Masterarbeit am Institut für Mathematik der Freien Universität Berlin,

Arbeitsgruppe Mathematical Geometry Processing

Neighborhood Computation of Point Set Surfaces

Martin Skrodzki
Matrikelnummer: 4707622
martin.skrodzki@fu-berlin.de
Betreuer: Prof. Dr. Konrad Polthier
Eingereicht bei: Prof. Dr. Konrad Polthier

Berlin, 12. Dezember 2014

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Datum: 12. Dezember 2014

Name: Martin Skrodzki

Unterschrift:

Contents

Chapter 1

Introduction

In this thesis we will present data structures for efficient neighborhood computation of point set surfaces. Given data structures will be tested within a smoothing application implemented in JavaView [Pol+].
During the last years, both 3D scanners and printers became very affordable. Therefore the range of applications got broader. Techniques of 3D printing are now used e.g. in medical applications [Ren+10], [Leu+05] and material sciences [Lam+02]. Furthermore methods from 3D scanning are applied in e.g. face recognition [BF05], traffic accident analysis [Buc+07], as well as in art related or archaeological settings [Lev+00]. A 3D scanner is shown in Figure 1.1.

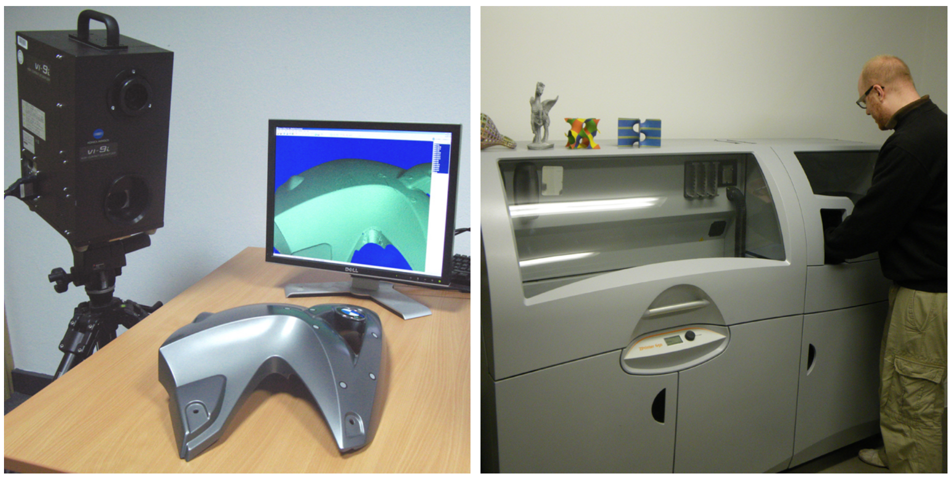


Figure 1.1: A 3D Scanner and a 3D printer. Picture taken from [Pro].

From the scanning process, a 3D computer model of the scanned object is obtained. This model is given by a set of points representing the original

object. If the scanned objects has e.g reflecting surfaces, or is not lighted correctly, the resulting sample points will be noised. Therefore when scanning perfectly flat objects, the resulting model on the computer might still show a rough structure. To not have the noise interfere in the printing process or in a rendering on the computer, the scanned surfaced is usually smoothed.
A common approach to smoothing techniques is to compute a mesh on the point samples obtained from the scanning process. This mesh is then used to smooth the point set, see e.g. [JDD03], [VMM99]. Although this approach is well studied it has one obvious disadvantage: The first step will always be the computation of a mesh. To overcome this disadvantage, several techniques have been developed which work on a point set only and do not require a mesh. For an overview see [GP11].
Not using a mesh does come with a prize. A mesh already encodes neighborhood information, which we will see to be very important for smoothing procedures. When using a solely point-based technique, neighborhoods are not available a priori. This thesis aims at presenting methods for efficient computation of neighborhoods in the point-based setting.

1.1 Overview

As stated above the main goal of this thesis is to present ways to efficiently compute neighborhoods in point sets. In order to emphasize the importance of neighborhoods within smoothing procedures, in Chapter 2 we will present the general idea of point cloud smoothing. The chapter will be mostly based on [LP05]. For a start tools of differential geometry are presented that are used for smoothing in a continuous setting. In order to apply these tools on the point-based approach, we introduce discrete versions of them. Furthermore the chapter introduces the concept of anisotropic smoothing for feature detection.
Having set the theoretical background of smoothing, in Chapter 3 we will give a first general idea of neighborhood computation in point clouds. This idea will lead us to the need for data structures. The field of computational geometry developed several structures suitable for our purposes. Namely we will present Kd-Trees and their implementation in Chapter 3, while two more data structures will be presented in Chapter 4. Although we motivated the need for neighborhood computation from 3D applications, our data structures will be able to handle general d-dimensional points. This is mainly to be able to use them in later, higher-dimensional applications as the theory of manifolds.

The general idea of Nearest Neighbor search will be made more explicit in Chapter 5. We give an implementation of nearest neighbor search within the JavaView [Pol+] framework using Kd-trees. Since our implementations will make heavy use of the median of a point set, in Chapter 6 we will consider three different algorithms of how to determine the median of an unordered sequence.
All implementations are included in a Java program which will be presented in Chapter 7. This is mainly done since the presented program will be used to obtain the computational results given in Chapter 8. All different strategies given in the previous chapters are then evaluated and advice for practical applications is given. Finally we give a conclusion and suggestions for further research in Chapter 9.
This thesis started as the project "Orthogonal Range Searching" [SS] in the course "Scientific Visualization", given by Konrad Polthier. We will also present results from this project.

1.2 Notation

Throughout the thesis we will make use of the following notational conventions.

- For $a \in \mathbb{N}$, by $[a]$ we denote the set $\{1, \dots, a\}$.
- Any vector v is a column vector. We denote its transpose by v^T.
- If not stated otherwise, $\|.\|$ and $\langle ., . \rangle$ denote the standard norm and scalar product in the ambient Euclidean space.

Chapter 2

Point Cloud Smoothing

To both fix a theoretical basis and suggest practical applications of nearest neighbor search on point sets, in this chapter we will present [LP05]. In our presentation we will generally follow the setup of the paper, but divert from it when it seems beneficial for a deeper understanding of the given concepts. The general outline will be a first review of the point based model in Section 2.1, including the definition of a discrete tangential space and isotropic fairing in the discrete setting. We then in Section 2.2 turn to the curvature(s) of point sets and approximate the notion of directional curvature from differential geometry within the discrete setting. In the last Section 2.3 the computed curvatures are used to define an anisotropic Laplacian for corresponding anisotropic fairing.

2.1 Review of the Point Based Model

Given a smooth surface S we will consider point samples taken from S in a sufficiently dense way to reflect the structure of the surface. In [ABK98] it was proven that from such a point set, the surface can be reconstructed in terms of its topological features and that for increasing sample size, the sampled surface converges to S. We assume that sample points sufficiently close together are distributed nearby the tangent plane of S. Considering Principal Component Analysis e.g. [Han10], it is not surprising that an approximation of the tangent plane of S can be derived from the sample points by the covariance matrix of neighboring points.
At first we present how several objects from differential geometry are translated to the point based model. That is, for any sample point, we need a definition of tangent space at this point. This will be done via a minimum least squares expression on the neighborhood of the point. Using this tan-

gent space, we will be able to transfer the techniques of mean curvature flows from triangulated spaces to point clouds.
The model presented here is very similar to the model used in [Pau+02], where a linear approximation of the tangent plane is done almost in the same way. However, the minimum least square expression is set up slightly different. In [Ale+03] and [Lev04] also linear approximations of a tangent space are computed via a minimum least squares technique. But they use a second step to define an implicit surface from the linear approximations, which will not be done here since the linear approximation is sufficient.
In the following, we assume that a smooth surface S, embedded in $\mathbb{R}^3$, is sampled by n points. The set of samples will be denoted by

$$P = \{p_\iota \mid 1 \leq \iota \leq n\} \tag{2.1}$$

and we assume that P describes S in the sense, that the density of P is high enough such that all features from S can be recovered from P. In particular, each p_ι is given by its three real-valued coordinates.

2.1.1 Neighborhoods

The main idea of the point based model is to perform all computations on neighborhoods induced by the Euclidean notion of vicinity rather than on combinatorial neighborhoods as in meshes. For a fine sample and a small Euclidean neighborhood, both notions will be similar. Different approaches on how to set up the neighborhood are presented in [FR01]. In certain aspects, we will follow their third presented method. In [Pau+02] all points from P are considered in the moving least squares method and are weighted according to their Euclidean distance. The authors introduce the notation $\tilde{N}_k(p_\iota)$ for the k nearest neighbors of p_ι relative to Euclidean distance.
We will consider an ε-k-neighborhood $N_k^\varepsilon(p_\iota)$ of a sample point p_ι, that is an intersection of the sample points contained in an ε-ball centered at p_ι and the k sample points closest to p_ι. In particular this means that $N_k^\varepsilon(p_\iota)$ must not necessarily contain k points. However, in the following we will assume that k does denote the size of the neighborhood. Since the parameters ε and k will be globally set, see Chapter 7, we will use the notation $N_k(p_\iota)$ for the neighborhood.

2.1.2 Tangent Spaces

The tangent spaces presented in this section will be approximation to the tangent spaces of the smooth surface S in a twofold sense. First, any tangent space that we derive from the point set P will be an approximation because of the noise on the point set. Second, our tangent planes will not necessarily contain the point of tangency. However, we will see that our concept of a discrete tangent space still converges to the smooth tangent space on S. Given a point p_ι of the point set and its neighborhood $N_k(p_\iota)$, we approximate a tangent space T_ι by minimizing

$$E(n,r) = \sum_{x\in N_k(p_\iota)} (\langle x-b,n\rangle - r)^2, \tag{2.2}$$

where $b\in\mathbb{R}^3$ is any point, n denotes the normal vector of T_ι and r is the distance of T_ι to b. We would now like to find a certain point b such that (2.2) simplifies. Therefore let

$$\bar{b} = \sum_{x\in N_k(p_\iota)} \frac{x}{k} \tag{2.3}$$

denote the barycenter of the neighborhood $N_k(p_\iota)$. Setting $b=\bar{b}$ in (2.2), we see

$$\begin{aligned}
0 &\leq E(n,r)\\
&= \sum_{x\in N_k(p_\iota)} (\langle x-\bar{b},n\rangle - r)^2\\
&= \sum_{x\in N_k(p_\iota)} \left(\langle x-\bar{b},n\rangle^2 - 2r\langle x-\bar{b},n\rangle + r^2\right)\\
&= \sum_{x\in N_k(p_\iota)} \langle x-\bar{b},n\rangle^2 - 2r\cdot \sum_{x\in N_k(p_\iota)} \langle x-\bar{b},n\rangle + \sum_{x\in N_k(p_\iota)} r^2\\
&= \sum_{x\in N_k(p_\iota)} \langle x-\bar{b},n\rangle^2 - 2r\left\langle \sum_{x\in N_k(p_\iota)} (x-\bar{b}), n\right\rangle + k\cdot r^2\\
&= \sum_{x\in N_k(p_\iota)} \langle x-\bar{b},n\rangle^2 - 2r\left\langle \underbrace{\sum_{x\in N_k(p_\iota)} x - \sum_{x\in N_k(p_\iota)} \bar{b}}_{=k\cdot\bar{b}-k\cdot\bar{b}=0}, n\right\rangle + k\cdot r^2\\
&= \sum_{x\in N_k(p_\iota)} \langle x-\bar{b},n\rangle^2 + k\cdot r^2.
\end{aligned}$$

Since the first summands do not depend on r, to minimize the expression $E(n,r)$ is to set $r=0$. That is, the barycenter $\bar{b}$ of $N_k(p_\iota)$ is necessarily a point in any minimizing plane. Therefore we can alter the minimum least square expression (2.2) to the following

$$E(n) := \sum_{x \in N_k(p_\iota)} \langle x - \overline{b}, n \rangle^2. \tag{2.4}$$

Compare this to the usual definition of a tangent plane, e.g. [Bär10]

Definition 1. *Let $S \subset \mathbb{R}^3$ be a regular surface, $p \in S$, then*

$$T_pS = \{T \in \mathbb{R}^3 \mid \exists \varepsilon > 0,\ \gamma : (-\varepsilon, \varepsilon) \to S,\ \gamma \in \mathcal{C}^\infty,\ \gamma(0) = p,\ \gamma'(0) = T\} \tag{2.5}$$

is called the ***tangential plane*** *of S in p.*

For the sake of simplicity in the smooth case the tangential plane is usually seen to be placed at the point p, i.e. one considers the affine tangential plane $T_pS + p$, see Figure 2.1.

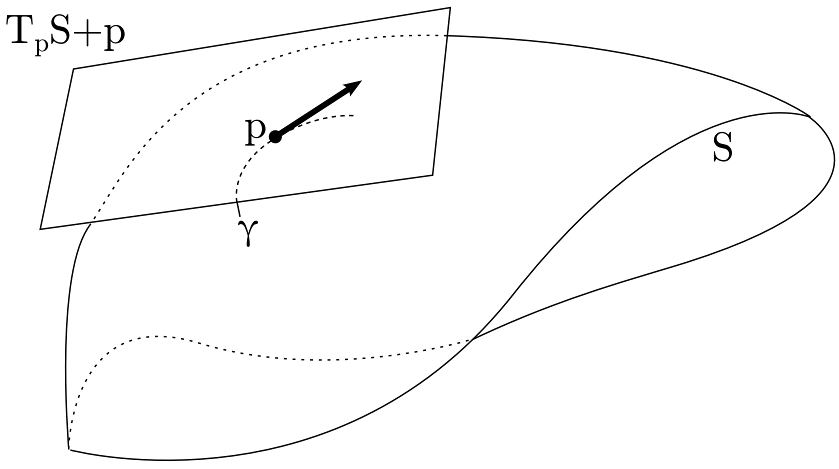


Figure 2.1: The affine tangential plane $T_pS + p$ on a regular surface S according to Definition 1.

