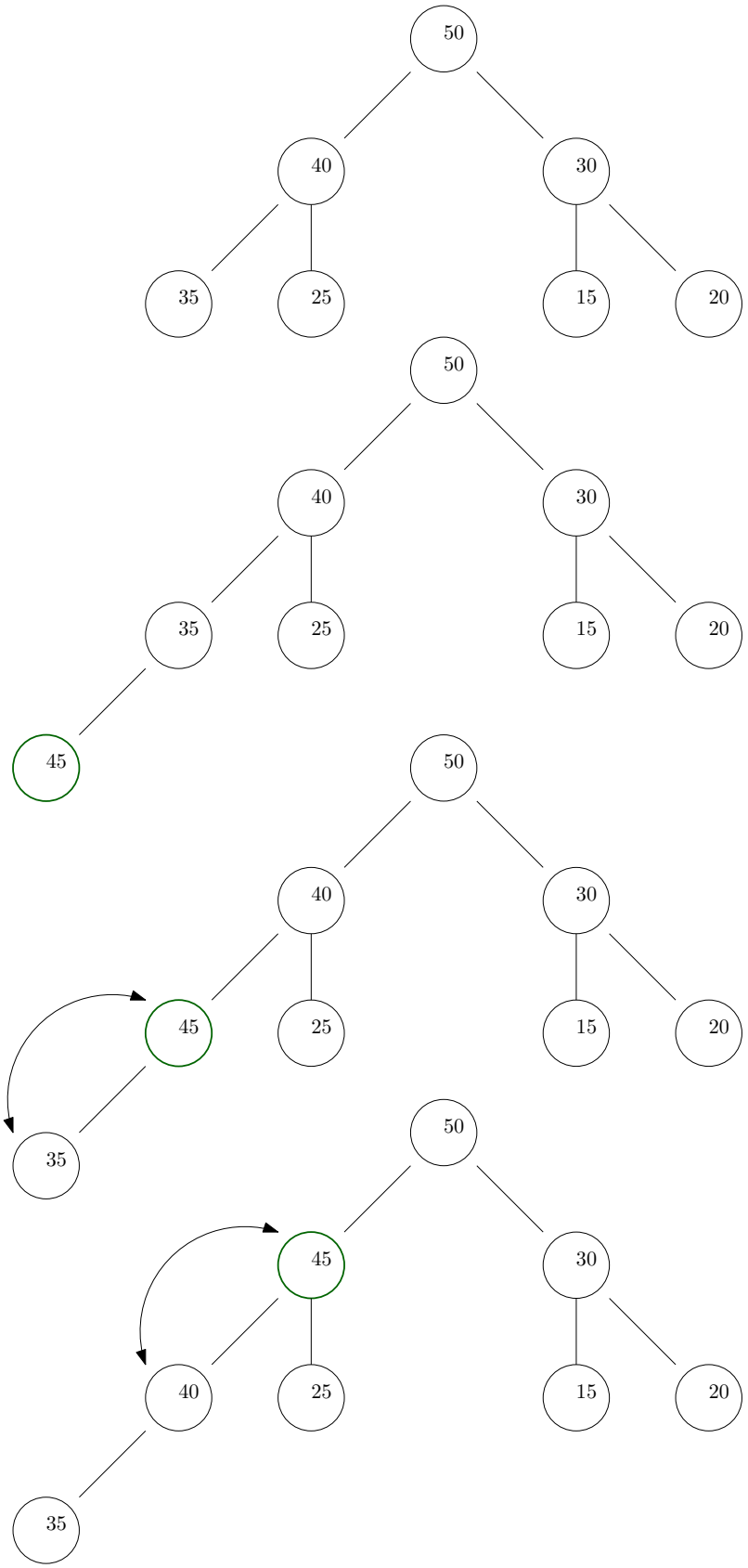

Figure 2.12: Heap Insert

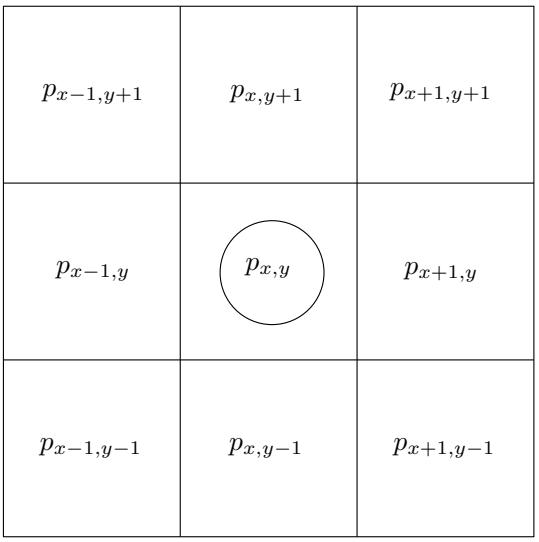

Figure 2.13: Grid neighborhood for the point $p_{x,y}$.

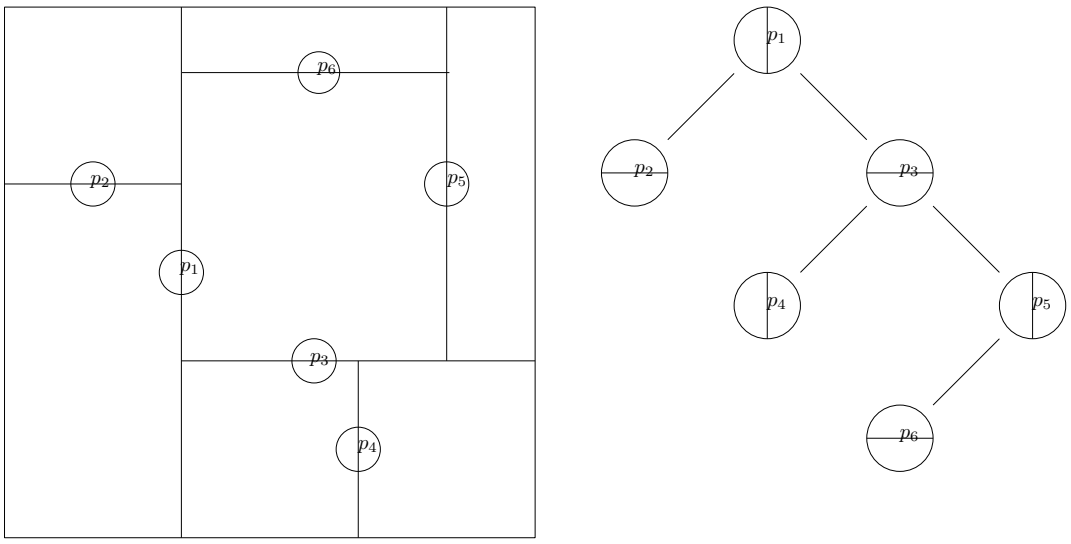

Figure 2.14: Space partitioning from kd-tree.

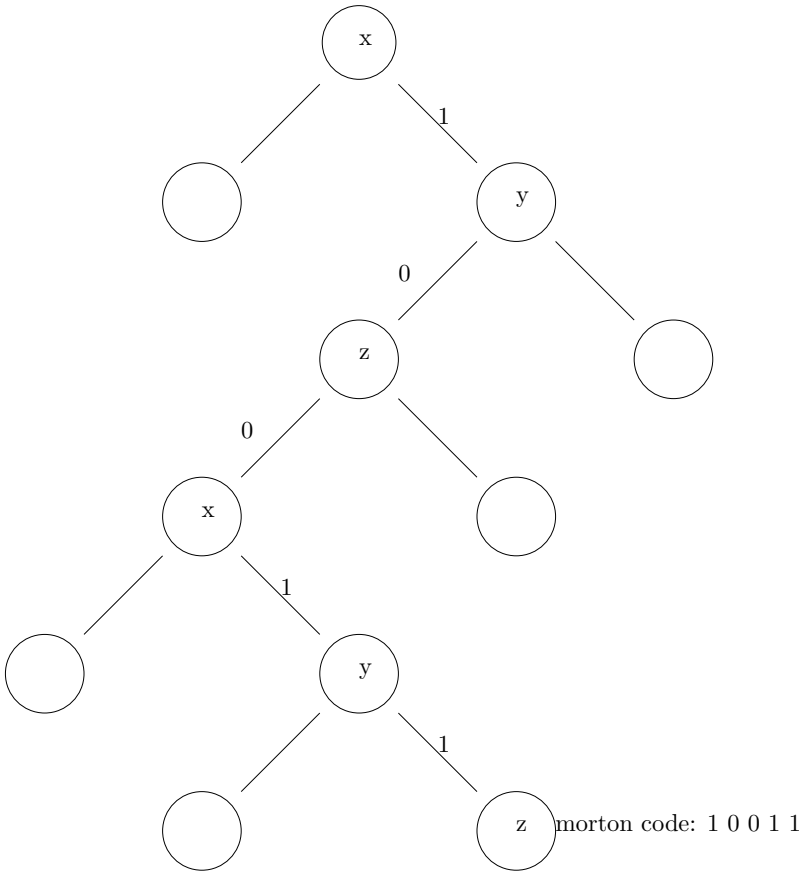

Figure 2.15: Morton code from BSP paths.

CHAPTER 3

External Sorting with LRU Cache

In order to find the nearest neighborhood and furthermore the normals the point cloud has to be sorted first. This is because the streaming approach is a two step process. First the point cloud has to be sorted along the main axis and then a plane is swept along this axis. The main axis is the axis with the widest spread. In this diploma thesis a tree based distribution sort (see Algorithm 19) strategy, to sort the point cloud externally, was developed. This approach is a distribution sort variant (see Chapter 2.3). The distribution sorting strategy distributes all points into buckets. After all points are distributed the buckets are sorted in parallel or one by one. Buckets are buffer with a fixed amount of storable points. The tree based distribution sort improves this strategy in that it uses an AVL tree to distribute the points into the buckets. First the point is read from the input stream. Then an insertion leaf node in the balanced binary search tree has to be found. The search key is given by the position value at the main axis. A balanced binary search tree (e.g. AVL tree) guarantees a complexity of $O(log(n))$ for the find, insertion and removal procedure. Each of the leaf nodes contain buckets with a fixed amount of storable points. Whenever the bucket of a leaf node is full this leaf node has to be split into two new leaf nodes. The splitting strategy is to take the median of the leaf node bucket and distribute the points into two new buckets. Each of the newly created buckets contains half of the points. This leads to a guaranteed splitting policy of 50 percent full buckets at any leaf node. The points inside the nodes of the AVL tree are managed by a least recently used (LRU) cache, so only nodes that have recently been visited hold their points in memory. After all points have been inserted the buckets are sorted. The last step is to traverse all nodes inorder and write the sorted buckets in a new point stream.

Algorithm 19 Tree Based Sorting

Require: A valid point stream.
Ensure: A valid sorted point stream.

```
1: while There is an existing point in the stream. do
2:    Read point from sorted stream.
3:    Find an insertion leaf.
4:    if bucket in the insertion leaf is not full then
5:       Insert into leaf node bucket.
6:    else
7:       if The least recently used cache is full. then
8:          Copy the least recently used bucket to the disk file.
9:       end if
10:      Create two new leaf nodes.
11:      Split insertion node bucket into the two new leaf node buckets.
12:      Balance the binary search tree (see Algorithm 20).
13:      Register the two new buckets in the least recently used cache.
14:      Insert the new point into the left or right leaf node bucket.
15:   end if
16: end while
17: Sort all buckets.
18: Build a valid sorted point stream.
```

3.1 AVL Tree

As mentioned before the complexity of $O(log(n))$ can only be guaranteed if the binary search tree is balanced. The AVL tree [TO96] is a self balancing binary search tree. It guarantees that the height difference between any left and right sub tree is at most one. Therefore it is one of the best trees to guarantee the proposed logarithmic complexity. The balancing strategy (see Algorithm 20) of the AVL tree is the fundamental operation.

Algorithm 20 AVL-Tree Balance

Require: A valid tree node.
Ensure: A balanced AVL tree.

```
1: while The tree node has a parent. do
2:    Call balance tree node (see Algorithm 21).
3:    Set the parent node as the new tree node to be balanced.
4: end while
5: Call balance tree node (see Algorithm 21).
```

There are four cases for balancing an AVL tree which have to be considered whenever a tree node is inserted or removed.

- Rotate left dominant (see Figure 3.1).
- Rotate right dominant (see Figure 3.2).
- Rotate left and right dominant (see Figure 3.4).
- Rotate right and left dominant (see Figure 3.3).

The Algorithm 21 shows how the four cases have to be used in order to keep an AVL tree in balance.

Algorithm 21 AVL-Tree Balance Tree Node

Require: A valid tree node.
Ensure: A balanced sub tree.

```
 1: Compute the height difference between the left and right sub tree.
 2: if The height difference is plus two. then
 3:    Compute the height difference of the right sub tree.
 4:    if The height of the right subtree is minus one. then
 5:       Rotate the sub tree left node to the right (see Figure 3.3).
 6:       Rotate the sub tree root node to the left (see Figure 3.3).
 7:    else
 8:       Rotate the sub tree root node to the left (see Figure 3.2).
 9:    end if
10: else
11:    if The height difference is minus two then
12:       Compute the height difference of the left sub tree.
13:       if The height of the left subtree is one then
14:          Rotate the sub tree right node to the left (see Figure 3.4).
15:          Rotate the sub tree root node to the right (see Figure 3.4).
16:       else
17:          Rotate the sub tree root node to the right (see Figure 3.1).
18:       end if
19:    end if
20: end if
```

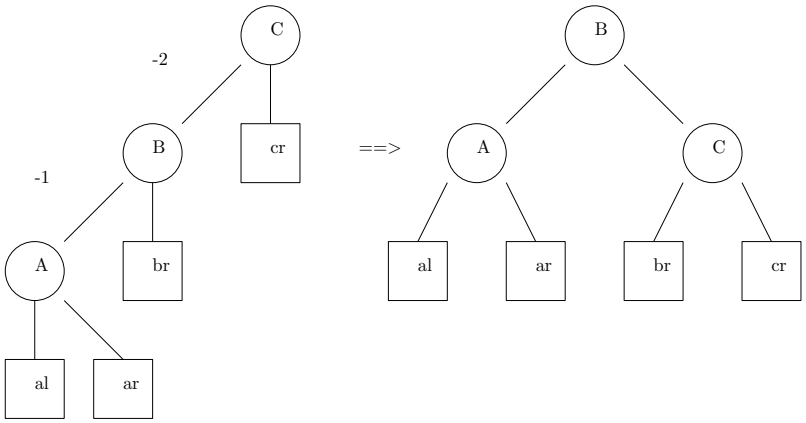

Figure 3.1: Rotate left dominant.

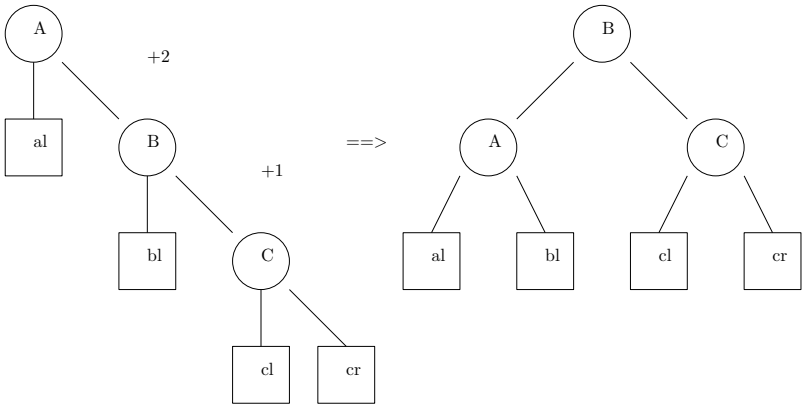

Figure 3.2: Rotate right dominant.

3.2 LRU Cache

In order to keep only those buckets in main memory that are recently updated a simple but effective least recently used cache was implemented. This least recently used cache is basically a linked list which updates the status of any recently used bucket whenever an element is inserted. Therefore the bucket has to be removed from its current position and reinserted into the first position of the list (see Figure 3.5).

Furthermore whenever there are too many buckets in main memory the least recently used bucket will be made persistent (flushed to disk file) and removed from the list. This least recently used bucket is always on the last position of the list. This simple but effective strategy allows to keep only those buckets in main memory that are recently

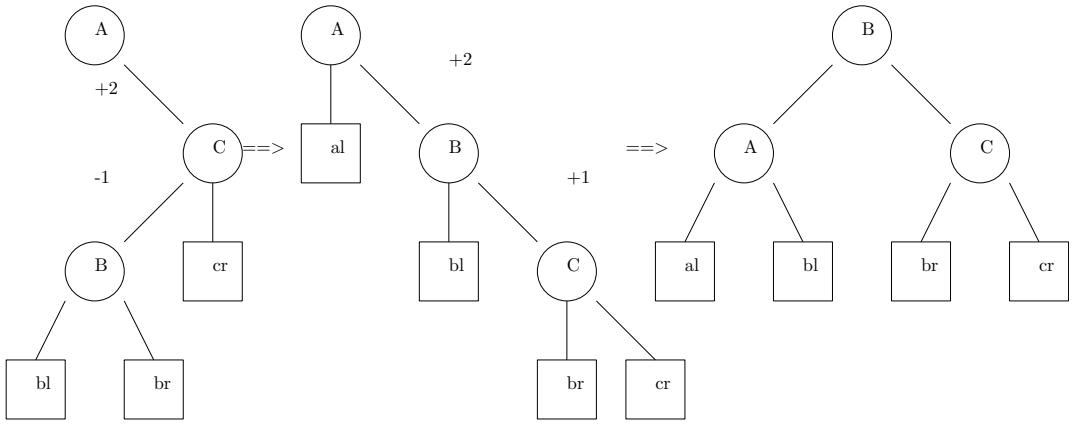

Figure 3.3: Rotate right and left dominant.

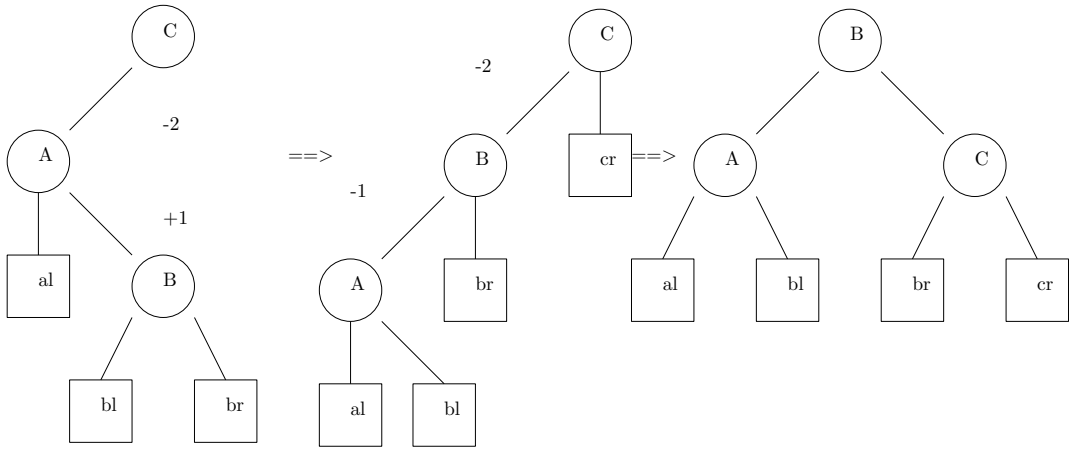

Figure 3.4: Rotate left and right dominant.

used by the tree insertion procedure (see Algorithm 19). The speed enhancement that comes from the least recently used cache strongly depends on the distribution of the points in the input stream. If the points are in a geometrically sense close together then the least recently used cache strongly speeds up the process. This is the case, because the nodes of the AVL tree that have been visited for the previous points will also be visited for the next points, so only few I/O accesses occur during the insertion process. If the points are further away then the least recently used cache will lead to no or very low speed enhancement because the points in the nodes have to swapped in and out of memory more often.

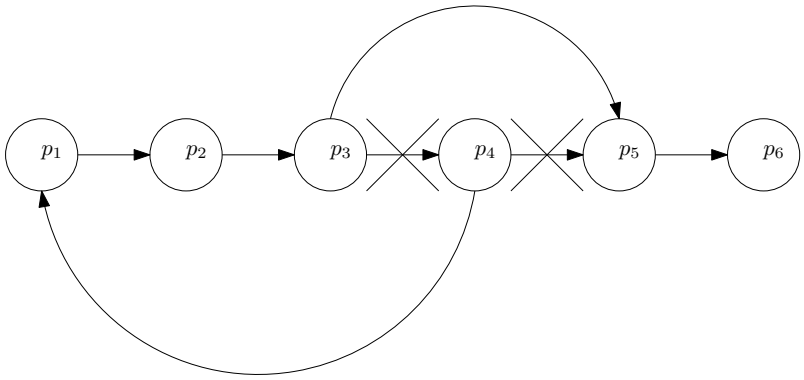

Figure 3.5: LRU Cache upadate procedure for an active point p_4.

3.3 Contribution

Tree based sorting is a simple and effective method to sort point clouds externally. The AVL tree allows to find the bucket in which a point has to be inserted with $O(log(n))$ complexity. The LRU cache keeps only those buckets in main memory that are recently updated. All buckets that store points which are far away from the recent points are kept persistent. The sorting stage sorts all the buckets parallel by using several threads. The number of available threads is dependent on the number of threads supported by the CPU. That means if the CPU allows to process the sorting stage in more than one thread, then all threads will be used to sort the buckets in parallel. The main advantage in this approach is that the main memory is used as second level cache because of the LRU cache. The LRU cache takes advantage of the spatial relationship of the points in that it keeps only those buckets in main memory that are recently updated. This is not possible in a merge sort approach. To the knowledge of the author of this diploma thesis there is no similar approach that takes advantage of the spatial relationship of the points.

CHAPTER 4

Point Streaming Framework

Estimating normals in a very large point cloud is a non trivial task, thats because it relies on finding the nearest neighborhood for any point in the point cloud. The nearest neighborhood of a point is useful not only for normal estimation but also for many other tasks. For example it is useful for surface reconstruction, fairing and many more. In this chapter an approach to process a very large amount of point based datasets is shown. The main focus is laid on finding the nearest neighborhood for any point in the point cloud. The main concern in finding the k nearest neighborhood (kNN) is to find a data structure that can handle range query requests for the kNN of a single point with $O(k * log(M))$ complexity. K are the nearest neighbors and M is the number of points managed by the data structure. Several data structures with the required time complexity exist for in-core processing of point clouds [Sam06] [Mou10] [Cha06]. Out-of-core data structures on the other hand are not that easy to find. The kd tree for example can handle the range query in the proposed complexity but if used as out-of-core data structure [AHW89] the range queries have to handle lots of disc accesses. This means the process of finding the nearest neighborhood for any point becomes very slow because each point has to handle a significant large number of I/O requests. Point streaming is another possibility to handle the problem with reduced I/O accesses. This approach divides the process for finding the k nearest neighborhood into two steps. The first step sorts the point cloud along an axis. This task is discussed in Chapter 3. The second step processes the points sequentially. This strategy allows to find the k nearest neighborhood by reading the points one by one and searching in main memory for the nearest neighborhood without any I/O disk accesses being involved. Even better it is possible to build a kd tree in memory. Which means that the search can be handled with the proposed complexity of $O(k * log(M))$. But this solution depends on the locality of the data. The locality can be guaranteed if the points are sorted along one direction. As mentioned in the work by [PS04], points can then be processed in a

streaming fashion, where operators work on the points that are currently held in main memory. The following chapter gives a short overview of the used point streaming framework and introduces the operator that was developed in this diploma thesis.

4.1 The Basic Idea

The following strategy for finding the k nearest neighborhood and the normal for any point in a stream is based on the work by [PS04] by Renato Pajarola. It shows a convenient way to build a framework for stream processing of points in three dimensions. In the Algorithm 22 the top level of the framework is shown where so called local operators θ are applied to any point in the point stream. Any local operator implements a specific task in the framework. The read operator reads points from the input stream. The k nearest neighborhood operator finds the nearest neighborhood for any point in the point cloud. The normal estimation operator computes the normal for the points. The write operator writes points to the output stream. These are only four examples for local operators there are many more tasks that can be implemented as local operators. For example a surface reconstruction operator a fairing operator and many more. Local operators are connected by a first in first out (FIFO) queue which allows a deferred processing of the points. This diploma thesis follows the design suggestions of the work by [PS04] and improves on the k nearest neighborhood operator.

Algorithm 22 The Main Algorithm

Require: A valid point stream. A set of local operators $\theta_1 ... \theta_n$.
Ensure: Any point has a normal.

1: Sort the point cloud P along the axis with the widest spread (main axis).
2: **for all** $p_i \in P$ **do**
3: Call the local operators $\theta_n(...(\theta_1(p_i)))$.
4: **end for**

First the point stream is sorted along the main axis. The main axis can be chosen to be the axis with the widest spread. Then the local operators θ are called sequentially for any point in the stream.

4.2 Local Operators

Any implementation of a local operator has to take the following definitions into account.

Definition 1. *A local operator $\theta(p_i)$ performs a function on point p_i that computes or updates a subset of attributes A_i associated with p_i. As function parameters, $\theta(p_i)$, only*

accepts p_i, A_i and a set of points $p_i \in N_i$ within close spatial proximity to p_i (and all their associated attributes A_j) [PS04].

The meaning of this formal definition is that any operation that is performed on any point can only take those points in consideration that are in main memory at the time this point is active. Points are in main memory if they are referenced by any other point in main memory and if they are not fully processed by all local operators.

Definition 2. *A local operator $\theta_k(p_i)$ is streamable, if it is computed in one single invocation on p_i and not called recursively on points $p_j \in N_i$. Additionally the FIFO semantic of its Queue Q_k ensures no interference between consecutive operators Q_{k+i} [PS04].*

This second definition states that stream operators can only be called sequentially. Recursions within the operator cascade is not allowed. Furthermore the local operators are linked by a first in first out queue (FIFO queue). All elements that are fully processed within a local operator are stored in a FIFO queue. The next operator in the operator cascade will fetch all processed elements from the previous operator by withdrawing the elements from the FIFO queue.

4.3 A Point Streaming Assembly

The following Algorithm 23 shows how the local operators are used within this diploma thesis. The point streaming assemply is a concatination of the local operators as they are called in the sequence. In this diploma thesis four local operators are implemented. Those operators are discussed in the corresponding subsections of this chapter.

Algorithm 23 A Point Streaming Assembly

Require: A valid point stream. A read operator θ_1. A k nearest neighbor operator θ_2. A normal estimation operator θ_3. A write operator θ_4.

Ensure: point stream has normals

1: External sort the point cloud P along the axis with the widest spread (main axis).
2: **for all** $p_i \in P$ **do**
3: Call $\theta_4(\theta_3(\theta_2(\theta_1(p_i))))$
4: **end for**

External Sorting

The sorting step guarantees that all elements are sorted in one axis direction. This can also be seen as sorting a stack of planes. The sorting operator was developed as tree based distribution sort (see Chapter 3).

Read Operator

The read operator reads all points from the sorted stream sequentially.

Algorithm 24 Read Operator

Require: A valid sorted point stream.
Ensure: Point p_i is in main memory.