Now compare Figure 2.1, showing a smooth tangent plane at $p \in S$, to Figure 2.2, showing a discrete approximation of a tangent plane at $p_\iota \in P$. The most notable difference here is that the point p_ι does not lie in the tangent plane T_ι, but the barycenter $\overline{b}$ of the neighborhood $N_k(p_\iota)$ does.
If we consider the limit of the sample density δ (see Section 2.2.3) and the

number of neighbors k both to infinity, then for every γ as in Definition 1 and for every $\epsilon > 0$, there are a necessary density δ and a k such that P contains two points p_1 and p_2 that have distance less than ϵ to the endpoints of γ. Therefore, in the limit, the tangential plane as defined via (2.4) coincides with the tangential plane from Definition 1.

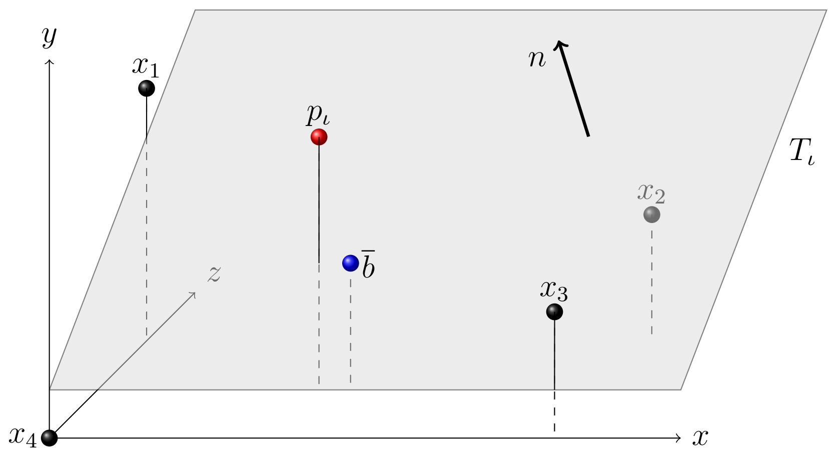


Figure 2.2: Given a point p_ι and its neighborhood $N_4(p) = \{x_1, \ldots, x_4\}$ with the corresponding barycenter $\overline{b}$. The tangent plane T_ι given by (2.4) includes $\overline{b}$ and has normal vector n. The points x_2 and x_4 lie below the plane, while x_1, x_3, and in particular p_ι lie above the plane.

Following [Han10], we define the covariance matrix of a given set of points to be the following.

Definition 2. *Given k vectors $x_1, \ldots, x_k$ of dimension d by $x_i = (x_{i1}, \ldots, x_{id})^T$ and their arithmetic mean $\overline{x} = \sum_{i=1}^{k} \frac{x_i}{k}$ with $\overline{x} = (\overline{x}_1, \ldots, \overline{x}_d)^T$, the* ***empirical covariance*** *of the jth and ℓth coordinate is given by*

$$s_{j\ell} := \sum_{i=1}^{k} (x_{ij} - \overline{x}_j)(x_{i\ell} - \overline{x}_\ell). \tag{2.6}$$

The matrix $S = (s_{j\ell})_{j,\ell \in [d]}$ is called the ***empirical variance-covariance matrix****.*

Since we will not deal with any other sort of covariance here, we will just refer to this construction by covariance matrix. As stated in [Han10], the covariance matrix can be computed using the following identity.

Lemma 1. *Given k vectors $x_1, \dots, x_k$ of dimension d by $x_i = (x_{i1}, \dots, x_{id})^T$ and their arithmetic mean $\overline{x} = \sum_{i=1}^{k} \frac{x_i}{k}$ with $\overline{x} = (\overline{x}_1, \dots, \overline{x}_d)^T$, the following identity holds*

$$(s_{j\ell})_{j,\ell \in [d]} = \sum_{i=1}^{k} (x_i - \overline{x})(x_i - \overline{x})^T. \tag{2.7}$$

Proof. Expanding one of the summands of the statement, we obtain

$$\begin{aligned}(x_i - \overline{x})(x_i - \overline{x})^T &= \begin{pmatrix} x_{i1} - \overline{x}_1 \\ \vdots \\ x_{id} - \overline{x}_d \end{pmatrix} \cdot (x_{i1} - \overline{x}_1, \dots, x_{id} - \overline{x}_d)^T \\ &= \begin{pmatrix} (x_{i1} - \overline{x}_1)(x_{i1} - \overline{x}_1) & \dots & (x_{i1} - \overline{x}_1)(x_{id} - \overline{x}_d) \\ \vdots & \ddots & \vdots \\ (x_{id} - \overline{x}_d)(x_{i1} - \overline{x}_1) & \dots & (x_{id} - \overline{x}_d)(x_{id} - \overline{x}_d) \end{pmatrix}\end{aligned}$$

Summing up these matrices and considering the entry in column j and row ℓ it is

$$\left(\sum_{i=1}^{k} (x_i - \overline{x})(x_i - \overline{x})^T \right)_{j\ell} = \sum_{i=1}^{k} (x_{ij} - \overline{x}_j)(x_{i\ell} - \overline{x}_\ell) = s_{j\ell}.$$

□

Now given $N_k(p_\iota) = \{x_1, \dots, x_k\}$ the neighborhood of p_ι and its barycenter $\overline{b}$, utilizing Lemma 1, we denote

$$M_\iota := \sum_{i=1}^{k} (x_i - \overline{b})(x_i - \overline{b})^T \tag{2.8}$$

the covariance matrix of p_ι.

Theorem 1. *Considering the minimum least squares expression (2.4) and the covariance matrix M_i of p_ι as given in (2.8), the following identity holds:*

$$E(n) = n^T \cdot M_\iota \cdot n. \tag{2.9}$$

In other words Theorem 1 states that any minimal vector n of $E(n)$ is also a minimum of the quadratic form $n^T \cdot M_\iota \cdot n$.

Proof. In our application we will only need the theorem in its three-dimensional application. However, we will prove it for the d-dimensional case here. In the proof we will use the following short-hand notation

$$s(i,u,k) := \frac{(k-1)x_{iu}}{k} - \sum_{j\in[k]\setminus\{i\}} \frac{x_{ju}}{k} = x_{iu} - \bar{b}_u.$$

Using this and denoting $N_k(p_\iota) = \{x_1,\dots,x_k\}$ as well as $n = (n_1,\dots,n_d)^T$, we can establish

$$\begin{aligned}
E(n) &= \sum_{x\in N_k(p_\iota)} \langle x - \bar{b}, n\rangle^2 \\
&= \sum_{i=1}^{k} \langle x_i - \bar{b}, n\rangle^2 \\
&= \sum_{i=1}^{k} \left\langle \frac{(k-1)x_i}{k} - \sum_{j\in[k]\setminus\{i\}} \frac{x_j}{k}, n \right\rangle^2 \\
&= \sum_{i=1}^{k} \left\langle \begin{pmatrix} s(i,1,k) \\ \vdots \\ s(i,d,k) \end{pmatrix}, \begin{pmatrix} n_1 \\ \vdots \\ n_d \end{pmatrix} \right\rangle^2 \\
&= \sum_{i=1}^{k} \left(\sum_{\ell=1}^{d} s(i,\ell,k) \cdot n_\ell \right)^2 \\
&= \sum_{i=1}^{k} \sum_{u,v\in[d]} s(i,u,k)\cdot n_u \cdot s(i,v,k) \cdot n_v \\
&= \sum_{u,v\in[d]} n_u \cdot n_v \cdot \sum_{i=1}^{k} s(i,u,k)\cdot s(i,v,k) \\
&= \sum_{u\in[d]} n_u \cdot \sum_{v\in[d]} n_v \sum_{i=1}^{k} s(i,u,k)\cdot s(i,v,k) \\
&= n^T \cdot \begin{pmatrix} \sum_{v\in[d]} n_v \sum_{i=1}^{k} s(i,1,k)\cdot s(i,v,k) \\ \vdots \\ \sum_{v\in[d]} n_v \sum_{i=1}^{k} s(i,d,k)\cdot s(i,v,k) \end{pmatrix}
\end{aligned}$$

$$
\begin{aligned}
&= n^T \cdot \begin{pmatrix} \sum_{i=1}^k s(i,1,k)\cdot s(i,1,k) & \dots & \sum_{i=1}^k s(i,1,k)\cdot s(i,d,k) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^k s(i,d,k)\cdot s(i,1,k) & \dots & \sum_{i=1}^k s(i,d,k)\cdot s(i,d,k) \end{pmatrix} \cdot n \\
&= n^T \cdot \begin{pmatrix} \sum_{i=1}^k (x_{i1}-\overline{b}_1)(x_{i1}-\overline{b}_1) & \dots & \sum_{i=1}^k (x_{id}-\overline{b}_d)(x_{i1}-\overline{b}_1) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^k (x_{i1}-\overline{b}_1)(x_{id}-\overline{b}_d) & \dots & \sum_{i=1}^k (x_{id}-\overline{b}_d)(x_{id}-\overline{b}_d) \end{pmatrix} \cdot n \\
&= n^T \cdot M_\iota \cdot n.
\end{aligned}
$$

□

We now want to use the equality established by Theorem 1 to find an alternative way of computing a minimum to expression (2.4). The following lemma provides us with the necessary tools.

Lemma 2. *Given a real, symmetric matrix $A \in \mathbb{R}^{d\times d}$, with smallest eigenvalue λ_0, then*

$$\min_{n^Tn=1} n^TAn = \lambda_0.$$

Proof. As for Theorem 1, we only need the three-dimensional case of this lemma in our setting. Nonetheless, we prove its d-dimensional version.
Let $A = UDU^T$ be the eigendecomposition of the matrix A, with D a diagonal matrix having the eigenvalues of A as entries. We get

$$
\begin{aligned}
\min_{n^Tn=1} n^TAn = \min_{u^Tu=1} u^TDu &= \min_{u^Tu=1} u^T \cdot \begin{pmatrix} \lambda_1 u_1 \\ \vdots \\ \lambda_d u_d \end{pmatrix} \\
= \min_{u^Tu=1} \sum_{i=1}^d \lambda_i \cdot u_i^2 &\geq \min_{u^Tu=1} \sum_{i=1}^d \lambda_0 \cdot u_i^2 \\
= \lambda_0 \cdot \min_{u^Tu=1} \sum_{i=1}^d u_i^2 &= \lambda_0 \cdot \min_{u^Tu=1} u^Tu = \lambda_0.
\end{aligned}
$$

since λ_0 is the smallest eigenvalue of A. Picking $n = v_0$ a unit-length eigenvector to the eigenvalue λ_0, we establish

$$\min_{n^Tn=1} n^TAn = v_o^TAv_0 = \lambda_0 v_0^T v_0 = \lambda_0.$$

□

Using Lemma 1 in the following we denote:

$$n_\iota := \text{ unit length eigenvector to the smallest eigenvalue of } M_\iota. \tag{2.10}$$

It will define the approximation of the tangent space at p_ι.
Both [Ale+03] and [Lev04] at this point introduce some higher order projection. They also approximate a tangent plane by a minimum least squares equation, but then approximate the smooth surface S by polynomials that are iteratively refined. We will not get into this here, since for our purposes the initial approximation of the tangent plane is sufficient.

2.1.3 Isotropic Gaussian Fairing

We will now present the method of isotropic Gaussian fairing using the Laplacian. As in Differential Geometry, the Laplacian will be thought of as the composition of the div and the ∇ operator. While these can be defined continuously, here we will have to work with their discrete analogues, just as we did in Section 2.1.2 concerning tangent spaces.
If we consider a sample point p_ι, its neighborhood $N_k(p_\iota) = \{x_1, \ldots, x_k\}$, and a function $f : \mathbb{R}^3 \to \mathbb{R}$, then the discrete version $(\nabla)|_{p_\iota}$ of the gradient, for $c_{\iota j} := p_\iota - x_j$, is given by

$$(\nabla_{x_j})|_{p_\iota}(f) = (f(p_\iota) - f(x_j))c_{\iota j}, \tag{2.11}$$

and

$$(\nabla)|_{p_\iota}(f) = \sum_{j=1}^{k}(f(p_\iota) - f(x_j))c_{\iota j}. \tag{2.12}$$

By introducing a factor $1/k$, expression (2.12) would become independent of the size of the neighborhood of p_ι. Just as the regular gradient

$$f : \mathbb{R}^n \to \mathbb{R}, \quad (u_1, \ldots, u_n) \mapsto f(u_1, \ldots, u_n), \quad \nabla f = \begin{pmatrix} \frac{\partial f}{\partial u_1} \\ \vdots \\ \frac{\partial f}{\partial u_n} \end{pmatrix}, \tag{2.13}$$

the discrete version as given in (2.12) points in the (approximated) direction of largest increase of the function. We will illustrate this concept of the $(\nabla_{x_j})|_{p_\iota}$ on a function $f : \mathbb{R}^2 \to \mathbb{R}$ in the Figure 2.3.
The div operator at p_ι with neighborhood $N_k(p_\iota) = \{x_1, \ldots, x_k\}$ will be interpreted in the following way. Consider the space $\mathbb{R}^k$, where each x_j is