1: **if** There is an existing point p_i in the sorted stream. **then**
2: Read point p_i from sorted stream.
3: Insert point p_i into the FIFO queue.
4: **end if**

K Nearest Neighbor Operator

The k nearest neighbor operator is the heart of any point streaming assembly. The following algorithms show a fast and efficient way to solve the problem of building a k nearest neighbor graph. The efficiency in this solution comes from the fact that only a fraction of the whole point cloud is kept in main memory while the rest can reside out-of-core. Also the I/O file access is reduced to read and write one point after the other from the input stream and into the output stream sequentially.

The simplest algorithm that solves the problem is shown in the algorithm 25. The basic idea in this solution is to use a doubly linked list (z-list) to search for the nearest neighbors. The z-list stores all points sorted along the z axis.

The search phase is divided into two stages.

- The first stage searches active left from any point p_i along the z axis.
- The second stage searches passive right from the point p_i along the z axis.

The active search finds the neighborhood for a point p_i by updating the heap of that point with any point in the z-list. The heap stores only k points and rejects all elements that are greater than the maximum distance in the heap. All points in the z-list are left from the point p_i and therefore only points in the direct neighborhood on the left side can be found.

The passive search updates the heap of the neighborhood points by the current active point p_i. The active point p_i is always right from the points in the neighborhood list. Therefore as soon as the neighborhood points are found in the active stage their neighborhood will also expand by the point p_i on the right side along the z axis.

The last step is that the newly read point p_i has to be inserted into the front of the z-list.

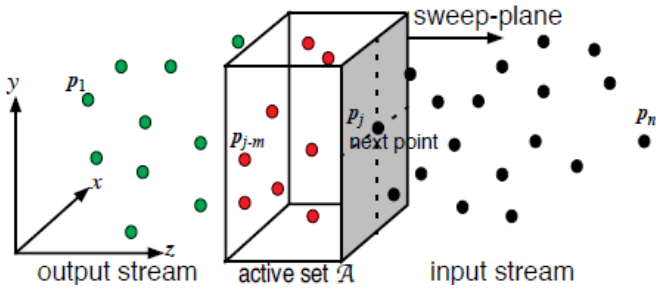

Figure 4.1: Sweeping a plane along the sorted z-axis [PS04]. All points in the active set are left from the point on the sweeping plane along the sorted z axis. The point on the sweeping plane is always right from all points in the active set along the sorted z axis.

Algorithm 25 Basic KNN Operator

Require: The previous read operator has valid points in the FIFO queue.
Ensure: Any point has k nearest neighbors.

while There is a point p_i in the previous FIFO queue. **do**
  Remove point p_i from previous FIFO queue.
  for all points p_j in the z-list **do**
    if The heap has k elements. **then**
      if The $distance(p_i, p_j) > distance(p_i, p_{max})$ with p_{max} being the maximum point in the maximum heap. **then**
        Remove point p_j from the z-list.
        Insert point p_j into the FIFO queue.
      else
        Update the maximum heap of the point p_i with the point p_j (see Algorithm 8).
      end if
    else
      Insert into the maximum heap of the point p_i with the point p_j (see Algorithm 10).
    end if
  end for
  for all points p_j in the maximum heap of point p_i **do**
    Update the heap from the point p_j with the point p_i (see Algorithm 8).
  end for
  Insert the point p_i into the z-list.
end while

Normal Estimation Operator

The normal estimation operator computes the normal vector. In this diploma thesis the QR Algorithm [WHP02] is used that solves the system of normal equations from the orthogonal linear least squares fitting in $O(n^2)$ complexity.

Algorithm 26 Normal Estimation Operator

Require: The previous k nearest neighborhood operator has valid points in the FIFO queue.
Ensure: Point p_i has a normal vector.

1: **while** There is a point p_i in the previous FIFO queue. **do**
2: Remove point p_i from previous FIFO queue.
3: Compute the normal vector for the point p_i.
4: Insert point p_i into the FIFO queue.
5: **end while**

Write Operator

The write operator first releases all references a point holds to its nearest neighbors and then writes the points to the output stream.

Algorithm 27 Write Operator

Require: The previous normal estimation operator has valid points in the FIFO queue.
Ensure: Point p_i is stored in disk file.

1: **while** There is a point p_i in the previous FIFO queue. **do**
2: Remove point p_i from previous FIFO queue.
3: Write point p_i into stream.
4: **end while**

4.4 A Case Study

In this section a simple case study for the k nearest neighborhood operator is shown. The sampling distance is uniform along the z-axis and the x and y values are all the same. The neighborhood graph operator has to find two neighbors for each point.

The first three steps in the case study can be seen in Figure 4.2. The first step starts with one point p_1. This point is singular and finds no neighbors. In the second step the point p_2 is inserted. The active search finds the point p_1 as direct neighbor on the left side. The passive search updates the point p_2 in the neighborhood heap of the point p_1 on the right side. The green half circle in the Figure 4.2 is the active search radius and the red circle is the passive search radius. The search radius is the length from the active point to the farthest away point in the found list of points. In this case this is the length from point p_2 to the point p_1. The event horizon is now at the plane l_2. The event horizon is the position of the sweeping plane at the active point. In this case this is the plane l_2 at the point p_2.

The third step inserts the point p_3. The event horizon is moved to the event plane l_3. The active search finds the points p_1 and p_2 and the passive search updates the point p_3 in the neighborhood heap of point p_1 and point p_2. The passive search radius for the point p_1 is widened to the point p_3.

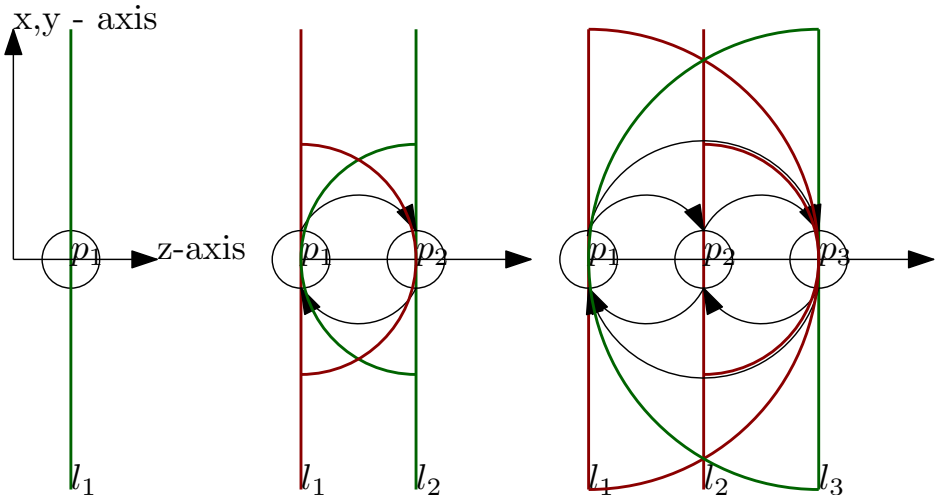

Figure 4.2: The first three steps of the case study.

The fourth step in the case study can be seen in Figure 4.3. The event horizon is now at l_4 and the event point that triggered the new event is inserted as point p_4. After the neighborhood points are found, they are updated as seen before. A cleanup step clears the active points list. Point p_1 and point p_2 are outside the search range and therefore

there is no need for the active search step to traverse those points whenever a new event point triggers a new search phase. This is because if the passive search range radius (red line) is less than the active search range radius (green line) of the event point p_4 then those points can not make a contribution to any search step that follows.

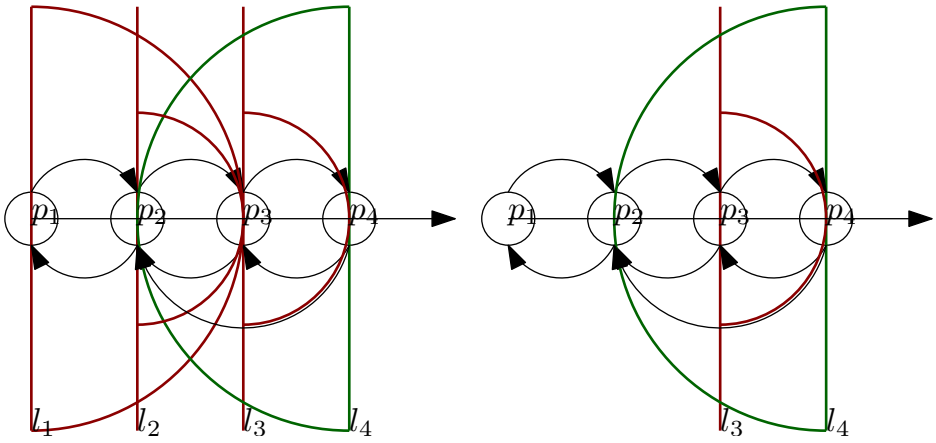

Figure 4.3: The fourth step of the case study.

The fifth step inserts point p_5 and finds the points p_3 and p_4 as can be seen in Figure 4.4. After those points are updated with point p_5 the cleanup step clears the point p_3 from the active list. This fifth step is repeated until there is no more event point.

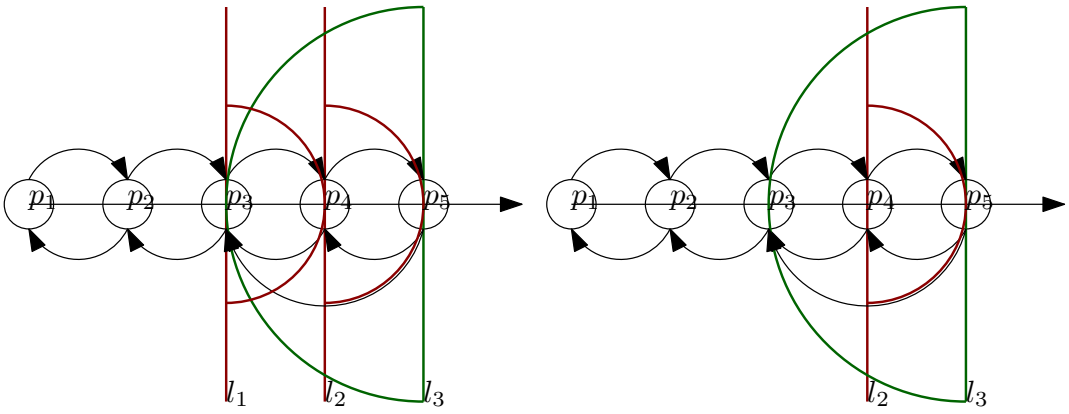

Figure 4.4: The fifth step of the case study.

4.5 Improvements

The following discussion shows how the basic k nearest neighbor operator can be improved. Therefore three operators will be introduced that can be used instead of the basic operator. All three operators were developed in this diploma thesis and tested for their practicability when working with real life point cloud datasets. After the introduction of the operators a short summary concludes in what cases the introduced operators are useful.

KD-tree Operator

The basic Algorithm 25 has one major disadvantage. If there are many points in the z-list, the algorithm tends to become slow. In fact the active searching procedure has $O(M * log(k))$ complexity, with M being the number of points in the z-list and k the number of nearest neighbors. This is because the search has to visit all points in the list and insert them into the binary maximum heap. In order to speed up the process the algorithm has to be enhanced by a search in the x and y axis directions.

The Algorithm 28 describes a way to speed up the process of finding the k nearest neighborhood by using a kd-tree search. As in the basic algorithm the active point is inserted in a z-list, but it is also managed by a kd-tree. This enables to search the points for the k nearest neighborhood in $O(k*log(M))$ complexity, with k being the number of nearest neighbors and m being the number of points in the kd-tree. In order to guarantee the complexity of $O(k*log(M))$ the kd-tree has to be as balanced as possible. Balancing a kd-tree is a non-trivial task, in fact the only known strategy to balance a kd-tree is to fully rebuild it (see Algorithm 14). The following advanced Algorithm 28 uses a simple but effective strategy to keep the kd-tree balanced with an amortized complexity of $O(M * log(M))$.

Whenever a constant number of points are removed from the kd-tree the kd-tree will be completely rebuild. This method is similar to the well known mark and sweep garbage collection algorithm [RJ96]. Points will be marked as removable, whenever a deallocation of that point occurs. After a constant number of points are marked as removable those points are really deleted within a sweep procedure. In this case the sweep procedure is given by the rebuild algorithm.

This method is similar to the one introduced by Pajarola and Boesch [PB09] which has been developed in parallel to the work in this diploma thesis.

Uniform Grid Operator

The simplest implementation of a local operator that finds the k nearest neighborhood is to use a uniform grid for the x and y axis. The grid cells store the points in a z-list. The n-dimensional space will be divided into sub-spaces of equal size in the x and y axis

Algorithm 28 KNN Operator

Require: The previous operator has valid points in the FIFO queue.
Ensure: Any point has k nearest neighbors.

1: **while** There is a point p_i in the previous FIFO queue. **do**
2: Remove point p_i from previous FIFO queue.
3: Search nearest neighborhood for point p_i in the kd-tree (see Algorithm 16).
4: **for all** points p_j in the maximum heap of point p_i **do**
5: Update the heap from the point p_j with the point p_i (see Algorithm 8).
6: **end for**
7: **for all** points p_j in the z-list **do**
8: **if** The $distance(p_i, p_j) > distance(p_i, p_{max})$ with p_{max} being the maximum point in the maximum heap. **then**
9: Remove point p_j from the z-list.
10: Mark the point p_j as removed.
11: Insert point p_j into the FIFO queue.
12: **end if**
13: **end for**
14: The recently read point p_i will be inserted into the kd-tree (see Algorithm 13) and the z-list.
15: Rebuild the kd-tree as soon as a constant number of points are marked as removable (see Algorithm 14).
16: **end while**

direction. Regardless of the coarseness of the grid, this solution will always be distribution dependent. This means if the dataset has regions that are sampled very densely in comparison to all other regions then only a few cells will store the whole information. In the case of datasets with densly sampled regions the grid solution will perform as bad as the afore mentioned basic k nearest neighborhood operator (see Algorithm 25). But if the point cloud is uniformly sampled then this solution will perform faster then the basic operator.

KD-tree Region Operator

The following k nearest neighborhood operator is different from the above (see Algorithm 30). While all the above operators search for the k nearest neighborhood one point after the other, the region operator reads and processes point regions along the z axis direction.

The first step in the algorithm reads a complete region into the main memory. This region will be merged together with the previously read region. The two regions together are building a kd-tree. The z distance of one region has to be large enough or else the

Algorithm 29 KNN Grid Operator

Require: The previous operator has valid points in the FIFO queue.
Ensure: Any point has k nearest neighbors.

1: **while** There is a point p_i in the previous FIFO queue. **do**
2: Remove point p_i from previous FIFO queue.
3: Search nearest neighborhood for point p_i in the uniform grid (see Algorithm 11).
4: Find the grid cell where the point has to be inserted.
5: The recently read point p_i will be inserted into z-list of the uniform grid cell.
6: **end while**

neighborhood will not find the k nearest neigborhood for all the points. If the region is too small than the search step will fail to find all k nearest neighbors. The decision how great the distance has to be depends on the distribution of the points in the point cloud. This means this method works best for point clouds with a nearly uniform sampling rate.

The active search step searches for the neighborhood sequentially point by point in the z axis direction left from the point. This search is different from the previous search strategies. It starts with a search range radius that is set to infinity. As soon as the first nearest neighbor is found the search radius will be set to the approximate distance value $distance(p, q_i) \leq k * (1 + \epsilon) * distance(p, q_j), i \neq j, \epsilon \geq 1$ (see Section 2.4).As mentioned in the previous section K Nearest Neighbor Graph subsection Construction the approximation is distribution dependent. This means if the point cloud was sampled with a nearly uniform sampling rate then this method is very fast and accurate. A nearly uniform sampling rate will enable to find all the k nearest neighborhood by the approximation $distance(p, q_i) \leq k * (1 + \epsilon) * distance(p, q_j), i \neq j, \epsilon \geq 1$. If the sampling rate is not nearly equal this strategy will not find the k nearest neighborhood for all points or it will find nearly all points in the whole region as nearest neighbors.

The last step in the algorithm is the passive search step (see Algorithm 8), which is the same as in the afore mentioned algorithms.

Conclusion

The basic k nearest neighbor operator shows a way to process points sequentially and independent of the distribution of the point cloud. But this operator is not practicable because the active search phase will take far too long to search in the list for nearest neighbors. This fact leads to three optimizations for the problem.

The kd-tree operator improves the basic k nearest neighbor operator by partitioning the space in the x and y axis using a kd-tree. The points are managed in the z-axis direction by a list and in the x and y axis by a kd-tree. The active search phase is searching in the kd-tree for the nearest neighbors. The implementation of this operator

Algorithm 30 KNN Region Operator

Require: The previous operator has valid points in the FIFO queue.
Ensure: Any point has k nearest neighbors.

1: **while** There is a point p_i in the previous FIFO queue. **do**
2: Remove the point p_i from the previous FIFO queue.
3: Store the point in a new region (array).
4: **end while**
5: Read from the previous operator as long as a distance from the last region to the new region in the z direction is reached.
6: Build the kd-tree with the previus region and the new region.
7: Search nearest neighborhood for point p_i in the kd-tree (see Algorithm 17).
8: **for all** points p_j in the maximum heap of point p_i **do**
9: Update the heap from the point p_j with the point p_i (see Algorithm 8).
10: **end for**

shows great results. It can handle arbitrary distributions of the point cloud sampling while keeping only a small fraction of the point cloud in main memory. This operator is the prefered strategy for solving the problem of finding the k nearest neighborhood. All point clouds in the results section where tested with the implementation of this operator.

The uniform grid operator is the logical extension to the basic k nearest neighbor operator. The underlying space is partitioned at the x and y axis using a uniform grid. Each grid cell manages a z-list to the points it containes. The active search is reduced to a simple lookup in which cell the point belongs and the eight neighbor cells. This operator works well for uniformly sampled point clouds. But real point clouds are not uniformly sampled and therefore this operator is not practicable either. When testing this operator with real life point clouds only a few regions were updated at a time while the rest were completely empty. This happens regardless of the coarseness of the grid. The reason for this is that points in real life point clouds tend to be clustered with different sampling density.

The kd-tree region operator follows a different strategy to search for the nearest neighborhood in that it processes regions instead of points. Therefore a region is read along the z-axis and merged with the previously read region into a kd-tree. The active search phase follows an approximative k nearest neighbor strategy. The first point inside the search region has to be found. The distance to this point is used as an approximation for the k nearest neighbor search distance. This means the new search range is given by k times the distance to the first point. The kd-tree region operator is distribution depended because it assumes that the k nearest neighbors have nearly the same distance than the distance to the first point. Therefore this strategy is only useful if the sampling distance between the points is nearly uniform.

4.6 Contribution

Point streaming allows to keep only a fraction of the whole point cloud in main memory while the rest of the point cloud can be kept persistent. The kNN operator is the most time consuming operator because it has to find the k nearest neighborhood graph. Four kNN operator were developed within this diploma thesis. The kd tree operator turned out to be both efficient and accurate when it comes to the normal estimation that is based on the k nearest neighborhood for any point in the point cloud. This operator is able to partition the space adaptive to the point cloud. Furthermore it is able to find the k nearest neighborhood independent of the distribution of the point cloud which is an improvement to the k nearest neighbor operator that is used by Pajarola [PS04]. Another very important aspect of this operator is that it allows to use a simple garbage collection strategy. This garbage collection strategy together with reference counting is the core of the algorithm which allows to keep only those points in memory that can contribute to the nearest neighborhood of any newly added point. The kNN operator has an overall complexity of $O(N * log(M))$ where N is the total number of points and M is the number of points in the kd tree. The developed operator is similar to the k nearest neighbor operator by Pajarola and Boesch [PB09].

CHAPTER 5

Implementation Details

This chapter will focus on some implementation details that were chosen in the implementation of the algorithms of the previous chapters.

5.1 Thread Pooling

Thread pooling is used by the external sorting algorithm. Once all points are distributed into buckets they have to be sorted one by one. This task of sorting the buckets can be split up into several threads, which are managed by a thread pool. The thread pool allows to distribute the threads to the available hardware threads [Jam08] on the central processing units (CPUs). Each hardware thread takes one sorting job at a time and stops if there is no more bucket to be sorted.

5.2 Memory Management

As mentioned before memory management is a hugh part of handling very large point clouds. Therefore several strategies to handle this issue were implemented.

External Sorting

The main issue in the sorting procedure is to keep only those parts in main memory that are necessary to distribute a new point into the corresponding bucket. Therefore each bucket allocates one MB. Whenever this space is exhausted an additional mega byte will be allocated. Each bucket can have up to eight mega byte before it will be split. This simple but effective strategy prevents memory fragmentation and enables very fast allocations from the available heap.

Intrinsic List

The LRU cache implementation is basically a doubly linked list that removes the least recently used bucket from the end of the list and reinserts an activly updated bucket at the beginning of the list. The doubly linked list is implemented as intrinsic list. An intrinsic list is a list that stores the linkage pointer to the previous and next list node in the data element itself. In the case of the LRU cache implementation the data elements are the buckets. The buckets store the link to the next and previous bucket. This intrinsic list strategy allows to keep the memory allocation of the list nodes at a minimum. The bucket can be seen as a list node and is allocated once when it is created together with the avl tree node. As a consequence the memory allocation is stable because the list node will not be allocated at insertion into the list but at the time of insertion into the avl tree.

Memory Pool

The point streaming framework implementation sequentially reads points from the sorted point stream. Therefore each point has to be allocated whenever it is read from the stream. In order to keep the number of memory allocations as low as possible a memory pool was implemented that manages the allocations and deallocations for each point. The concept of memory pooling is to preallocate as many elements as necessary and handle allocations and deallocations within one pool of elements. Therefore the elements have to be linked in a singly linked list and whenever an element will be allocated that element will be withdrawn from the pool. The deallocation of an element is to simply link the element within the pool.

Reference Counting

Whenever a node is referenced in the nearest neighbour heap, this node can not be deleted on kd-tree removal, else the point would not exist. It has to be in memory as long as the reference exists. The k nearest neighborhood consists of one center point and k edges to the nearest neighbor point. This means the points can not be simply deallocated when they are removed from the z-list. Once a point falls out of the range and furthermore is removed from the z-list it can be marked as deletable. Whenever there is a range search inside the kd-tree those points will be ignored. The exact time when a removable point can be deallocated is defined as whenever there are no more edges to that point in the k nearest neighborhood graph. In order to keep only those elements in memory that are referenced within the k nearest neighbor graph, the references will be counted. And as soon as the reference counter is null the point will be deallocated. As mentioned above the deallocation strategy of the point is to return the point into the memory pool.

There are 4 cases were the reference counter to a point will be updated:

- The point is read from the input stream.
- The point is written into the output stream.
- On nearest neighbor heap insertion.
- On nearest neighbor heap removal.

With this strategy the points have a controlled lifecycle that starts whenever a point is read from the input stream and stops whenever there are no more references to that point.

CHAPTER 6

Results

This chapter shows a few results and a discussion on the quality of the results which can be expected, with respect to the quality of the point cloud data. The selected point clouds in the following test samples consist of one or more scan positions. Each scan position shows an area of the whole point cloud and consists of millions of points. Most of the point clouds that where scanned by the laser range scanner showed massiv z-noise of about $+/-10$ standard deviation. Also some of the scan positions suffer from bad registration which leads to additional noise in the corresponding areas. As a consequence the distribution dependent k nearest neighborhood operators (grid operator, kd region operator) achieve bad results. Only the distribution independent adaptive k nearest neighborhood operators (kd tree operator) reach sufficient speed and accuracy. On point clouds with such a high z-noise, only a neighborhood of 20 points or more results in a homogenous orientation of point normals. The following images shows the reconstructed normals. The normals are false color coded by applying RGB colors with respect to the scalar values of the normal vector.

The algorithms were tested with the smaller point clouds on an Intel Core2 Duo E6400 system with 2 GHz, 2 GB RAM and a Seagate Barracuda 250 GB hard disk drive rotating at a speed of 7200 RPM. The Domitilla and Stephansdom point clouds were tested on an Intel Core2 Quad Q6600 system with 2.4GHz, 4 GB RAM and an RAID0+1 with 4 Western Digital VelociRaptor 300 GB that are rotating at 10.000 RPM.

The Table 6.1 shows the used datasets and how many points it contains.

The Table 6.2 lists the total timing in seconds for the external sorting step.

Figure 6.1 shows a comparison of the time needed to process the four k nearest neighborhood operators each executed with 20 neighbors.

The Table 6.3 lists the time needed to process the point streaming step.

The Table 6.4 outlines how much time the k nearest neighbor operator needs to process the k nearest neighborhood for all points.

Name	Number of Points	Scan Positions
Domitilla	1,921,537,902	1,826
Stephansdom	460,811,539	193
Hanghaus	8,537,584	1
Dragon	3,609,600	1

Table 6.1: The table shows the used datasets and how many points they contain.

Name	Timings in seconds
Domitilla	37,579
Stephansdom	5,040.63
Hanghaus	22.35
Dragon	8.14

Table 6.2: The table lists the total timing in seconds for the external sorting step.

Name	Timings in seconds
Domitilla	87,903.90
Stephansdom	64,521.80

Table 6.3: The table lists the time needed to process the very large point clouds with 20 neighbors.

6.1 Dragon

The dragon point cloud is the smallest point cloud used to show results. It was post processed to a point where it has nearly uniform sampling.

Figure 6.2 shows a comparison of the time needed to process the kd-tree based k nearest neighborhood operator with 10, 20, 30 or 40 nearest neighbors.

Figure 6.3 shows the Dragon point cloud processed with 10 nearest neighbors. There is visually no difference between a nearest neighborhood of 10, 20, 30 or 40 points.

Name	Timings in seconds
Domitilla	76,968.90
Stephansdom	61,822.50

Table 6.4: The table outlines how much time the k nearest neighbor operator needs to process the k nearest neighborhood for all points with 20 neighbors.

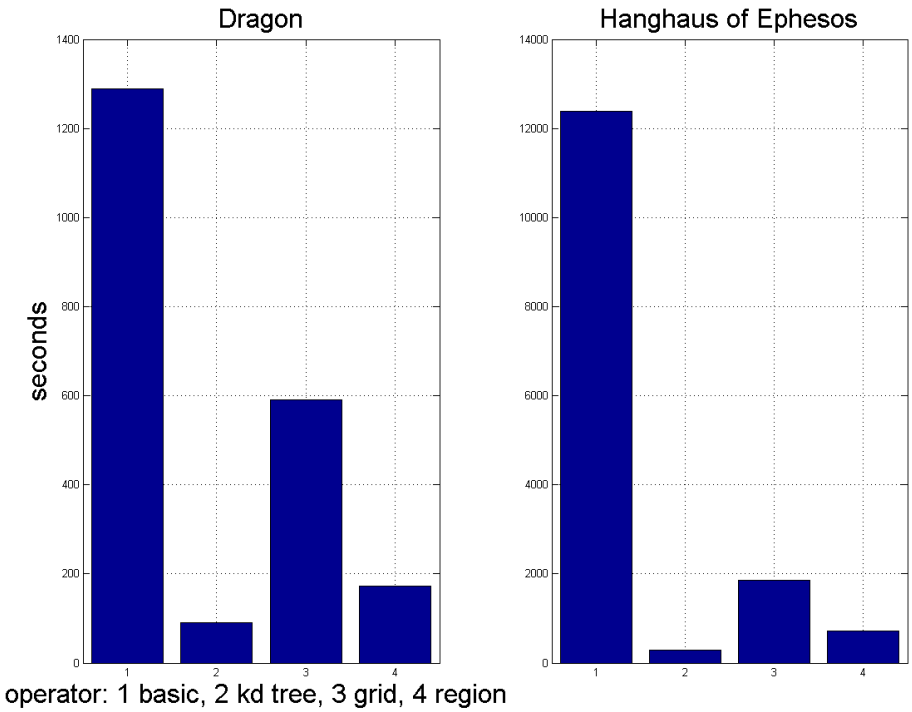

Figure 6.1: The bar plot shows a comparison of the time needed to process the four k nearest neighborhood operators each executed with 20 neighbors.