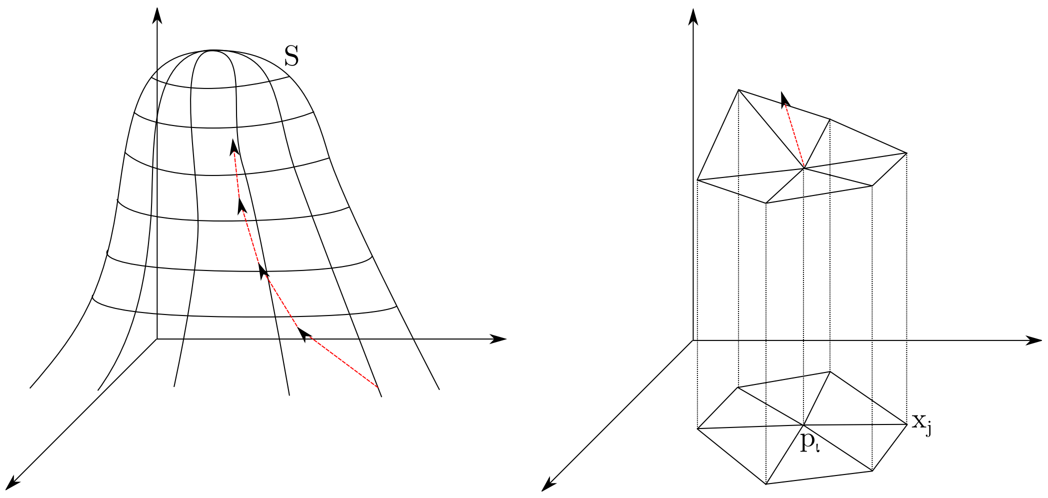


Figure 2.3: On the left the usual smooth ∇ in red on a smooth surface S. On the right the approximation $(\nabla)|_{p_\iota}$ in red on a point set.

identified with an element e_j of an orthonormal basis. Any vector $\tilde{v} \in \mathbb{R}^k$ can be mapped to $\mathbb{R}^3$ by

$$\pi : \tilde{v} = \sum_{j=1}^{k} v_j e_j \;\mapsto\; \sum_{j=1}^{k} v_j c_{\iota j} = v, \tag{2.14}$$

with $c_{\iota j} = p_\iota - x_j$ as above. Then the divergence $\operatorname{div}|_{p_\iota}$ at p_ι in direction v is given by

$$\operatorname{div} v|_{p_\iota} = \sum_{j=1}^{k} \langle \tilde{v}, e_j \rangle_{\mathbb{R}^k} = \sum_{j=1}^{k} v_j, \tag{2.15}$$

where the inner product $\langle ., . \rangle_{\mathbb{R}^k}$ is the euclidean inner product on $\mathbb{R}^k$, where the c_{ij} are identified with an orthonormal basis. Note that div is only well-defined on the space $\mathbb{R}^k$, where $\tilde{v}$ can be uniquely decomposed in a linear composition of the e_j. Since the map π is in general neither injective nor surjective div is in $\mathbb{R}^3$ only defined on the image of π and is in general not well-defined in $\mathbb{R}^3$. In fact, as soon as $|N_k(p_\iota)| \geq 4$, the decomposition of v into the $c_{\iota j}$ is not unique any more. However, note that the discrete gradient as defined in (2.12) is by definition in the image of π and hence the div operator can be applied to it. Now compare this to the usual definition of div, given as

$$F = (F_1, \ldots, F_n) : \mathbb{R}^n \to \mathbb{R}^n, \quad (u_1, \ldots, u_n) \mapsto F_i(u_1, \ldots, u_n) \;\forall i = 1, \ldots, n,$$

$$\operatorname{div} F = \sum_{i=1}^{n} \frac{\partial}{\partial x_i} F_i.$$

A common interpretation of the div operator is that it measures the outgoing flow at a point. That is, $\operatorname{div} F$ is the sum of flows in the directions x_i. In a certain sense the discrete operator $\operatorname{div} v|_{p_\iota}$ also measures the outgoing flow, but the flow in a direction v, which needs to be approximated by the information provided with the neighborhood. See Figure 2.4 for an illustration.

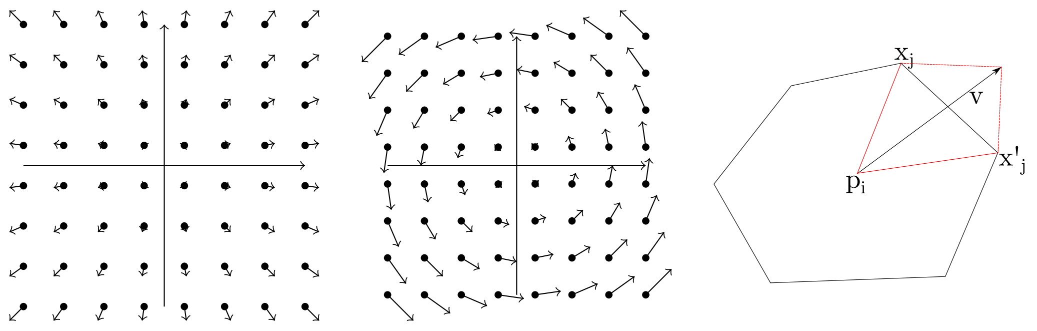


Figure 2.4: On the left a vector field $F : (x, y) \mapsto (2x, 2y)$, with $\operatorname{div} F = 2 + 2 = 4$. In the middle a vector field $G : (x, y) \mapsto (-y, x)$ with $\operatorname{div} G = 0$. On the right a point p_ι and a vector v that is expressed in terms of the neighborhood of p_ι, defining the divergence, i.e. the flow of p_ι in direction v.

As stated above, we now define the Laplace operator Δ to be the composition of the ∇ operator as defined in (2.12) and the div operator as defined in (2.15) and obtain the isotropic Laplacian $\Delta|_{p_\iota}$ to be

$$\Delta|_{p_\iota} f = \sum_{x_j \in N_k(p_\iota)} (f(p_\iota) - f(x_j)). \tag{2.16}$$

As mentioned above, the discrete ∇ as given in (2.12) could be given a factor $1/|N_k(p_\iota)$. If we did introduce this factor, we would arrive at the discrete Laplacian as given in [Des+99] by

$$\Delta|_{p_\iota} = \frac{1}{|N_k(p_\iota)|} \sum_{x_j \in N_\iota} x_j - p_\iota.$$

However, this is the same operator as given in [Pau+02] for the choice of $\omega_j = 1$ for all $x_j \in N_k(p_\iota)$ and $\omega_j = 0$ otherwise, which by the authors is called the uniform umbrella:

$$\Delta|_{p_\iota} = \frac{1}{\Omega} \sum_{x_j \in N_\iota} \omega_j (p_\iota - x_j), \qquad \Omega = \sum_{x_j \in N_\iota} \omega_j.$$

Using the same weights as given above, [Her12] also defines the Laplacian as an umbrella operator in this way. We see that there are several ways to set up a discrete Laplacian and that the term umbrella operator is used in several different ways across literature. For this thesis we will use the discrete Laplacian as defined in (2.16).
In [Des+99], the authors present a diffusion process to reduce noise on meshed surfaces. It arises from minimizing the functional of the total curvature of a surface S which in the smooth setting is given by

$$E(S) = \int_S \kappa_1^2 + \kappa_2^2 \, dS,$$

where κ_1, κ_2 are the principal curvatures of S. We will show how to compute κ_1 and κ_2 in our discrete setting in Section 2.2. Nevertheless, we follow [Des+99] in considering a different functional, namely

$$E_{membrane}(X) = \frac{1}{2} \int_\Omega X_{u_1}^2 + X_{u_2}^2 \, du_1 \, du_2,$$

where X denotes a mesh parametrized over Ω. The corresponding variational derivative is

$$L(X) = X_{u_1 u_1} + X_{u_2 u_2},$$

which is the Laplacian. Therefore the diffusion process to eliminate noise is for a given surface S described by the following PDE

$$\frac{\partial S}{\partial t} = \lambda \Delta S. \tag{2.17}$$

It can be solved using explicit Euler integration which gives the following iterative formula

$$S^{n+1} = (Id + \lambda \partial t \Delta) S^n. \tag{2.18}$$

Using this formula the smoothness of the surface can be controlled by the scale parameter λ, the number of iterations and the size of the local neighborhood used in the approximation of Δ. Following [Pau+02], it is recommended in [LP05] to compute the neighborhood of a point p_ι only in the first step and cache it for the following iterations. This not only improves efficiency, but also prevents clustering effects which might arise by tangential drift of sample points.

2.2 Curvatures of Point Sets

We will now present the notions of directional curvature and the Weingarten Map and give approximations of both within the point set setting. The Weingarten map, or shape operator, encodes curvature information of a surface. By approximating the curvature at a point p_ι we will be able to detect features of the original model, present in the point cloud, that are not to be smoothed by our process, e.g. corners or edges.

2.2.1 Directional Curvatures

In Section 2.1.2 we defined a discrete tangent space. Using this definition, we can also approximate the notion of directional curvature from differential geometry. Let $S \subset \mathbb{R}^3$ be an orientable regular surface with a smooth unit normal field N and let $p \in S$. Let $T \in T_pS$ be any unit length vector in the tangent space at p, $\gamma : (-\epsilon, \epsilon) \to S$ be a curve parametrized by arc length with $\gamma(0) = p$ and $\gamma'(0) = T$. Then the directional curvature $\kappa_p(T)$ is defined by $\gamma''(0) = \kappa_p(T)N$.
If we expand $\gamma(s)$ into a Laurent series at 0 up to second order, we obtain

$$\begin{aligned}\gamma(s) &= \gamma(0) + \gamma'(0) \cdot s + \frac{1}{2}\gamma''(0)s^2 + \mathcal{O}(s^3)\\ &= p + T \cdot s + \frac{1}{2}\kappa_p(T)Ns^2 + \mathcal{O}(s^3).\end{aligned}$$

Now we have the equivalences

$$\begin{aligned}&\gamma(s) = p + T \cdot s + \frac{1}{2}\kappa_p(T)Ns^2 + \mathcal{O}(s^3)\\ \Leftrightarrow&\gamma(s) - p = T \cdot s + \frac{1}{2}\kappa_p(T)Ns^2 + \mathcal{O}(s^3)\\ \Leftrightarrow&N^T(\gamma(s) - p) = \underbrace{N^T T}_{=0} \cdot s + \frac{1}{2}\kappa_p(T)\underbrace{N^T N}_{=1} s^2 + \mathcal{O}(s^3)\\ \Leftrightarrow&2N^T(\gamma(s) - p) = \kappa_p(T)s^2 + \mathcal{O}(s^3)\end{aligned}$$

and

$$\begin{aligned}&\gamma(s) = p + T \cdot s + \frac{1}{2}\kappa_p(T)Ns^2 + \mathcal{O}(s^3)\\ \Leftrightarrow&\gamma(s) - p = T \cdot s + \frac{1}{2}\kappa_p(T)Ns^2 + \mathcal{O}(s^3)\\ \Leftrightarrow&\langle\gamma(s) - p, \gamma(s) - p\rangle = \langle T \cdot s + \frac{1}{2}\kappa_p(T)Ns^2 + \mathcal{O}(s^3), T \cdot s + \frac{1}{2}\kappa_p(T)Ns^2 + \mathcal{O}(s^3)\rangle\end{aligned}$$

$$\Leftrightarrow \|\gamma(s)-p\|^2 = s^2\cdot\underbrace{T^TT}_{=1}+s\kappa_p(T)\underbrace{\langle T,N\rangle}_{=0}+\frac{1}{4}\kappa_p^2(T)\underbrace{N^TN}_{=1}s^4+\mathcal{O}(s^3)$$

$$\Leftrightarrow \|\gamma(s)-p\|^2 = s^2+\mathcal{O}(s^3).$$

From the last two equalities we get by division

$$\frac{2N^T(\gamma(s)-p)}{\|\gamma(s)-p\|^2} = \kappa_p(T)+\mathcal{O}(s),$$

which by taking the limit $s\to 0$ can be transfered into the following formula for the directional curvature

$$\kappa_p(T) = \lim_{s\to 0}\frac{2\langle N(p),\gamma(s)-p\rangle}{\|\gamma(s)-p\|^2}. \tag{2.19}$$

Since we defined a tangent space in Section 2.1.2 and a normal by (2.10) at each point p_ι of the point set, we can now define the directional curvature $\kappa_{\iota j}$ at p_ι in direction $x_j\in N_k(p_\iota)$ by

$$\kappa_{\iota j} := \frac{2\langle n_\iota, x_j-p_\iota\rangle}{\|x_j-p_\iota\|^2}. \tag{2.20}$$

Note that in the smooth setting the direction curvature κ_p has the following property.

Theorem 2. *The directional curvature κ_p is a quadratic form and thereby satisfies the identity*

$$\kappa_p(T) = \begin{pmatrix}t_1\\t_2\end{pmatrix}^T\begin{pmatrix}\kappa_p^{11} & \kappa_p^{12}\\ \kappa_p^{21} & \kappa_p^{22}\end{pmatrix}\begin{pmatrix}t_1\\t_2\end{pmatrix}, \tag{2.21}$$

where $T=t_1T_1+t_2T_2$ is a tangent vector to S at p, $\{T_1,T_2\}$ is an orthonormal basis of the tangent space to S at p, $\kappa_p^{11}=\kappa_p(T_1)$, $\kappa_p^{22}=\kappa_p(T_2)$, and $\kappa_p^{12}=\kappa_p^{21}$.

The vectors $\{T_1,T_2\}$ are called principal curvature directions of S if $\kappa_p^{12}=\kappa_p^{21}=0$. The corresponding directional curvatures are the principal curvatures which in the following will be denoted by κ_p^1 and κ_p^2 instead of κ_p^{11} and κ_p^{22}. For a proof of this theorem see for example [Tho79].