6.2 Hanghaus of Ephesos

The Hanghaus point cloud consists of several scans combined to a single point cloud and has not been post processed by any reconstruction procedure.

The Figure 6.4 compares the time needed to process 10, 20, 30 or 40 nearest neighbors. As expected the time to process 10 nearest neighbors is the smallest. But the time needed to process 20 nearest neighbors is the most efficient compared to all others.

The following Figure 6.5 shows the effect of different k nearest neighbors. The left most top Figure shows the k nearest neighborhood processed with 10 nearest neighbors. The right most top Figure shows the k nearest neighborhood processed with 20 nearest neighbors. The left most bottom Figure shows the k nearest neighborhood processed with 30 nearest neighbors. The right most bottom Figure shows the k nearest neighborhood processed with 40 nearest neighbors. As can be seen in the visual results the biggest change is between 10 and 20 nearest neighbors, while the difference between 20 and 30 or 40 nearest neighbors is small. As a consequence all the following very large

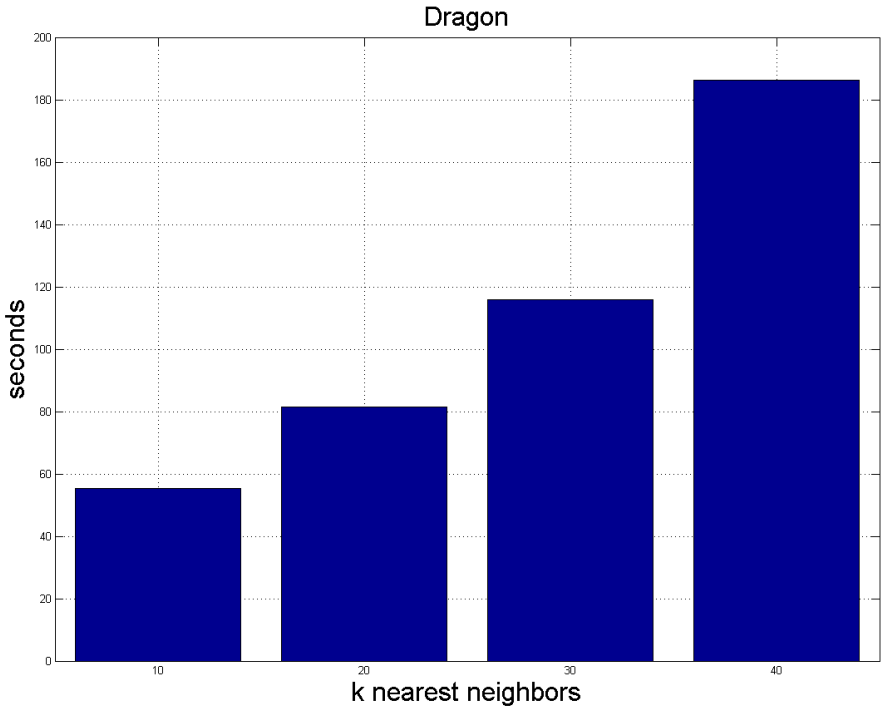

Figure 6.2: The bar plot shows a comparison of the time needed to process the kd-tree based k nearest neighborhood operator with 10, 20, 30 or 40 nearest neighbors.

point cloud data sets are processed with 20 nearest neighbors.

6.3 Stephansdom

The Stephansdom point cloud (see Figure 6.6) is a raw data set directly from the acquisition step and consists of 193 scan positions. It shows huge z-noise (see Figure 6.9) and suffers from small registration errors of the different scan positions (see Figures 6.7 and 6.8).

6.4 Domitilla

The Domitilla point cloud (see Figure 6.11) is the biggest point cloud that was tested with this point streaming approach. It consists of nearly two billion points that come directly from the acquisition step. The z-noise is as bad as in the Stephansdom point

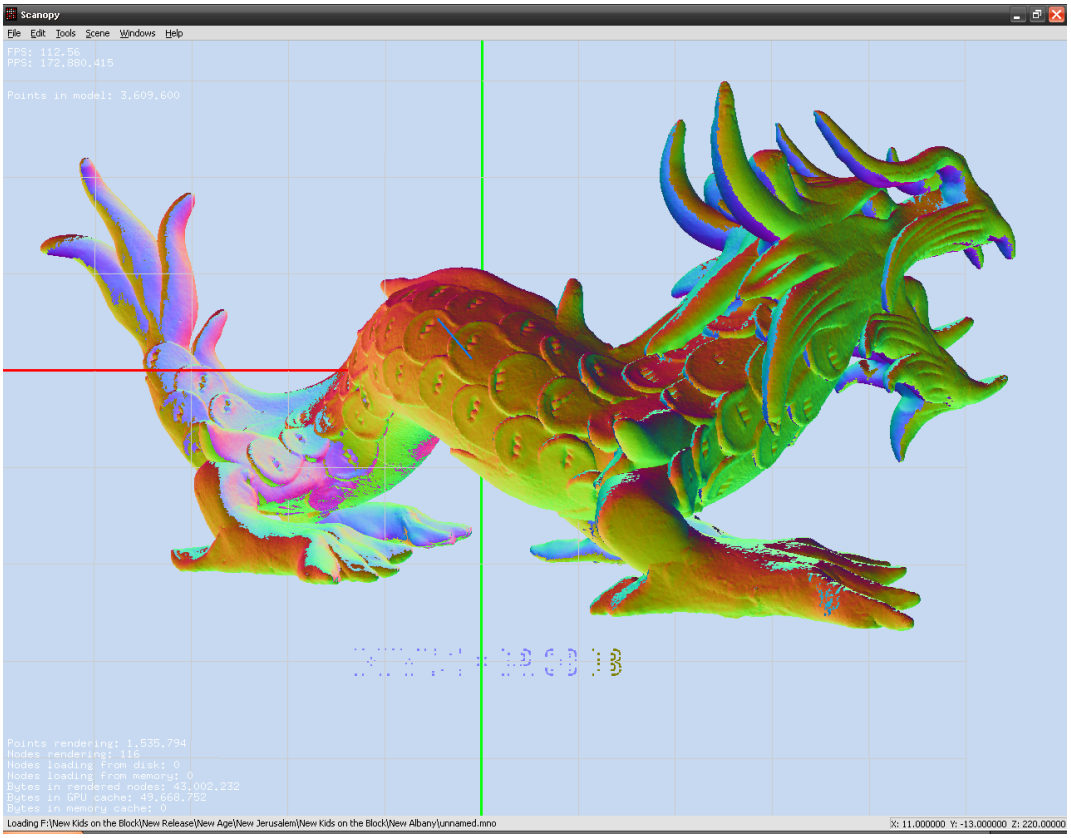

Figure 6.3: Dragon point cloud with 10 nearest neighbors.

cloud (see Figure 6.13 and 6.12). The small registration errors can accumulate to about half a meter between the points of two local neighboring scans. This is because it is very hard to find reference points in the underground of the Domitilla.

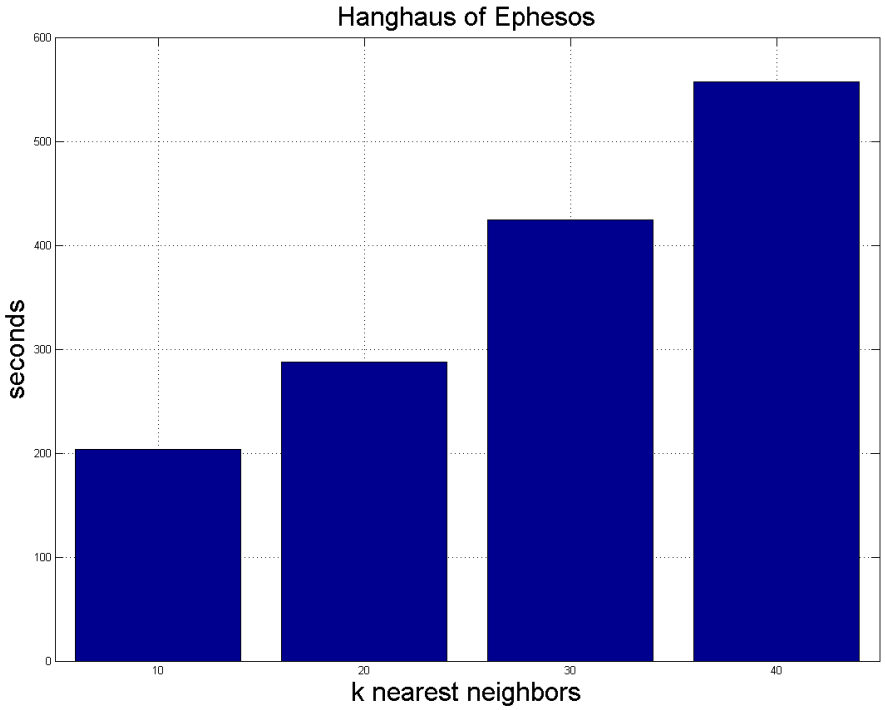

Figure 6.4: The bar plot shows a comparison of the time needed to process the kd-tree based k nearest neighborhood operator with 10, 20, 30 or 40 nearest neighbors.

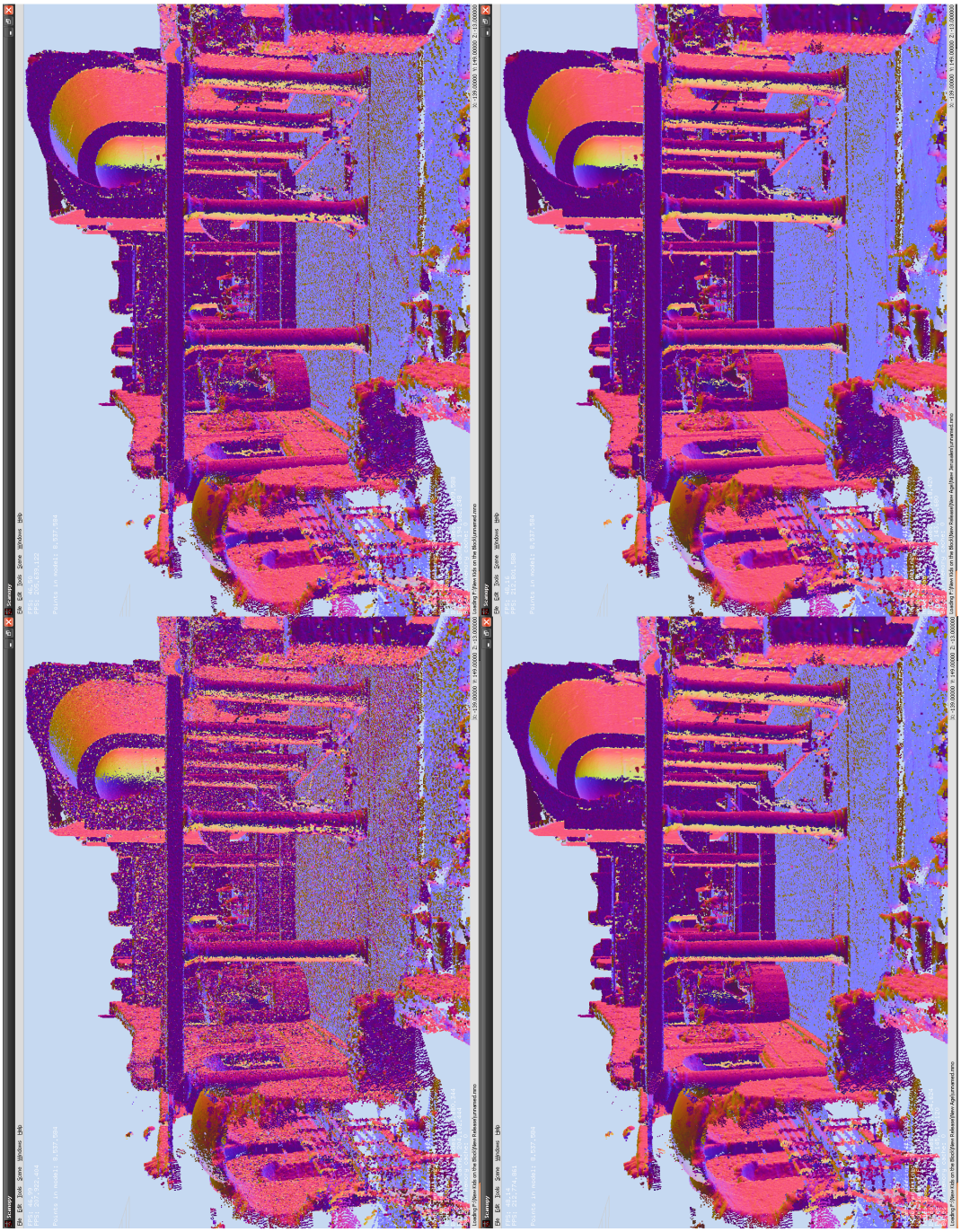

Figure 6.5: Hanghaus with 10, 20, 30 and 40 nearest neighbors.

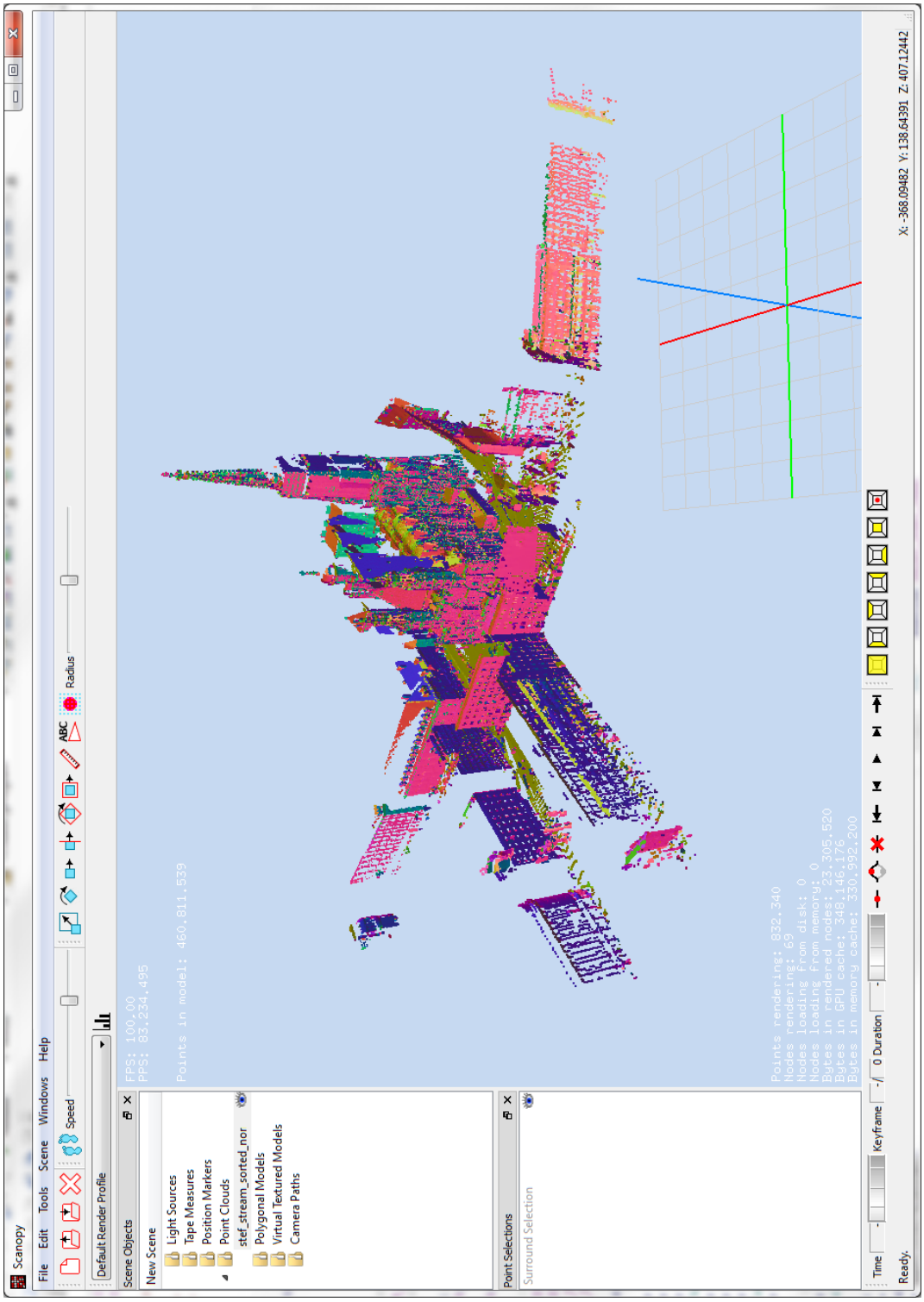

Figure 6.6: Outside view of the complete Stephansdom point cloud.

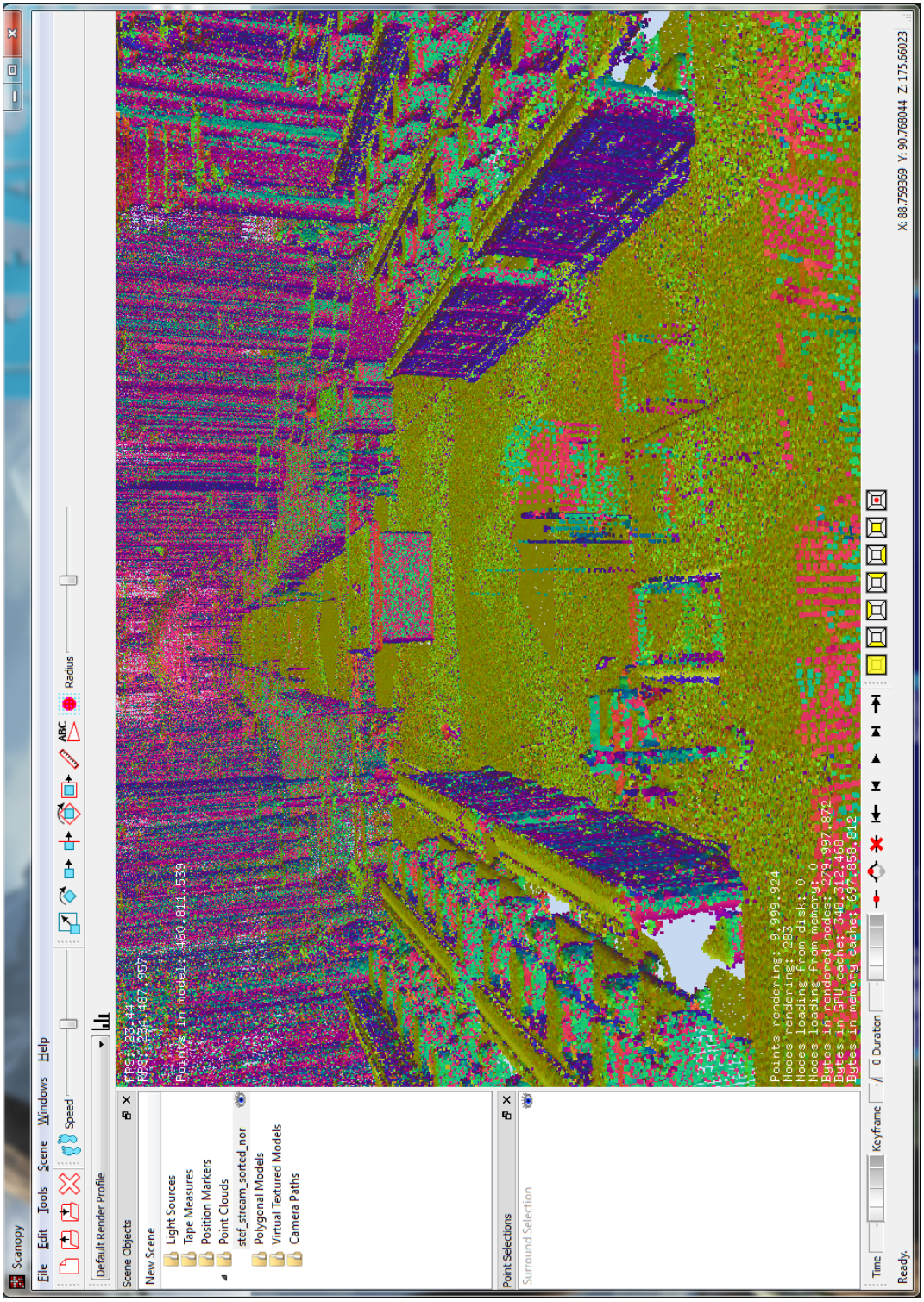

Figure 6.7: View on the altar of the Stephansdom point cloud.

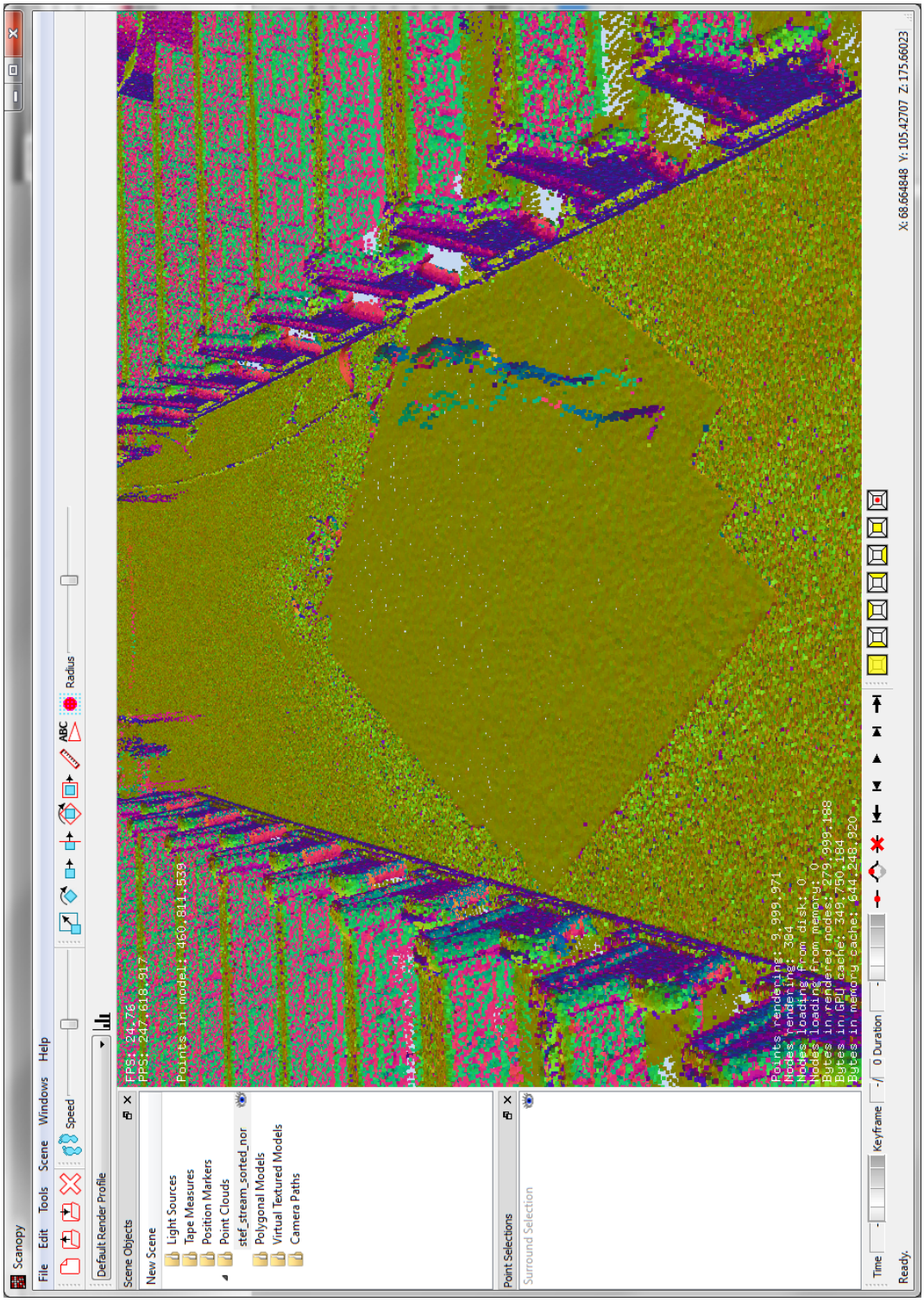

Figure 6.8: View from the main entrance to the altar.

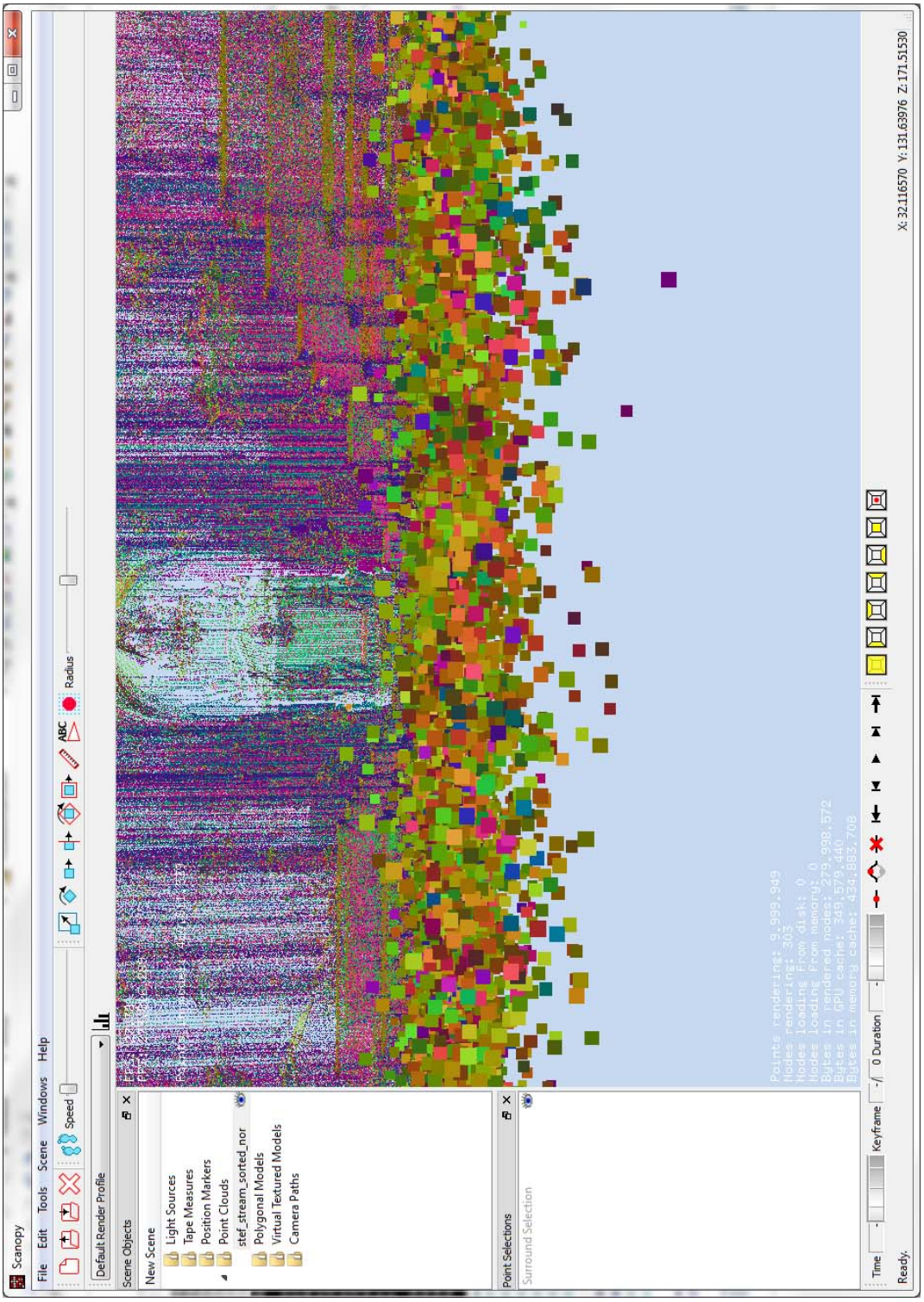

Figure 6.9: The floor area of the Stephansdom point cloud shows huge z-noise.