2.2.2 Weingarten Map

We will now divert from [LP05] and follow the approach of [Tau95] to compute a point set estimation of the Weingarten map. In differential geometry, the Weingarten map, or the shape operator is a linear map $W_p : T_pS \to T_pS$ defined by

$$W_p(T) = -\nabla_p N(T),$$

where $N(.)$ is the Gauss-map and $T \in T_pS$ some tangential vector. The actual estimate of the Weingarten map will be given in the next section. At first we will derive an alternative version of the Weingarten map by writing it in terms of an integral. Therefore we extend (2.21) to non-tangential direction. Adding the normal vector N to the orthonormal basis $\{T_1, T_2\}$ of $\mathbb{R}^2$, we obtain an orthonormal basis $\{N, T_1, T_2\}$ of $\mathbb{R}^3$. The extension of (2.21) is now given by

$$\kappa_p(T) = \begin{pmatrix} n \\ t_1 \\ t_2 \end{pmatrix}^T \begin{pmatrix} 0 & 0 & 0 \\ 0 & \kappa_p^1 & 0 \\ 0 & 0 & \kappa_p^2 \end{pmatrix} \begin{pmatrix} n \\ t_1 \\ t_2 \end{pmatrix}, \tag{2.22}$$

where $T = nN + t_1T_1 + t_2T_2$ is an arbitrary vector in $\mathbb{R}^3$.
Ultimately we want to approximate the Weingarten map using the point set. To do this, we will first write it as an integral, which can later be approximated by a sum. For $-\pi \leq \theta \leq \pi$ let T_θ be a unit length tangent vector

$$T_\theta = \cos(\theta)T_1 + \sin(\theta)T_2,$$

with $\{T_1, T_2\}$ the orthonormal principal curvature directions of S at p. We define the symmetric matrix

$$W_p := \frac{1}{2\pi} \int_{-\pi}^{\pi} \kappa_p(T_\theta) T_\theta T_\theta^T \, d\theta. \tag{2.23}$$

Since the normal vector N is orthogonal to T_θ, we see that N is an eigenvector of W_p associated to the eigenvalue 0. Using the fact that $\{T_1, T_2, N\}$ is an orthonormal basis of $\mathbb{R}^3$, we can factorize W_p as

$$W_p = T_{12} \begin{pmatrix} w_p^{11} & w_p^{12} \\ w_p^{21} & w_p^{22} \end{pmatrix} T_{12}^T,$$

where $T_{12} = [T_1, T_2]$ is the 3×2 matrix constructed by concatenating the column vectors T_1 and T_2 and $w_p^{12} = w_p^{21}$ because of the symmetry of W_p.

Furthermore, by plugging T_θ into (2.22) we obtain

$$\begin{aligned}\kappa_p(T_\theta) &= \begin{pmatrix} 0 \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix}^T \begin{pmatrix} 0 & 0 & 0 \\ 0 & \kappa_p^1 & 0 \\ 0 & 0 & \kappa_p^2 \end{pmatrix} \begin{pmatrix} 0 \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix} \\ &= \kappa_p^1 \cos(\theta)^2 + \kappa_p^2 \sin(\theta)^2, \end{aligned} \tag{2.24}$$

the Euler formula. Using the decomposition of W_p and the Euler formula we can now compute

$$\begin{aligned} w_p^{12} &= \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} w_p^{11} & w_p^{12} \\ w_p^{21} & w_p^{22} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ &= T_1^T T_{12} \begin{pmatrix} w_p^{11} & w_p^{12} \\ w_p^{21} & w_p^{22} \end{pmatrix} T_{12}^T T_2 \\ &= T_1^T W_p T_2 \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \kappa_p(T_\theta) T_1^T T_\theta T_\theta^T T_2 \, d\theta \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} (\kappa_p^1 \cos^2(\theta) + \kappa_p^2 \sin^2(\theta)) \cdot T_1^T (\cos(\theta) T_1 + \sin(\theta) T_2) \\ &\quad \cdot (\cos(\theta) T_1^T + \sin(\theta) T_2^T) T_2 \, d\theta \\ &= \frac{\kappa_p^1}{2\pi} \int_{-\pi}^{\pi} \cos^3(\theta) \sin(\theta) \, d\theta + \frac{\kappa_p^2}{2\pi} \int_{-\pi}^{\pi} \cos(\theta) \sin^3(\theta) \, d\theta = 0, \end{aligned}$$

where both integrals in the last step are 0, because the integrands are point-symmetric with respect to the origin. But this means that the the other two eigenvectors of W_p, apart from N, are the two principal curvature directions T_1 and T_2. However, the corresponding eigenvalues are not κ_p^1 and κ_p^2, but:

$$\begin{aligned} w_p^{11} &= \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} w_p^{11} & w_p^{12} \\ w_p^{21} & w_p^{22} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= T_1^T T_{12} \begin{pmatrix} w_p^{11} & w_p^{12} \\ w_p^{21} & w_p^{22} \end{pmatrix} T_{12}^T T_1 \\ &= T_1^T W_p T_1 \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \kappa_p(T_\theta) T_1^T T_\theta T_\theta^T T_1 \, d\theta \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} (\kappa_p^1 \cos^2(\theta) + \kappa_p^2 \sin^2(\theta)) \cdot T_1^T (\cos(\theta) T_1 + \sin(\theta) T_2) \\ &\quad \cdot (\cos(\theta) T_1^T + \sin(\theta) T_2^T) T_1 \, d\theta \end{aligned}$$

$$
\begin{aligned}
&= \frac{\kappa_p^1}{2\pi}\int_{-\pi}^{\pi}\cos^4(\theta)\,d\theta + \frac{\kappa_p^2}{2\pi}\int_{-\pi}^{\pi}\cos^2(\theta)\sin^2(\theta)\,d\theta \\
&= \frac{3}{8}\kappa_p^1 + \frac{1}{8}\kappa_p^2.
\end{aligned}
$$

Using a similar computation we find

$$w_p^{22} = T_2^T W_p T_2 = \frac{1}{8}\kappa_p^1 + \frac{3}{8}\kappa_p^2.$$

To summarize, given a matrix W_p as in (2.23), we obtain the principal curvatures at p as functions of the nonzero eigenvalues of W_p by

$$
\begin{aligned}
\kappa_p^1 &= 3w_p^{11} - w_p^{22} \\
\kappa_p^2 &= 3w_p^{22} - w_p^{11}
\end{aligned}
\tag{2.25}
$$

Note that all computations in this section have been performed in the smooth setting. In the next section we will give an estimate of the Weingarten map within the discrete setting.

2.2.3 Estimation of weights and Principal Curvatures

We now translate the integral formula (2.23) into the discrete setting of a point set. For each point $p_\iota \in P$ we approximate the matrix W_ι with a weighted sum over the neighborhood $N_k(p_\iota)$:

$$W_\iota = \frac{1}{|N_k(p_\iota)|}\sum_{x_j \in N_k(p_\iota)} \omega_{\iota j}\kappa_{\iota j}T_{\iota j}T_{\iota j}^T, \tag{2.26}$$

where for each neighbor x_j of p_ι we define $T_{\iota j}$ to be the normalized tangential part of the vector $c_{\iota j} = p_\iota - x_j$, that is we project $c_{\iota j}$ onto the tangent plane $\langle n_\iota \rangle^\perp$ and normalize by

$$T_{\iota j} = \frac{(I - n_\iota n_\iota^T)c_{\iota j}}{\|(I - n_\iota n_\iota^T)c_{\iota j}\|}.$$

Last the weights $\omega_{\iota j}$ have to be determined in such a way that (2.26) approximates (2.23) correctly.
This will be done by considering the tangential parts of the covariance matrix M_ι as defined in (2.8). Just as for the Weingarten map, we express the covariance matrix in terms of an integral

$$M_\iota = \frac{1}{2\pi}\int_{-\pi}^{\pi}\delta_\theta T_\theta T_\theta^T\,d\theta,$$

where $\delta_\theta = \delta_1 \cos(\theta)^2 + \delta_2 \sin(\theta)^2$ is a quadratic form to estimate the density of the point set in direction T_θ. By performing similar computations as above we can determine values δ_1 and δ_2 in terms of the two largest eigenvalues of M_ι. That is, since we defined n_ι to be the eigenvector to the smallest eigenvalue. Utilizing the values δ_1, δ_2 we can define the density in direction $p_\iota - x_j$ in terms of the two largest eigenvectors v_1 and v_2 of M_ι to be

$$\delta_{\iota j} := \delta_1 \langle T_{\iota j}, v_1 \rangle + \delta_2 \langle T_{\iota j}, v_2 \rangle. \tag{2.27}$$

This expression gives us the density for a regular $|N_k(p_\iota)|$-gon of radius 1. In general we do not want points $x_j \in N_k(p_\iota)$ in a very dense region to have a larger influence on a p_ι than those points of the neighborhood from a less dense region. Hence the weight $\omega_{\iota j}$ of a point $x_j \in N_k(p_\iota)$ is given by the density of the sample in direction $(p_\iota - x_j)$ and the distance of x_j to p_ι:

$$\omega_{\iota j} = \frac{1}{\delta_{\iota j} \cdot \|p_\iota - x_j\|}. \tag{2.28}$$

Therefore we finally obtain

$$W_\iota = \frac{1}{|N_k(p_\iota)|} \sum_{x_j \in N_k(p_\iota)} \frac{1}{\delta_{\iota j} \cdot \|p_\iota - x_j\|} \kappa_{ij} T_{\iota j} T_{\iota j}^T. \tag{2.29}$$

It is worth noting that the general approach concerning equation (2.26) is the same in both [LP05] and [Tau95]. However, since [Tau95] works on polygonal meshes, in that setting it is not necessary to include a density estimation, since the density can be computed from the volume and angles of the triangulation. Having no such mesh at hand we have followed [LP05] here in giving a discrete directional density measure. Furthermore note that the eigenvalues of the derived discrete shape operator (2.29) are the principal curvatures and the eigenvectors are the principal curvature directions at p_ι. They will be crucial in the following section when anisotropic fairing is introduced.

2.3 Anisotropic Mean Curvature Flow

Applying (2.18) with the discrete Laplacian as given in (2.16) has the effect outlined in Figure 2.5.
To not loose features of the original point set, such as sharp edges, we now introduce an anisotropic Laplacian Δ^A to smooth the point set. In the continuous case, considering a function $\rho : \mathbb{R}_0^+ \times \Omega \to \mathbb{R}$, this leads to the parabolic problem

$$\frac{\partial}{\partial t}\rho - \operatorname{div}(A(\nabla \rho_\epsilon)\nabla \rho) = f(\rho), \qquad \text{in } \mathbb{R}^+ \times \Omega,$$

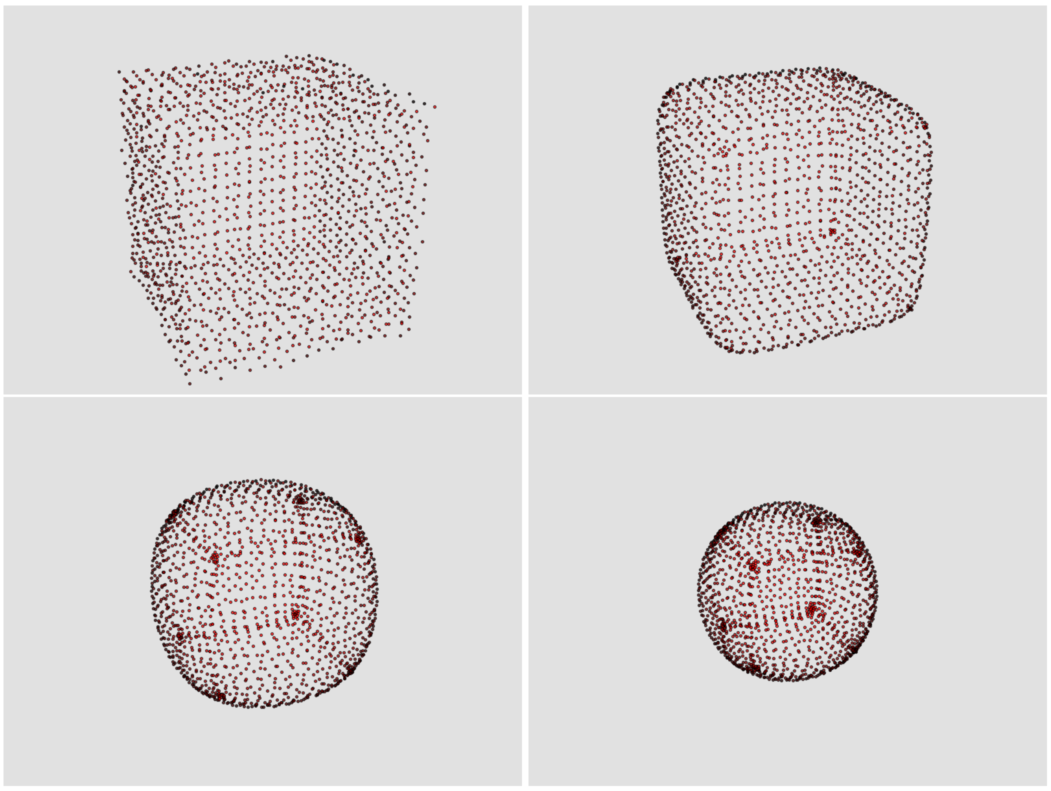


Figure 2.5: Smoothing a noised cube with the the process from (2.18) and the Laplacian as given in (2.16). The picture shows the original noised point set as well as the smoothed point set after 60, 390, and 890 iterations.