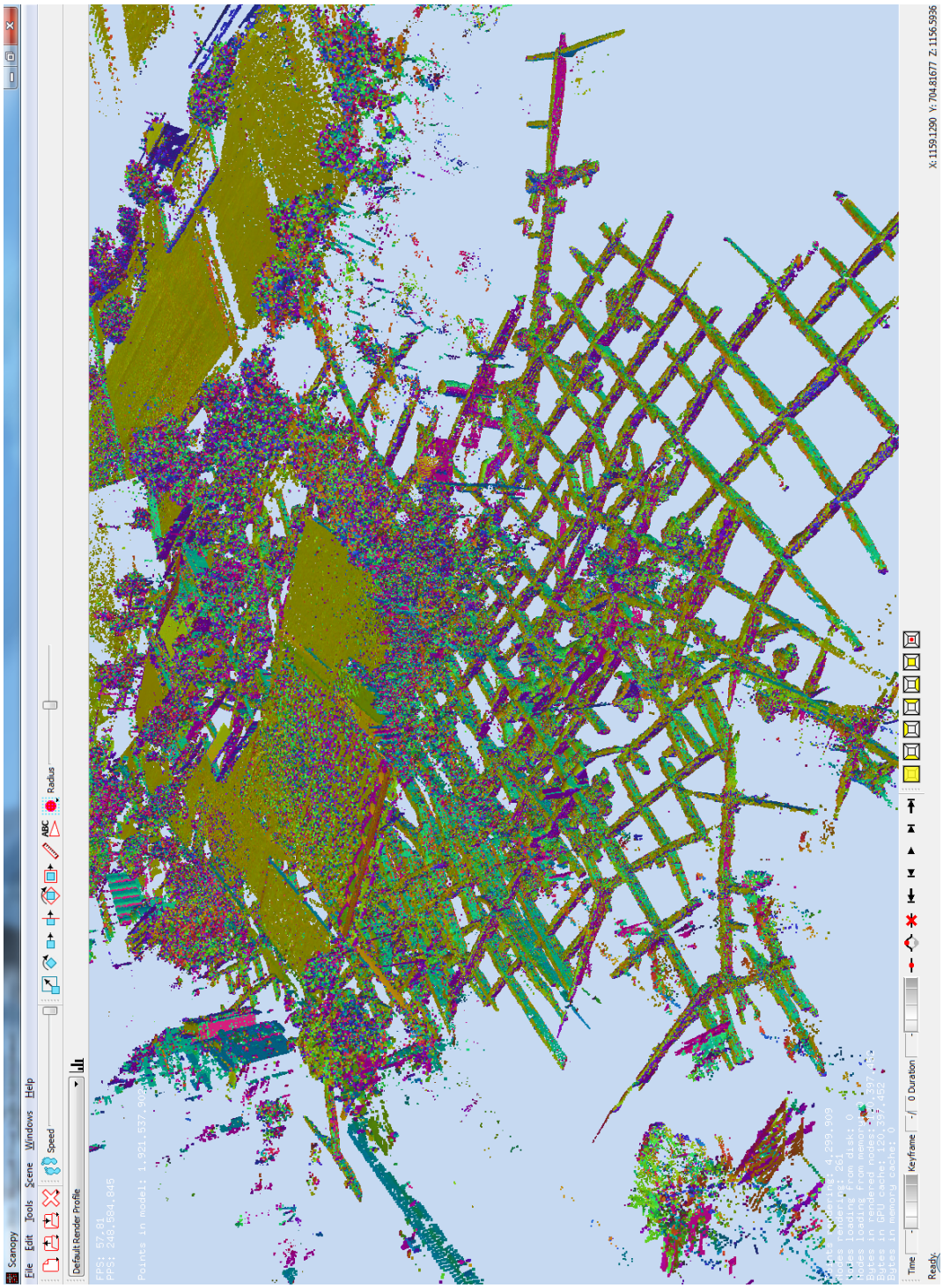

Figure 6.10: View on the complete Domitilla point cloud.

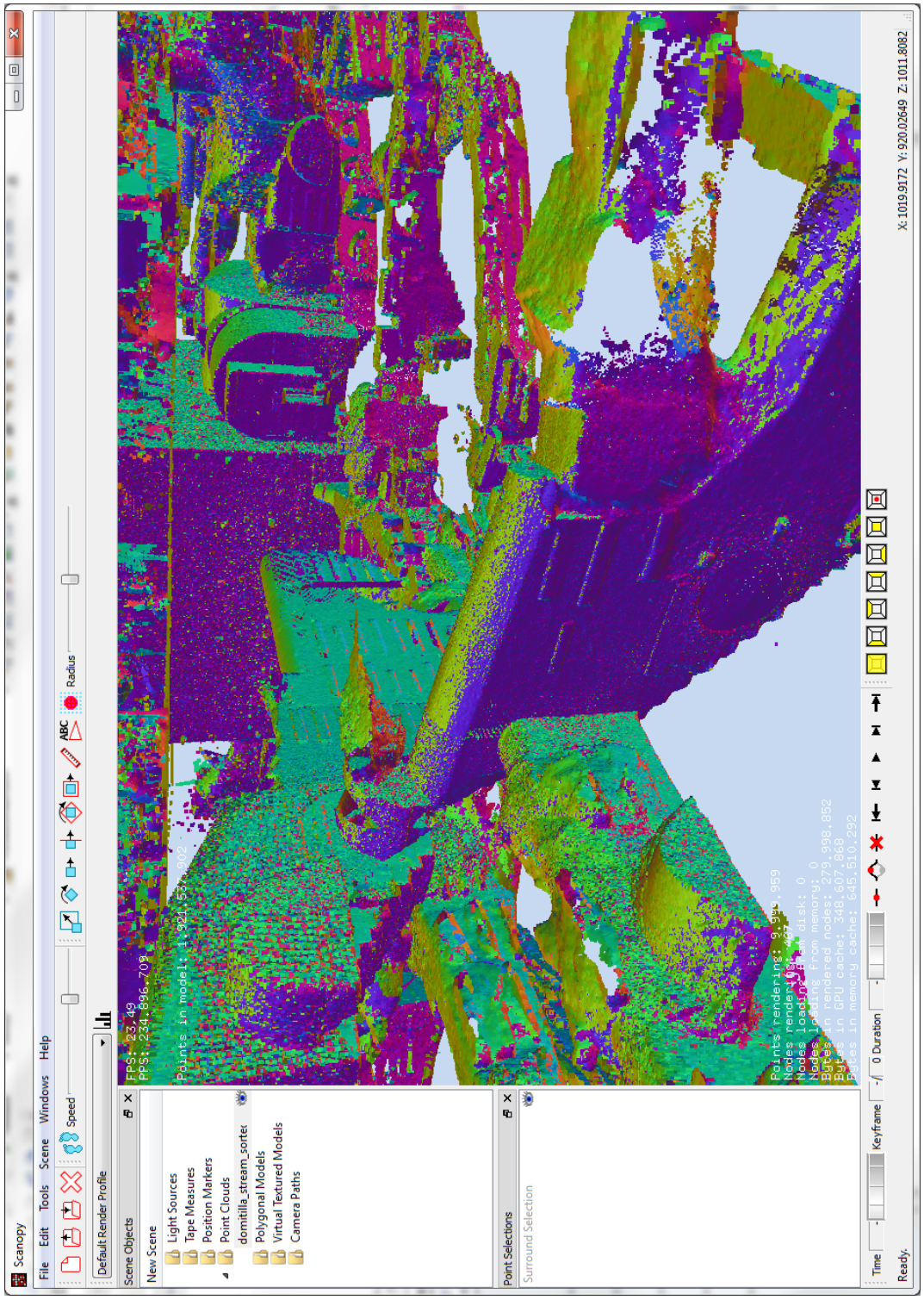

Figure 6.11: View from the Domitilla underground in direction of the Basilika.

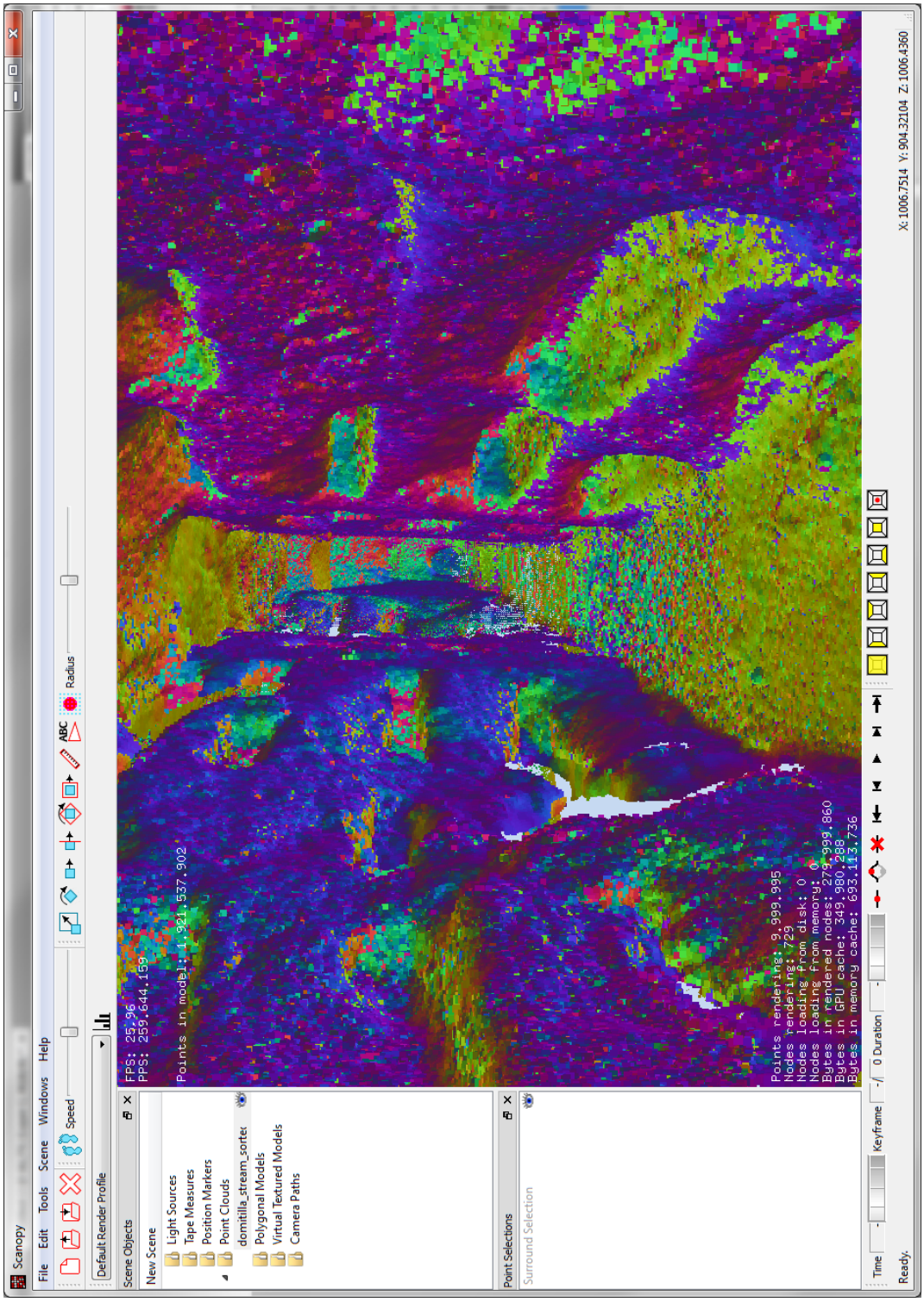

Figure 6.12: View on a gallery inside the catacomb.

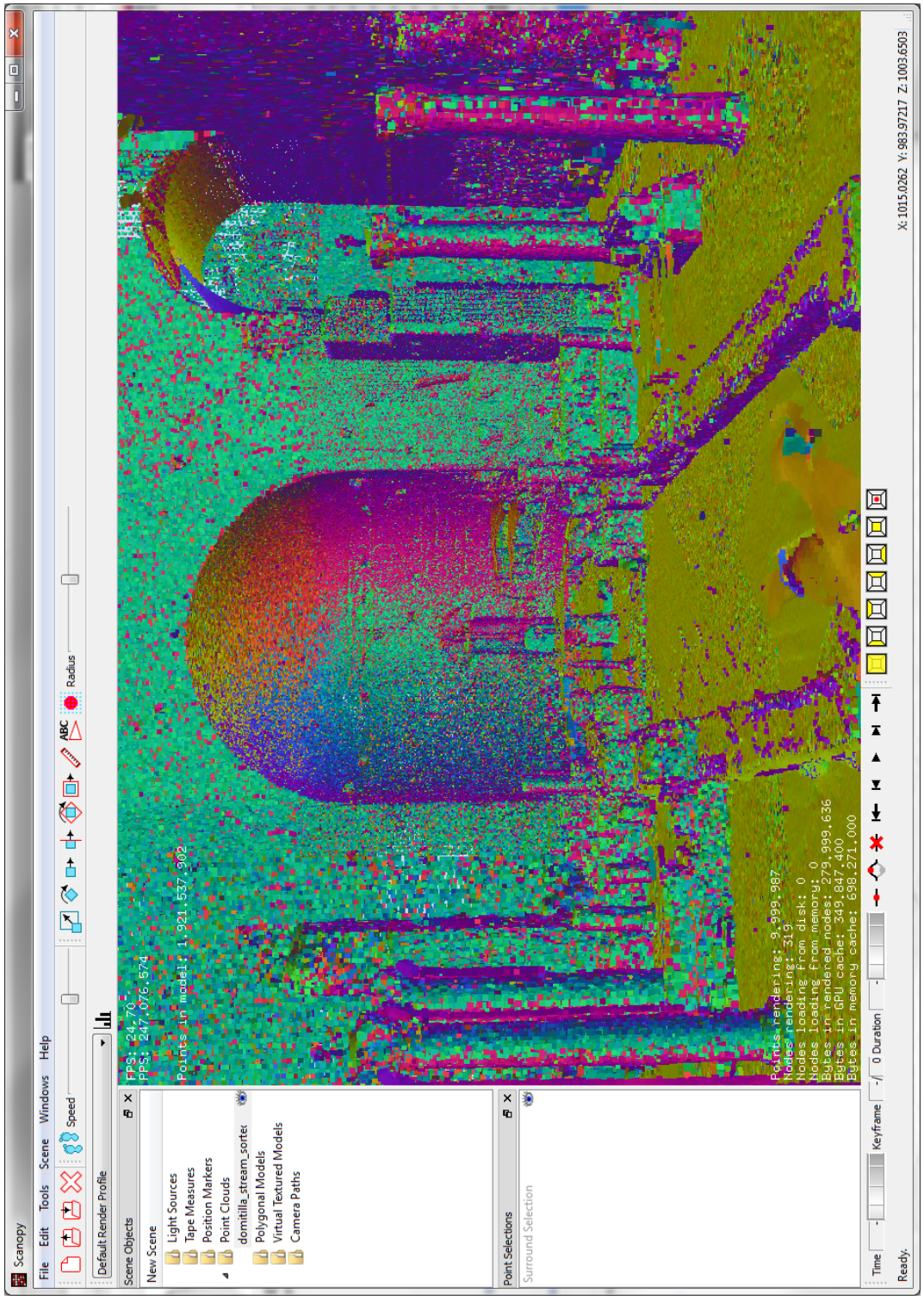

Figure 6.13: View on the Basilika Apsis.

CHAPTER 7

Conclusion and Summary

In this diploma thesis a strategy for stream processing points was presented. The main concern in any out-of-core application is to keep the number of I/O disk access as low as possible. The reason for this is simply that the read and write access is more expensive compared to any other memory access. In order to achieve the goal of minimizing the number of I/O disk access a divide and conquer strategy was used in this diploma thesis. First the points are sorted along the axis with the widest spread. This leads to a partitioning of the space equal to sorting x,y planes along the z axis. Then the points are read from the sorted stream and processed sequentially. For this a smart concatenation of local operators is used. Any of these operators implement one specific task. The most important operator implements a way to manage the points in a k nearest neighbor graph. This k nearest neighborhood is used by any following operator to estimate normals and other local surface features of the points.

7.1 External Sorting

For this diploma thesis a distribution sort variant was developed, namely the tree based sorting algorithm. This technique has the major advantage that a LRU cache can be used. The LRU cache is managing the buckets of the tree in the availabe RAM. Whenever there is an overflow of the available RAM the LRU cache drops the least recently used bucket. This allows to keep only those buckets in main memory that were recently used. The efficiency in this solution comes from the simple fact that the used point clouds consists of many scan positions. From each of the scan position only a fraction of the whole point cloud was scanned. Those scan positions are read sequentially into the sorting procedure and therefore the points are geometrically close together. This strategy is optimized for point clouds in which the scan positions are close together. If

the scan positions are not in a spatial sense close together then a merge sort can achieve similar results.

7.2 Point Streaming Framework

The improvements section of the Chapter 4 showed 4 possible solutions for the k nearest neighborhood operator. The list based operator was a simple implementation that stores and searches the elements in a doubly linked list. The kd tree based operator in the discussion showed that a kd tree that manages the points will lead to great speed enhancement. The grid based operator manages the values in the x, y plane efficiently if the point cloud was sampled uniformly. The region based operator uses a distribution dependent approximation for the search radius to estimate how large a region has to be in order to get all nearest neighbors.

If the points in a point cloud are nearly uniformly sampled then the grid or the region based operators are preferred. This is because they have nearly no algorithmic overhead. The point sampling in the point clouds that came directly from the laser scans was completely non uniform and therefore the kd tree based operator achieved the goal of finding the nearest neighborhood best.

The kd tree operator is an improvement to the k nearest neighbor operator that is used by Pajarola [PS04] in that it works for point clouds that have non uniform sampling of the points. The developed kd tree operator is similar to the k nearest neighbor operator by Pajarola and Boesch [PB09].

APPENDIX A

UML Class Diagram

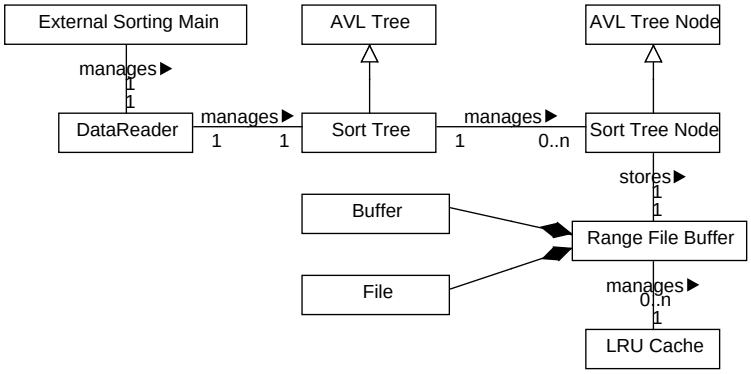

Figure A.1: UML class diagram for the external sorting part of the diploma thesis.

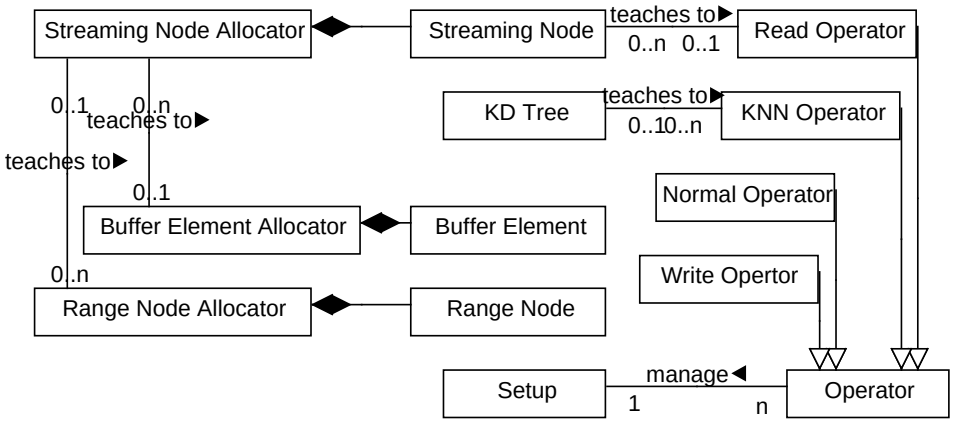

Figure A.2: UML class diagram for the point streaming framework part of the diploma thesis.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AHW89] Hans-Werner Six Andreas Henrich and Peter Widmayer. The lsd tree: spatial access to multidimensional point and non-point objects. Technical report, FernUniversität Hagen, Universität Freiburg, 1989.

[Ben75] John Louis Bentley. Multidimensional binary search trees used for associative searching. Technical report, Stanford University, 1975.

[CDS03] Christian Vogelgsang Carsten Dachsbacher and Marc Stamminger. Sequential point trees. *SIGGRAPH '03 ACM SIGGRAPH 2003 Papers*, 22:657 – 662, 7 2003.

[Cha06] Timothy M. Chan. A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions. Technical report, School of Computer Science, University of Waterloo, 5 2006.

[Ebe09] David Eberly. Least squares fitting of data. Technical report, Geometric Tools, LLC, 2009.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pages 47–57. ACM, 1984.

[HHS92] Tom Duchamp John McDonald Hugues Hoppe, Tony DeRose and Werner Stuetzle. Surface reconstruction from unorganized points. Technical report, Department of Computer Science and Engineering, Department of Mathematics, Department of Statistics, University of Washington, 1992.

[HPG00] J. van Baar H. Pfister, M. Zwicker and M. Gross. Surfels: Surface elements as rendering primitives. Technical report, ETH Zürich, Switzerland, MERL, Cambridge, MA., 2000.

[Jam08] James Reinders (INTEL). on processors, cores and hardware threads, 2008.

[JNS84] H. Hinterberger J. Nievergelt and K. C. SEVCIK. An adaptable, symmetric multikey file structure. *Transactions on Database Systems, Vol. 9, No.1, March 1984*, 99:38–71, 1984.

[JSV07] Hanan Samet Jagan Sankaranarayanan and Amitabh Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers Graphics 31 (2007)*, pages 157–174, 2007.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Longman, Amsterdam, 10 1998.

[LC87] William E. Lorenson and Harvay E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. Technical report, General Electric Company, Corporate Reasearch and Development, 1987.

[LRZ02] H. Pfister Liu Ren and M. Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. Technical report, Carnegie Mellon University, Pittsburgh, ETH Zürich, Switzerland, MERL, Cambridge, MA., 2002.

[LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, Computer Science Department, University of North Carolina, 1985.

[MAS01] Daniel Cohan-Or Shachar Fleishman David Levin Marc Alexa, Johannes Behr and Claudio T. Silva. Point set surfaces. *Vis '01 Proceedings of the conference on Visualization 01*, 99:21–28, 2001.

[MAS03] Daniel Cohen-Or Shachar Fleishman David Levin Marc Alexa, Johannes Behr and Claudio T. Silva. Computing and rendering point set surfaces. Technical report, TU Darmstadt, ZGDV Darmstadt, Tel Aviv University, ATT Labs, 2003.

[MBH07] Randal Burns Matthew Bolitho, Michael Kazhdan and Hugues Hoppe. Multilevel streaming for out-of-core surface reconstruction. *Eurographics Symposium on Geometry Processing 2007*, pages 69–78, 2007.

[MBK05] Matthias Zwicker Mario Botsch, Alexander Hornung and Leif Kobbelt. High-quality surface splatting on todays gpus. Technical report, Computer Graphics Group, RWTH Aachen Technical University, Computer Graphics Group, Massachusetts Institute of Technology, 2005.

[MC08] P. Kumar M. Connor. Parallel construction of k-nearest neighbor graphs for point clouds. Technical report, Department of Computer Science, Florida State University, 2008.

[MN03] Niloy J. Mitra and An Nguyen. Estimating surface normals in noisy point cloud data. Technical report, Stanford Graphics Laboratory, Stanford University, 2003.

[Moo91] Andrew W. Moore. An intoductory tutorial on kd-trees. Technical report, Carnegie Mellon University, 1991.

[Mou10] David M. Mount. Ann programming manual. Technical report, Department of Computer Science and Institut of Advanced Computer Studies, University of Maryland, College Part, Maryland, 2010.

[MZG02] Oliver Knoll Matthias Zwicker, Mark Pauly and Markus Gross. Pointshop3d: An interactive system for point-based surface editing. Technical report, ETH Zürich, 2002.

[PB09] Renato Pajarola and Jonas Boesch. Flexible configurable stream processing of point data. Technical report, Visualization and Multimedia Lab, University of Zürich, 2009.

[PS04] Renato Pajarola and Miguel Sainz. Stream-processing point data. Technical report, Department of Computer Science, University of California, 3 2004.

[RJ96] Rafael Lins Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 7 1996.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. Technical report, Stanford University, 2000.

[Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[Sch09] Claus Scheiblauer. Domitilla catacomb walkthrough dealing with more than 1 billion points. Technical report, Department of Computer Graphics, Vienna University of Technology, 2009.

[Sed92] Robert Sedgewick. *Algorithmen in C++*. Addison-Wesley, 1992.

[TO96] P. Widmayer T. Ottmann. *Algorithmen und Datenstrukturen*. Spektrum, Akademischer Verlag, 1996.

[TWG04] R. Keiser-S. Heinzle S. Scandella T. Weyrich, M. Pauly and M. Gross. Post-processing of scanned 3d surface data. *Proceedings of Eurographics Symposium on Point-Based Graphics 2004*, pages 85–94, 2004.

[WHP02] William T. Vetterling Brian P. Flannery William H. Press, Saul A. Teukolsky. *Numerical Recipes in C++, The Art of Scientific Computing*. Cambride, University Press, 2002.

[WS09] Michael Wimmer and Claus Scheiblauer. Instant points. Technical report, Department of Computer Graphics, Vienna University of Technology, 2009.