$$\begin{aligned} \rho(0,.) &= \rho_0, && \text{on } \Omega, \\ \frac{\partial}{\partial \nu}\rho &= 0, && \text{on } \mathbb{R}^+ \times \partial\Omega \end{aligned}$$

for given initial density $\rho_0 : \Omega \to [0,1]$, see [PR99] or [CDR00]. The authors of [PR99] propose a diffusion coefficient

$$A = G(\|\nabla \rho_\epsilon\|),$$

where $G : \mathbb{R}_0^+ \to \mathbb{R}^+$ is a monotone decreasing function satisfying certain properties. The anisotropic Laplacian reduces to the isotropic version in case of $A \equiv 1$. We now want to mimic the continuous case in our discrete setting and define

$$\Delta^A|_{p_\iota} := \operatorname{div}|_{p_\iota} \circ (A_i \cdot \nabla)|_{p_\iota}, \tag{2.30}$$

where

$$(A_i \cdot \nabla_{x_j})|_{p_\iota}(f) := g_{\iota j} \cdot (f(p_\iota) - f(x_j))(p_\iota - x_j),$$

for all $x_j \in N_k(p_\iota)$ and $g_{\iota j}$ some function mapping to $[0,1] \subset \mathbb{R}$. With this definition, the anisotropic Laplacian behaves different for different functions $g_{\iota j}$. For a threshold λ we can e.g. consider

$$g_{\iota j}^{\text{sharp}} = \begin{cases} 1, & \text{if } |\kappa_{\iota j}| < \lambda, \\ 0, & \text{if } |\kappa_{\iota j}| \geq \lambda; \end{cases} \tag{2.31}$$

$$g_{\iota j}^{\text{cont}} = \begin{cases} 1, & \text{if } |\kappa_{\iota j}| < \lambda, \\ \frac{\lambda^2}{\lambda^2 + 10(\kappa_{\iota j} - \lambda)^2}, & \text{if } |\kappa_{\iota j}| \geq \lambda; \end{cases} . \tag{2.32}$$

Now in both cases the anisotropic smoothing prefers neighbors x_j of p_ι that have curvature $\kappa_{\iota j}$ less than λ. In the case of (2.31) all other neighbors are neglected in the computation, while in case of (2.32) neighbors with curvature greater or equal to λ are only considered to a small extend. That is, if the neighbor x_j lies in a "flat direction" from p_ι its influence on p_ι is higher than the influence of a neighbor at a "steep direction".
As announced at the end of Section 2.2.3 we will now use principal curvatures in the setting of anisotropic smoothing. In [LP05] three possible ways of utilizing principal curvature are given. First, the user defines a parameter called the edge quotient Q. At each point p_ι of the point set a feature at p_ι is to be enhanced, if the quotient q_p of the principal curvatures is less than the chosen parameter, i.e. the condition is

$$q_p := \frac{\kappa^1_{p_\iota}}{\kappa^2_{p_\iota}} < Q. \tag{2.33}$$

A second approach consists of considering the point p_ι as a feature that is to be enhanced, if the larger principal curvature exceeds a threshold K. That is the condition

$$\max\{\kappa^1_{p_\iota}, \kappa^2_{p_\iota}\} > K. \tag{2.34}$$

The third and last approach given qualifies a point p_ι to be a feature of the sampled surface, if there is some directional curvature exceeding a threshold D. That is p_ι is considered a feature to be enhanced if

$$\exists x_j \in N_k(p_\iota) \text{ such that } |\kappa_{\iota j}| > D. \tag{2.35}$$

Note that all three approaches given by (2.33), (2.34), and (2.35) include the necessity of choosing a suitable parameter Q, K, or D, respectively. Considering 2.5 we now present a corresponding series of images in Figure 2.6, except this the anisotropic Laplacian is used. For better visibility of features,

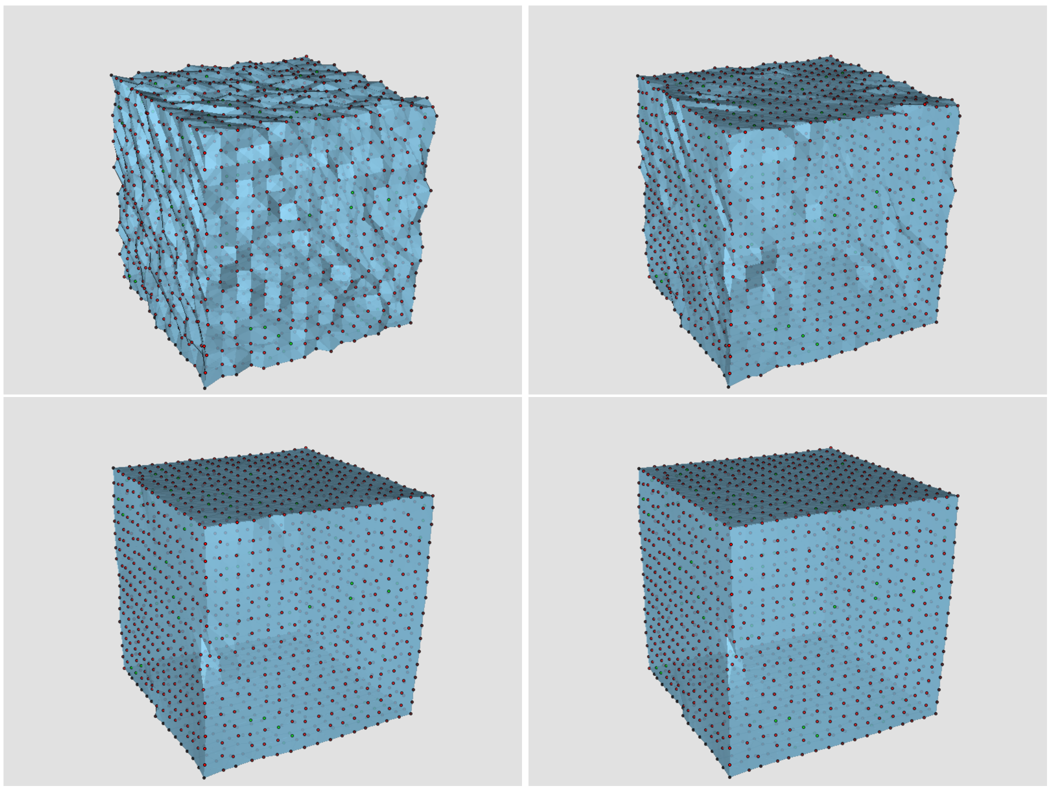


Figure 2.6: Smoothing a noised cube with the the process from (2.18) and the anisotropic Laplacian as given in (2.30). The picture shows the original noised cube as well as the smoothed cube after 30, 80, and 150 iterations. The faces of a triangulation are only shown for better feature visibility and the triangulation is not used in the actual procedure, where only the underlying point set is used.

a triangulated geometry is shown, but the algorithm only acts on the underlying point set.

As a final remark to this section we would like to emphasize the choice of a good neighborhood $N_k(p_\iota)$ for each point $p_\iota \in P$. The importance of a well chosen neighborhood becomes obvious in the following setup. Assume that we have a smooth point sample of an object. Now we can compute the neighborhood on this smooth point sample and store it. We add noise to the point set and obtain a noised sample. When applying the techniques from this section to this noised sample, we use the stored neighborhood of the smooth sample instead of computing a neighborhood on the noised sample. The striking result at this point is that in this admittedly artificial setup, the results are better than in the case where a neighborhood from the noised sample is used. Figure 2.7 from [LP05] illustrates this. Although this might be an artificial setup, it still shows the importance of a well chosen neighborhood and poses the question how to find the ”best” neighborhood from a noised point set.

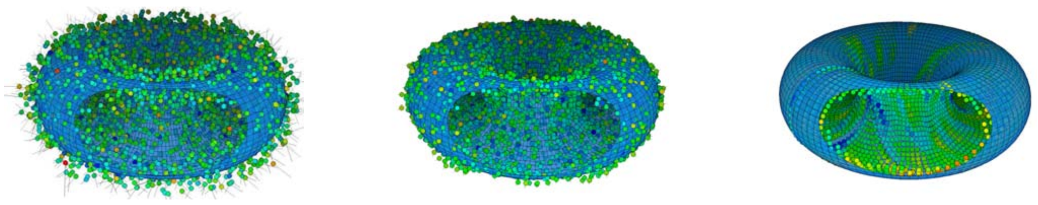


Figure 2.7: A figure taken from [LP05] to illustrate the benefits of using a neighborhood from a smooth point set. From left to right the noised sample points and the smoothed point set after 15 respectively 50 iterations.

Chapter 3

General idea of neighborhood computation and Kd-Trees

In Chapter 2 we set a theoretical basis that depended heavily on the notion of neighborhoods. In this chapter we will in Section 3.1 give an informal description of a procedure for the computation of nearest neighbors. This informal description will motivate the inspection of certain data structures, of which Kd-Trees are presented in Section 3.2 in terms of the underlying theory. Finally we close this chapter with some remarks on the implementation of Kd-Trees in Section 3.3.

3.1 General idea of neighborhood computation

In Chapter 2 all computations did rely on the knowledge of a neighborhood $N_k(p_\iota)$ for a given sample point p_ι. A first naive approach for finding the neighborhood $N_k(p_\iota)$ would be to iterate through all points in the point set $P = \{p_\iota \mid 1 \leq \iota \leq n\}$ and report the k points closest to p_ι. This algorithm has time complexity $c(k) \cdot n$. Therefore, finding the neighborhoods for all n points in P would take time $\mathcal{O}(n^2)$.
In order to obtain a shorter computation time for the neighborhoods, we turn to the concept of **Spatial Indexing**. It is much like the index in the back of a book, which can be used to easily find the page, where a certain notion occurs. Similarly, a spatial database indexes over points or objects in space, providing easy access to them without the necessity to go through the whole database. See [SC03] for an introduction to Spatial Databases and [CZ05] for applications other than those mentioned in this thesis.
In this chapter and in Chapter 4 we will present three tree-like data struc-

tures, where each internal node and each leaf of the tree represents a region of the three-dimensional space. Then the general idea of a fast nearest neighbor computation is outlined in the following algorithm.

Algorithm 1 NearestNeighbor

```
1: procedure NearestNeighbor($D, p_\iota$) //Data structure $D$, point $p_\iota$
2:     Find the region(s) in $D$ storing $p_\iota$.
3:     Add all neighboring regions to a Stack $S$.
4:     while $S$ is not empty do
5:         Pop a region $R$ from $S$.
6:         if $R$ might contain a point closer to $p_\iota$ than the current NN then
7:             Consider all points from $R$ as possible NN to $p_\iota$
8:             Replace the current NN if applicable.
9:             Add all neighboring regions $R'$ of $R$ to $S$.
10:        end if
11:    end while
12: end procedure
```

By using Algorithm 1, we expect that in line 6 certain regions are rejected. Thereby their corresponding neighboring regions will never be examined and in general, to find a nearest neighbor of a point p_ι, will not require to compute the distance to all n points from the point set P. In Section 5.2 we will present an implementation of Algorithm 1.

3.2 Theory of Kd-Trees

In this section we will present the theory behind the data structure of Kd-Trees. Since the main idea of a Kd-Tree will be a generalization of Binary Search to arbitrary dimension, in Section 3.2.1 we will first recall Binary Search. In the same section we turn to one dimensional Range Searches. We use them to illustrate the different concepts of balanced Binary Search Trees and AVL Trees. Having these concepts at hand, in Section 3.2.2 we present the concept of a Kd-Tree. In this section we will generally follow the presentation of [Ber+08], but we divert from it as its seems necessary. In particular we immediately consider a Kd-Tree in arbitrary dimension, while [Ber+08] mostly presents two-dimensional trees. Some texts and illustrations in this section are taken from [SS]. Note that Kd-Trees are only the first of three data structures presented in this thesis. See Chapter 4 for the presentation of two more concepts.

3.2.1 Binary Search and linear Range Search

The concept of Binary Search will be the basis for the following discussion. Hence we will shortly recall it. Given an ordered list of numbers $r_1, \dots, r_n$, Binary Search either finds a given number q amongst the r_i or states that $q \neq r_i$ for all $i = 1, \dots, n$. A pseudo-code adaption of Binary Search is given as Algorithm 2, which gives the index i for a query number q or returns -1 if q is not equal to any of the r_i.

Algorithm 2 Binary Search

```
1: procedure BinarySearch($L := \{r_1, \dots, r_n\}, q$) //ordered $r_i$, query point $q$
2:     if $L$ is empty then
3:         return -1
4:     else
5:         $Median \leftarrow r_{\lfloor n/2 \rfloor}$
6:         if $Median = q$ then
7:             return $\lfloor n/2 \rfloor$
8:         else if $Median > q$ then
9:             return BinarySearch($\{r_1, \dots, r_{\lfloor n/2 \rfloor - 1}\}, q$)
10:        else if $Median < q$ then
11:            return BinarySearch($\{r_{\lfloor n/2 \rfloor + 1}, \dots, r_n\}, q$)
12:        end if
13:    end if
14: end procedure
```

Binary Search on n numbers has a runtime of $\log(n)$ and is furthermore an optimal searching strategy [SW11]. Now Binary Search can also be used to find all numbers $\{r_i \mid a \leq r_i \leq b\}$ for given a, b, by performing Binary Search on $\{r_1, \dots, r_n\}$ twice, with input a and input b and respective output i_a and i_b according to Algorithm 2. Now the numbers greater equal a and less equal b are given by

$$\{r_i \mid a \leq r_i \leq b\} = \{r_i \mid i_a \leq i \leq i_b\}.$$

Instead of an ordered list or an array, one might also make use of the data structure of a balanced Binary Search Tree. While an array or a list need to be sorted in an initial step to be able to use them in Algorithm 2, a balanced Binary Search Tree needs to be built from the numbers r_i. In the balanced Binary Search Tree presented in [Ber+08], the numbers $r_1, \dots, r_n$ are stored in the leafs of the tree while the queried median values are stored in the nodes above. For instance consider the set $\{2, 3, 5, 7, 11, 13, 17\}$ consisting of the first seven prime numbers. The corresponding balanced Binary Search Tree is given in Figure 3.1.

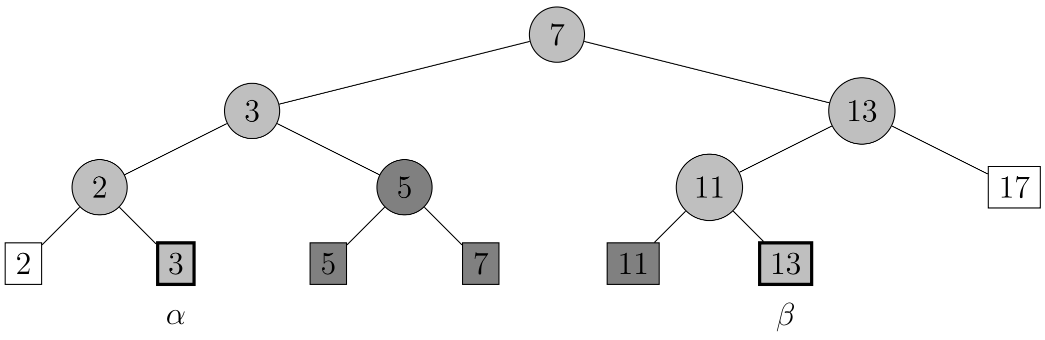


Figure 3.1: A balanced Binary Search Tree for the first seven prime numbers. Circular shapes denote stored median values, while rectangular shapes denote stored input numbers r_i.

Querying the tree from Figure 3.1 with the region $[3, 12]$ we see that the nodes and leafs colored light-gray are traversed during the Binary Search for $a = 3$ and $b = 12$. The last number α seen when performing Binary Search with $q = 3$ is $\alpha = 3$. Accordingly the last number β seen when performing Binary Search with $q = 12$ is $\beta = 13$. In any case it is necessary to report all numbers between α and β, which are marked dark gray in the figure. They can be obtained by traversing certain subtrees. Finally it needs to be checked whether α and β lie in the queried range and they are reported accordingly. We can formalize this 1-dimensional range query on a Binary Search Tree in the following algorithm.
We state the correctness of this algorithm in the following theorem.

Theorem 3. *Algorithm 3 reports exactly those points from the input tree T that lie in the queried range $[a, b]$.*

Proof. First consider any reported point p. If p is stored at the leaf where the path to a or b ends, p is tested explicitly for inclusion in the query range in lines 28 and 39. Otherwise p has been reported in a call of line 22 or line 33. Assume without loss of generality that p has been added to the result by line 22. Recall that the node v_{split} is a splitting node in the sense that the paths to a and b in the tree T go to the left and right subtree of v_{split} respectively,

Algorithm 3 1DimRangeQuery

```
1: procedure 1DimRangeQuery($T$, $[a, b]$) //Binary Search Tree $T$, range $[a, b]$
2:     $v_{split} \leftarrow T.getRoot()$
3:     while $v_{split}$ is not a leaf and ($b \leq$ value($v_{split}$) or $a >$ value($v_{split}$)) do
4:         if $b \leq$ value($v_{split}$) then
5:             $v_{split} \leftarrow v_{split}.getLeft()$
6:         else
7:             $v_{split} \leftarrow v_{split}.getRight()$
8:         end if
9:     end while
10:    //Now $v_{split}$ is the node where $a$ and $b$ lie in the left and right subtree respectively
11:    if $v_{split}$ is a leaf of $T$ then //$v_{split}$ is a leaf of $T$
12:        if value($v_{split}$) $\in [a, b]$ then
13:            return value($v_{split}$)
14:        else
15:            return NULL
16:        end if
17:    else//$v_{split}$ is an internal node of $T$
18:        Result $= \emptyset$
19:        $v_\ell \leftarrow v_{split}.getLeft()$ //Follow a path to $a$
20:        while $v_\ell$ is not a leaf do
21:            if value($v_\ell$) $\geq a$ then
22:                Add all points from the right subtree of $v_\ell$ to the Result
23:                $v_\ell \leftarrow v_\ell.getLeft()$
24:            else
25:                $v_\ell \leftarrow v_\ell.getRight()$
26:            end if
27:        end while
28:        if value($v_\ell$) $\in [a, b]$ then Add value($v_\ell$) to the Result
29:        end if
30:        $v_r \leftarrow v_{split}.getRight()$ //Follow a path to $b$
```

```
31:         while $v_r$ is not a leaf do
32:             if value($v_r$) $< b$ then
33:                 Add all points from the left subtree of $v_r$ to the Result
34:                 $v_r \leftarrow v_r.getRight()$
35:             else
36:                 $v_r \leftarrow v_r.getLeft()$
37:             end if
38:         end while
39:         if value($v_r$) $\in [a, b]$ then Add value($v_r$) to the Result
40:         end if
41:         return Result
42:     end if
43: end procedure
```

see Figure 3.2. Since v_ℓ and hence its right subtree $v_\ell.getRight()$ lie in the left subtree of the splitting node v_{split}, we have $p \leq \text{value}(v_{split})$. Because the search path of b goes into the right subtree of v_{split} we know $p < b$. On the other hand, the search path of a goes into the left subtree of v_ℓ and p is in the right subtree of v_ℓ, therefore $a < p$. It follows that $p \in [a, b]$. Therefore every reported point p does indeed lie in the queried range $[a, b]$.
Now consider a point $p \in [a, b]$ that is stored in the leaf μ of T. Then there exists a node v of T with maximal depth that is visited by Algorithm 3 and is an ancestor of μ. Claim: $v = \mu$, i.e. p is reported. Assume for a contradiction that p is not reported by Algorithm 3. Then v cannot be a node visited in line 22 or line 33, since all descendants of these nodes are reported. Therefore v is a node on the search path to a, on the search path to b, or on both paths. First assume that v is on the search path of a, but not on the search path of b. Then the search path must go left from v and μ must be in the right subtree of v, otherwise v would not be the ancestor with maximal depth. But then μ is included in the call of line 22 and is therefore reported.
The case of v lying on the search path to b and not on the search path to a is similar, hence we will close by considering the case of v lying on both the path to a and the path to b. Assume first that μ is in the left subtree of v. Then the search path of a goes right at v, otherwise v would not be the ancestor of maximal depth. But then $p < a$. Similarly, if μ is in the right subtree of v, the search path of b goes left at v, therefore $p > b$. Both contradict to the assumption that p lies in the queried range. Thus all points in the queried range are indeed reported by Algorithm 3. □

Note how it is crucial in Algorithm 3 to traverse many of the inner nodes of

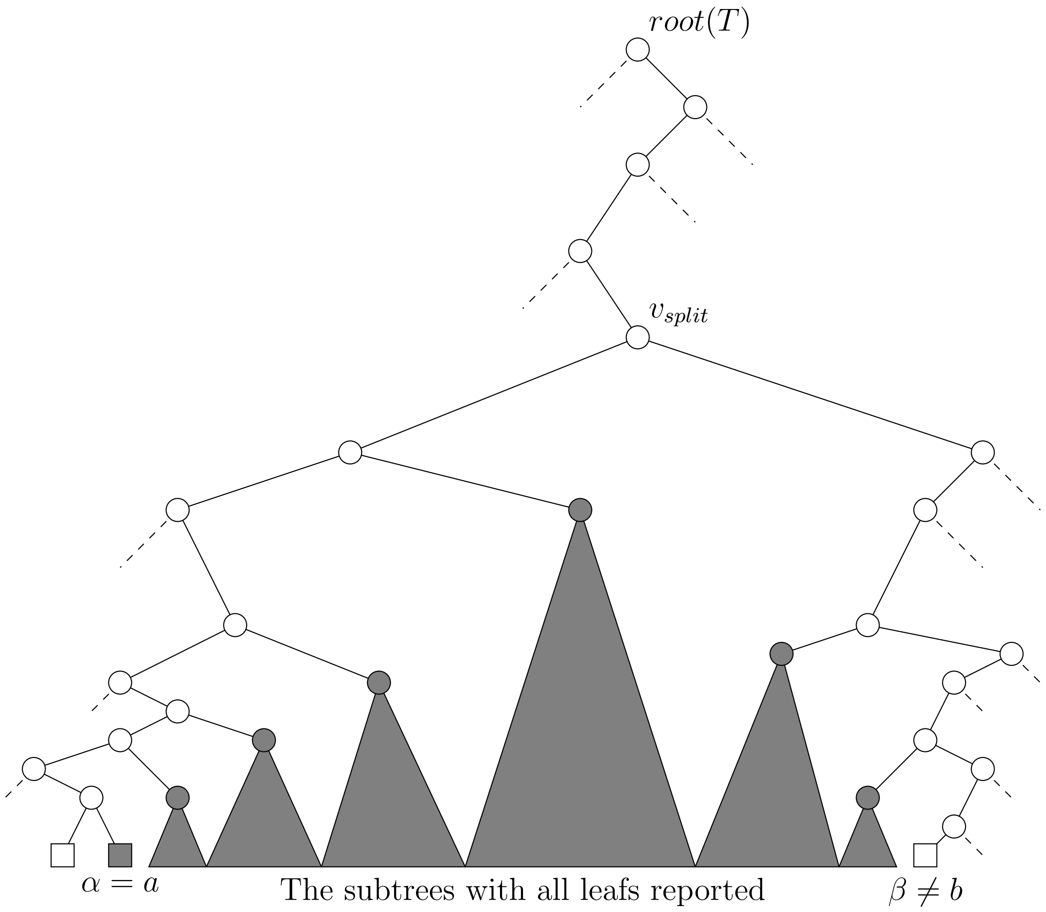


Figure 3.2: A Binary Tree T which is queried for values a and b. The value a is actually stored in a leaf $\alpha = a$ of the tree, while the search path for value b in T ends in leaf β. Compare [Ber+08].

the tree to determine the output. For example, in Figure 3.1 one can see that all inner nodes need to be explored in order to find the numbers from the queried range. To overcome this problem and to compute the output faster we will use AVL Trees [OW02] instead of balanced Binary Search Trees. In the case of AVL Trees all nodes and not only the leafs of the tree will store numbers from the point set. That is, instead of only storing the value of the median, the median itself is stored in the node and does not appear in the left nor in the right subtree. The advantage here is a faster reach of certain nodes since it is not necessary to traverse down to leaf-level anymore. Also the split nodes only storing median values do not have to be stored any more hence saving storage. Explicitly, we know that a Binary Tree with n leafs has $n - 1$ internal nodes [Sed92]. Therefore, by using AVL Trees instead of balanced search tree, we can reduce the needed storage by almost one half. An AVL Tree corresponding to the example from Figure 3.1 is given in Figure 3.3.

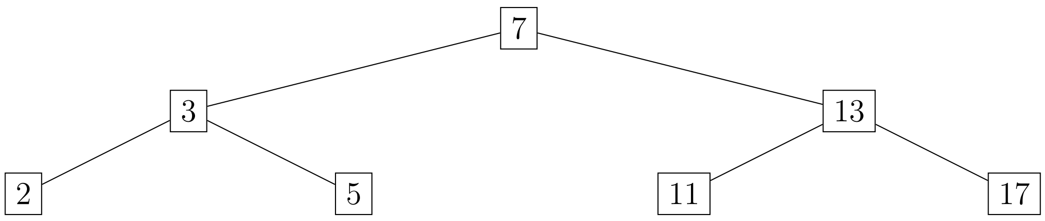


Figure 3.3: An AVL Tree for the first seven prime numbers.

We will now use the ideas from this section and generalize them to higher dimensions. In our presentation we will follow [Ber+08], but will divert from their idea by using AVL Trees rather than balanced Binary Search Trees to use the mentioned benefits.

3.2.2 Kd-Trees in Dimension d

After having presented the general idea of Binary Search and one-dimensional range queries, we will now generalize it to an arbitrary dimension d. For the remainder of this section we will make the following assumption:

Assumption 1. *Given a set of points* $\{p_1, \ldots, p_n\} \subset \mathbb{R}^d$*, denoting by* $p_{i1}, \ldots, p_{id}$ *the d coordinates of the point* p_i*, we assume that*

$$p_{ik} \neq p_{jk} \quad \forall i, j = 1, \ldots, n, \; i \neq j, \; \forall k = 1, \ldots, d. \tag{3.1}$$

Since this is a very strict assumption, in Section 3.2.3 we will show that it can be dropped. We just state it here for the sake of simplicity.
For a given set of points $P = \{p_1, \ldots, p_n\}$ their Kd-Tree is recursively defined. In the first step, the points from P are ordered according to their first coordinate. By Assumption 1 we know that these first coordinates are distinct and hence we can find a unique median m_1 according to this ordering. We introduce a hyperplane $h_1 = \{x \in \mathbb{R}^d \mid x_1 = m_1\}$. Now we split the point set P into a "left" and a "right" half, according to whether $p_{i1} \leq m_1$ or $p_{i1} > m_1$, that is the chosen median always lies in the "left" half and hence the size of the halves differs by at most one. Now we apply this same technique to both the left half and the right half, but the median is

now not taken from the ordered first coordinates, but the second coordinates. Also, the introduced hyperplanes only start at the hyperplane h_1 introduced earlier. When the median is taken from the d-th coordinate, in the next recursion step, the median will be once again taken from the first coordinate, i.e. it is taken in cyclic order from the coordinates.
In Figure 3.4 this recursion is illustrated. The first hyperplane h_1 splits the pointset on the x-axis. The second hyperplane splits the left half of the pointset on the y-axis, but only starts at hyperplane h_1 and then goes off to negative infinity. Since in this case, the points are two-dimensional, the third hyperplane again splits on the x-dimension.

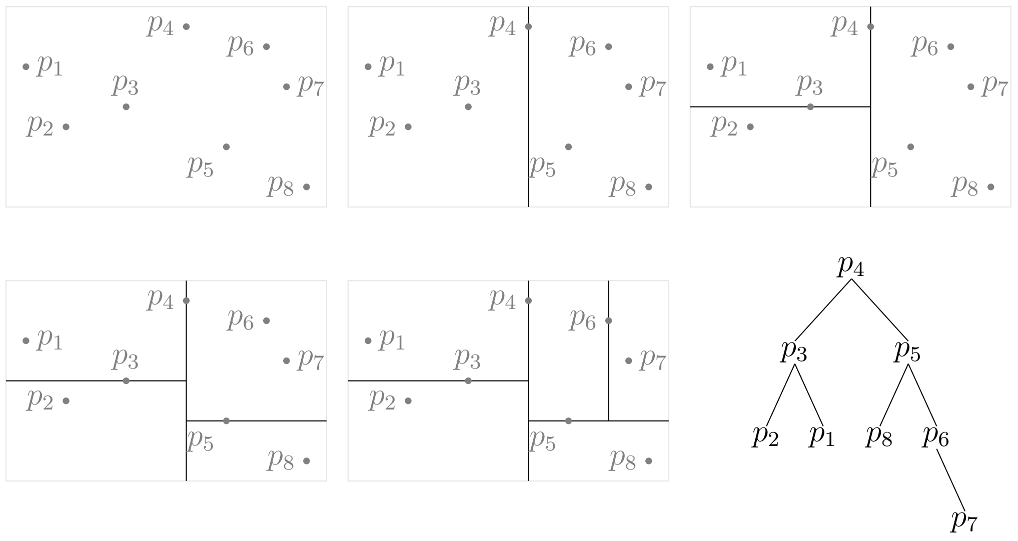


Figure 3.4: A set of eight points is recursively split with four hyperplanes that define the corresponding Kd-Tree, which is also shown. Note that the internal representing p_6 has only one child.

Using this recursion ensures that the result is an AVL Tree that stores the input points corresponding to hyperplanes in its inner nodes, while all other points are stored in the leafs. To formalize the construction of a Kd-Tree we state this building procedure in Algorithm 4.

The algorithm uses the convention that the median belongs to the "left" part of the hyperplane. For this to still obtain an AVL Tree, the median has to be defined in the following way.

Algorithm 4 Build Kd-Tree

```
1:  procedure BUILD$Kd$-TREE(P,$d$,$\delta$) //point set $P$ containing points of dimension $d$, current depth $\delta$
2:      if $|P| = 1$ then
3:          return A leaf storing the single point of $P$
4:      else
5:          $sd \leftarrow \delta \mod d$ //The dimension where to split $P$
6:          $m \leftarrow$ Median of $P$ in the splitting Dimension //Let $m$ bis the median of $P$ according to the values of the points in dimension $sd$.
7:          $H \leftarrow \{x \in \mathbb{R}^d \mid x_{sd} = m_{sd}\}$ //The hyperplane $H$ is an axis-parallel hyperplane containing all points whose $sd$-th coordinate is equal to the $sd$-th coordinate of $m$.
8:          $P_\ell \leftarrow P \cap \{x \in \mathbb{R}^d \mid x_{sd} < m\}$ //All points from $P$ that have $sd$th coordinate strictly smaller to the $sd$-th coordinate of $m$.
9:          $P_r \leftarrow P \cap \{x \in \mathbb{R}^d \mid x_{sd} > m\}$//All points from $P$ that have $sd$-th coordinate strictly larger to the $sd$-th coordinate of $m$.
10:         //Note that because of Assumption 1 there is no other point in $P$ except $m$ with $sd$-th coordinate equal to the $sd$-th coordinate of $m$.
11:         $n_\ell \leftarrow$BUILD $Kd$-TREE$(P_\ell, d, \delta + 1)$
12:         $n_r \leftarrow$BUILD $Kd$-TREE$(P_r, d, \delta + 1)$
13:         return A node $\nu$ storing $m$, representing $H$, with $n_\ell$ as left and $n_r$ as right child.
14:     end if
15: end procedure
```

Definition 3. *Given points $P = \{p_1, \ldots, p_n\}$ with $p_i \prec p_{i+1}$ according to some total order $\prec$. Then the median according to the order $\prec$ is $p_{\lceil n/2 \rceil}$.*

Considering Algorithm 4 and following [Ber+08] we can show the next lemma.

Lemma 3. *A Kd-Tree for a set of n points uses $\mathcal{O}(n)$ storage and can be constructed in $\mathcal{O}(n \log(n))$ time.*

Proof. In Algorithm 4 in each recursion step the most time-consuming part is finding the median. This can be done in $\mathcal{O}(n)$, see Chapter 6. Therefore the building time $T(n)$ of a Kd-Tree satisfies the following recursion

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{if } n = 1, \\ \mathcal{O}(n) + 2 \cdot T(\lceil n/2 \rceil), & \text{if } n > 1, \end{cases}$$

which solves to $T(n) = \mathcal{O}(n \log(n))$ [SW11].
Note that one can also find the median by initially sorting d lists according to one of the d dimensions respectively and passing on the sorting to the recursion steps. The initial sorting takes $\mathcal{O}(n \log(n))$ time and splitting the sorting to pass it on takes $\mathcal{O}(n)$ time which subsumes the proven bound.
Concerning the storage bound, each point p_i from the input point set is stored in either an internal node or a leaf, hence to store the n input points takes $\mathcal{O}(n)$ storage. $\square$

3.2.3 Generalization to finite d-dimensional point sets

In practical applications, Assumption 1 is generally not satisfied. Therefore we need to be able to drop this assumption and still be able to perform Algorithm 4. From the description in Section 3.2.2 we know that Assumption 1 is necessary to assure that a median can be found on any dimension. In order to overcome this, we turn to the concept of General Points as introduced in [Ber+08].

Definition 4. *Consider a given point $p \in \mathbb{R}^d$, with coordinates $p_1, \dots, p_d$. The corresponding i-th* ***composite coordinate*** *p'_i is the cyclic concatenation*

$$p'_i = (p_i | p_{i+1} | \dots | p_d | p_1 | \dots | p_{i-1}). \tag{3.2}$$

The ***General Point*** *p' corresponding to p is*

$$p' = \begin{pmatrix} p'_1 \\ \vdots \\ p'_d \end{pmatrix}. \tag{3.3}$$

Two composite coordinates $a' = (a_1 | \dots | a_d)$, $b' = (b_1 | \dots | b_d)$ are ordered lexicographically, that is

$$a' < b' :\Leftrightarrow \exists j \in \{1, \dots, d\} \text{ s.t. } a_j < b_j \text{ and } a_k = b_k \ \forall 1 \leq k < j. \tag{3.4}$$

Note that p'_i from the definition can be seen as a row-vector with d entries and p' can be interpreted as a $d \times d$ matrix. Using this lexicographical order, we can find the median of a point set according to a given dimension. Since the composite coordinates as defined in (3.2) carry the original point coordinate of the given dimension as first item, by ordering lexicographically, we ensure that the desired order is kept intact. Therefore by using General Points instead of the points in the input set we can weaken Assumption 1 to the following.

Assumption 2. *Given an input point set $\{p^1, \dots, p^n\} \subset \mathbb{R}^d$, we assume that*

$$p^i \neq p^j \ \forall i, j = 1, \dots, n, \ i \neq j. \tag{3.5}$$

To wrap up things so far, consider the points p, q, and their corresponding General Points.

$$p = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}, \quad p' = \begin{pmatrix} 2|3|5 \\ 3|5|2 \\ 5|2|3 \end{pmatrix}, \quad q = \begin{pmatrix} 2 \\ 7 \\ 10 \end{pmatrix}, \quad q' = \begin{pmatrix} 2|7|10 \\ 7|10|2 \\ 10|2|7 \end{pmatrix},$$

Note that the points p and q violate Assumption 1, but satisfy Assumption 2. Considering the first dimension, we can establish the ordering

$$2|3|5 < 2|7|10$$

according to the lexicographical order as defined in (3.4).
To end this chapter, note that in an explicit implementation of a Kd-Tree Assumption 2 can be dropped by telling apart two points with identical coordinates. We will explain an approach to this in Section 3.3.1. Furthermore, although Definition 4 suggests that for each point p of the input a matrix p' would have to be stored, this is not the case. In Section 3.3.1 we will show how this problem can be solved by using a lexicographical order on the input points. Furthermore note that we did introduce one-dimensional range queries in Section 3.2.1 but did give a generalization on Kd-Trees. That is not to say that higher dimensional range queries can not be performed on Kd-Trees, but they are not necessary for our applications as given in Chapter 2. The interested reader can find a description of higher dimensional range queries in [Ber+08]. Finally note that if a Kd-Tree is stored in external memory with e.g. each node in its own page, then for each node a disk I/O has to be performed. This problem can be overcome as outlined in [Pro+03].

3.3 Implementations

In this section we want to briefly present how the concepts introduced in Section 3.2 are implemented. For all implementations in this thesis, we will use the JavaView framework, see [Pol+]. Benchmarks on the different implementations are given in Chapter 8.

3.3.1 Lexicographical Order

As presented in Section 3.2.3 to be able to apply the data structure of Kd-Trees, we need to impose an order on the points $p_\iota \in P$ such that we can uniquely determine a median according to a given dimension k. An appropriate order is given in Definition 4. For our implementation we will make use of the generic Java interface `Comparator<T>` as documented in [Ora]. Since our implementation will be based on JavaView and thus uses the corresponding class `PdVector` for d-dimensional vectors, our Comparator implementation will be `Comparator<PdVector>`. The only essential values to store are the `dimension` of the `PdVectors` which are to be compared, as well as the `startDimension`, that is the dimension in which the median for the building process of the Kd-Tree is searched.

By definition of the `Comparator<T>` interface, the only necessary function of a `Comparator<T>` implementation is the `compare(T arg0,T arg1)` method which is to return

$$\text{compare}(a,b) = \begin{cases} -1 & \text{if } a < b, \\ 1 & \text{if } a > b, \\ 0 & \text{if } a = b. \end{cases}$$

Recall that by Assumption 2 we want to have all points distinct. In fact we will realize the `compare` function in a way such that even two `PdVectors` p, q that agree in all coordinates are considered to be distinct in the sense, that one of them in our order will be smaller than the other. To achieve this, we evaluate the hash code of the object, which is a stable value throughout all computations. Thereby we can even drop Assumption 2. The general idea of the `compare` function is given in Algorithm 5 and the whole class Java implementation is given in Appendix B. The outlined `Comparator<PdVector>` will be used in the following.

Algorithm 5 Compare

```
1: procedure COMPARE($p$,$q$) //Points $p$ and $q$
2:     for int $i = 0$;$i < d$;$i$++ do
3:         dim ←(startDimension+$i$)  mod dimension
4:         if $p_i < q_i$ then return -1
5:         else if $p_i > q_i$ then return 1
6:         end if
7:     end for
8:     if $p$.hashCode()< $q$.hashCode() then return -1
9:     else if $p$.hashCode()< $q$.hashCode() then return 1
10:    end if
11:    return 0
12: end procedure
```

3.3.2 Abstract Kd-Tree

When implementing the Kd-Tree within the JavaView framework, we can make use of the different polymorphic abilities of the Java language as e.g. presented in [Fla98]. A Kd-Tree will, independent of the way it is build, provide an implementation of the methods outlined in Section 3.1 and explained in Chapter 5. To realize this, we will implement an abstract Kd-Tree incorporating all features for nearest neighbor search as given in Chapter 5. Although the abstract Kd-Tree already provides a `root` variable to access its root node, it does not provide a constructor, which has to be provided by the extending class. In order to assure that the user of the abstract class actually implements a constructor, we introduce the item given in Listing 3.1.

```java
/** This methods creates a KdTree from the given set of
    points.
* It has to be implemented by the class implementing this
* abstract class.
* @param points The set of points to be represented by the
* KdTree.
*/
protected abstract void buildTree(PgPointSet points);
```

Listing 3.1: An abstract method to force the user of the class to implement a constructor-like method. Note that this method still needs to be called by the user in the constructor of the class implementing the abstract Kd-Tree.

Apart from the stated method in Listing 3.1, the abstract Kd-tree provides the methods listed in Listing 5.1. They are either generic or related to the procedure of finding nearest neighbors and hence will be explained in Section 5.3.

3.3.3 Sorting

During the process of building a Kd-Tree, the median of the point set according to a given dimension has to be determined. This can be done quite trivially by sorting the set using the lexicographical comparator presented in Section 3.3.1. Assume that the point set is sorted using the correct order and has n elements, then the median is the $\lfloor n/2 \rfloor$th element. We could now in each recursion step sort the considered subset. However, we aim for a slightly more elaborate approach.
Assume that the Kd-Tree is to handle d-dimensional data. Then the idea is to initially create d lists $L_1, \ldots, L_d$, each containing the whole point set and each list L_i is sorted using Algorithm 5 on `startDimension` i. Now when the median m is to be determined on dimension j, we easily find it as the $\lfloor n/2 \rfloor$th element of the list L_j. For the recursion step, each list L_i, $i \neq j$, has to be partitioned in a way such that all $\lfloor n/2 \rfloor$ elements smaller than m are put in front of the median and all$\lfloor n/2 \rfloor$ elements larger than m are stored behind the median in L_i. Here, "smaller" and "larger" refer to the order induced by Algorithm 5 with `startDimension` j. However, the elements on L_i are moved in a way such that any two points below the median are still ordered according to the order obtained with `startDimension` i. We formalize this

in Algorithm 6. The whole class implementation is given in Appendix C. Finally note that if storage was an issue we could in this procedure also store the indices of points in the list and work on them.

3.3.4 Median

Sorting of a list containing n elements can be done in $\mathcal{O}(n \log(n))$, see [SW11]. However, we will see in Chapter 6 that sorting is not the fastest way to determine the median. Hence apart from a Kd-Tree implementation that simply sorts its points to build it, we also offer an implementation that actually computes the median from the list in each recursion step. The advantages are obvious: We have no need for storing a list for each dimension, but we only need to work on the original point cloud. Also the overhead of sorting the lists, as outlined in the previous section, is unnecessary. Since we do not want to settle for a particular median algorithm at the start, we use the strategy design pattern as given in [Gam+94]. That is, we provide an interface for a median algorithm as shown in Listing 3.2 and provide the Kd-Tree during its building process with an instance implementing this interface. See Chapter 6 for the offered instances.

```
public interface IMedianAlgorithm {
  public PdVector median(PdVector[] points, int dim);
}
```

Listing 3.2: Interface to use the Strategy design pattern for median computation

The whole code of the Kd-Tree class using median computation is given in Appendix D. Note that computing the median still requires partition on the point set to be able to recursively find medians on the lower and upper halves. However, only one partition has to be performed and not dimension many partitions as in Section 3.3.3.

Algorithm 6 Build Kd-Tree Sorting

```
procedure BuildSorting($P$,$d$) //Point set $P$ of dimension $d$
    for int $i = 0$;$i < d$;$i$++ do
        Create a List $L_i$ to hold all points from $P$ and sort it lexicographically with start dimension $i$.
    end for
    RecursiveBuild($L_1, \dots, L_d$,0,0,$|P| - 1$)
end procedure

procedure RecursiveBuild($L_1, \dots, L_d$,$j$,$s$,$e$)//Sorted Lists $L_i$, current dimension $j$, start index $s$, end index $e$
    if $e < s$ thenreturn NULL
    else if $s = e$ thenreturn Leaf storing the $L_j(s)$.
    end if
    $m_i \leftarrow \lfloor (s + e)/2 \rfloor$ //index of the median
    $m \leftarrow L_j(m_i)$ //find the median
    for all Lists $L_i$, $i \neq j$ do
        Partition($L_i$,$m$,$j$,$s$,$e$)
    end for
    $\ell \leftarrow$ RecursiveBuild($L_1, \dots, L_d$,$j + 1 \mod d$,$s$,$m_i - 1$)
    $r \leftarrow$ RecursiveBuild($L_1, \dots, L_d$,$j + 1 \mod d$,$m_i + 1$,$e$)
    return Node storing $m$ with left child $\ell$ and right child $r$
end procedure

procedure Partition($L_i$,$m$,$j$,$s$,$e$)//List $L_i$ ordered on dimension $i$, median $m$ obtained in order on dimension $j$, start and end index $s$ and $e$
    Create a queue $Q$
    $k \leftarrow s$
    for all points $p$ in $L_i$ from $s$ to $e$ do
        if $p <_j m$ then//Compare using Algorithm 5
            $p$ is stored at position $k$ in $L_i$ and $k \leftarrow k + 1$.
        else if $p >_j m$ then//Compare using Algorithm 5
            Add $p$ to $Q$.
        end if
    end for
    Place $m$ at index $\lfloor (s+e)/2 \rfloor$ in $L_i$, put all elements of $Q$ to the places $\lfloor (s + e)/2 \rfloor, \dots, e$ in $L_i$, keeping their order from $Q$.
end procedure
```

Chapter 4

Two more Spatial Data Structures

In Chapter 3 we introduced the general idea of neighborhood computation on spatial data structures. We did also introduce the data structure of a Kd-Tree. In this chapter, we will present two more data structures that are used in similar applications as the Kd-Trees. For both present structures we will reason why they are inferior to Kd-Trees in our application.

4.1 Quadtree and Octree

Quadtrees are spatial data structures that split a d-dimensional space recursively into 2^d equally sized cells. In the three dimensional case they are also called Octrees. Since the generalization to the third dimension is straight forward, we will present it along the way of giving the concept for two dimensions.
Given a point set P, the construction of a two-dimensional quadtree starts with a bounding square around P, which will be the initial cell of the quadtree. Recursively, each cell is subdivided into four equally sized square cells until every cell of the Quadtree only contains one point p_ι of the input set P. In the two-dimensional case, the cells are usually labeled according to the respective cardinal direction, i.e. North-East (NE), North-West (NW), South-West (SW), and South-East (SE). For Octrees and higher dimensional Quadtrees, the cells are usually numbered, as shown in Figure 4.1.

This procedure gives a tree, where the root corresponds the initial bounding square. Each internal node now has four children, corresponding to the four equally sized square cells that the cell corresponding to the considered node

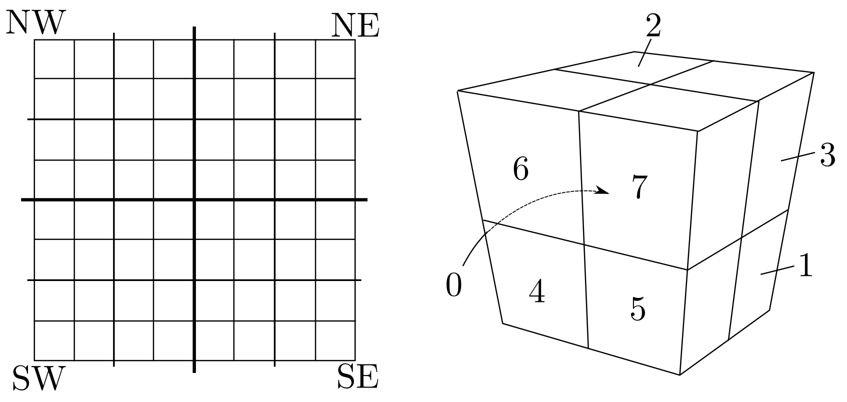


Figure 4.1: The labeling of a Quadtree and an Octree, cf [Bri05] and [Eri05].

is split up into. The leafs of the tree either represent a point from the input set P or are empty. An example for a Quadtree and its building process is given in Figure 4.2.

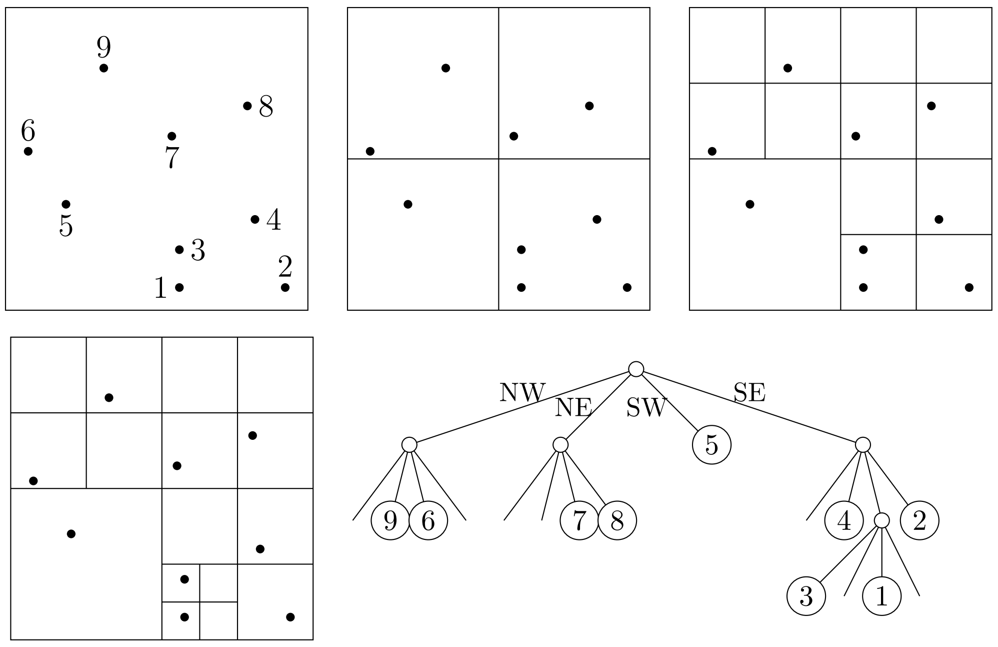


Figure 4.2: A set of nine points is recursively put into a Quadtree data structure. The final Quadtree is also shown. Note how some of the leafs are empty and the tree is not balanced. Compare to [Bri05].

For a more thorough introduction see [Bri05] and for implementation detail see [Eri05]. For details on nearest neighbor procedures on Quadtrees see

[Vai89]. From the example in Figure 4.2 it already becomes obvious that an Octree is not balanced. On the contrary, for any distribution of points apart from a uniform distribution, we expect an Octree to be unbalanced. The difference of depth of certain leafs can be arbitrary high already for an Octree on three points. To see this consider the points

$$p_1 = \left(\frac{1}{4}, \frac{1}{4}\right), \quad p_2 = \left(\frac{2^{k+1}-3}{2^{k+1}}, \frac{2^{k+1}-1}{2^{k+1}}\right), \quad p_3 = \left(\frac{2^{k+1}-1}{2^{k+1}}, \frac{2^{k+1}-3}{2^{k+1}}\right). \tag{4.1}$$

For $k \in \mathbb{N}$, $k \geq 1$, the Quadtree representing these three points has height k. For an illustration see Figure 4.3. Although this is a theoretical example and

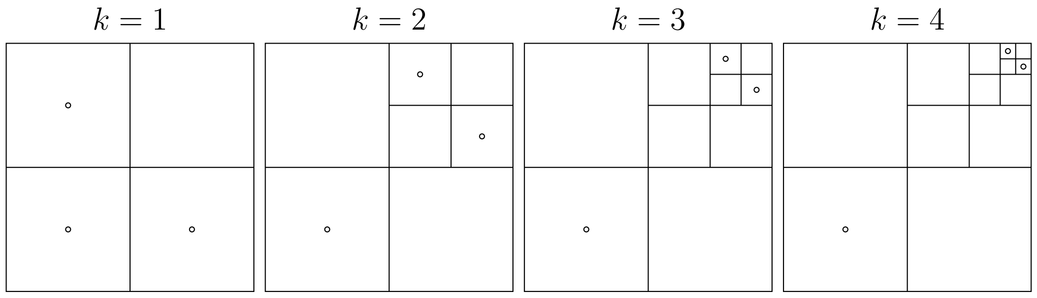


Figure 4.3: From left to right: Different Quadtrees for the points p_1, p_2, p_3 as given in (4.1) for values $k = 1, 2, 3, 4$. Note how the Quadtree has exactly depth k in each illustration.

in terms of our application as outlined in Chapter 2 these extreme behaviors of a Quadtree will not arise, we still dismiss the concept of the Quadtree in favor of Kd-Trees which are always balanced.
To end this section on Quadtress note that they do possess a big advantage over Kd-Trees. Namely insertion and deletion of points can be efficiently done in a Quadtree, while a Kd-Tree in general has to be re-built. However, in the application given in Chapter 2 we do not need our data structure to change, since we fix a neighborhood at the beginning of the computation and do not change it throughout the algorithm, see Section 2.1.3.

4.2 R-Tree

The indexing structure of R-Trees has been proposed by [Gut84]. It is a generalization of B-Trees [Com79]. By design the R-Tree is to handle a spatial database, which consists of index record entries of the form

$$((p_{\min}, p_{\max}), id),$$

where the pair $p_{\min}, p_{\max} \in \mathbb{R}^n$ defines an n-dimensional rectangular bounding box containing a spatial object O and id is an identifier for O. Note here that for $O = p \in \mathbb{R}^n$ a single point, p can be stored in the degenerated hyper-rectangle $p_{\min} = p_{\max}$. While the records introduced above are stored in the leafs of an R-Tree, the internal nodes store similar records

$$((p_{\min}, p_{\max}), child),$$

where the pair $p_{\min}, p_{\max} \in \mathbb{R}^n$ again defines an n-dimensional rectangular bounding box and *child* is a pointer to a child of the internal node. The key properties of an R-Tree are given by [Gut84] to be

1. Every leaf of the R-Tree stores between m and M many records unless it is the root.

2. For each record $((p_{\min}, p_{\max}), id)$ in a leaf node, $(p_{\min}, p_{\max})$ is the smallest hyper-rectangle that contains the object O identified by id.

3. Every internal node has between m and M many children unless it is the root.

4. For each record $((p_{\min}, p_{\max}), child)$ in an internal node, $(p_{\min}, p_{\max})$ is the smallest hyper-rectangle that contains all rectangles in the given *child* node.

5. The root has at least two children unless it is a leaf.

6. All leafs appear on the same level.

Note that the hyperrectangles of internal nodes and leafs can be pairwise overlapping. An illustration of the R-Tree concept is provided in Figure 4.4.

R-Trees provide search algorithms that output all ids whose rectangles overlap with a search rectangle $(s_{\min}, s_{\max})$. Furthermore new objects can be inserted into the tree and objects from the tree can be deleted. In the first case nodes might be split to satisfy they have between m and M many children. Different splitting strategies and a full description of the algorithms on R-Trees can be found in [Gut84], [GUW02], as well as in [SC03] where also different models as R+-Trees and R*-Trees are discussed. For an explicit description about a nearest neighbor algorithm using R-Trees, see [RKV95]. In [Els+12] several data structures are compared according to their performance on nearest-neighbor search strategies. The authors come to the following conclusion:

> The R-tree library SpatialIndex performs about on par to the STANN library. Both were generally slower than the Kd-Tree implementations.

Therefore we dismiss R-Trees as possible data structures for our application as outlined in Chapter 2. It is also worth mentioning [Jun11], where it is shown that on the chosen benchmark example of the thesis, a static Kd-Tree performs $221,608$ writing operations while the R-Tree from [Gut84] performs $1,539,451$ writing operations during the same example.

4.3 Choice of a Data Structure, Curse of Dimensionality

In Sections 4.1 and 4.2 we gave reasons for our choice to pick the data structure of a Kd-Tree over a Quadtree or an R-Tree. Our reasoning was mostly based on assumptions and experimental data as presented in [Els+12]. At this point we would like to pose the question:

> For a set of points P in dimension d, what is the fastest data structure to use in order to perform nearest neighbor queries on P?

Sadly, there is no final answer to this question. This is mainly because the dimension here is variable. That is to answer the question above we would need to find a data structure that behaves good independent of both the dimension of the ambient space and the dimension of the embedded point set. However, as dimension grows, new phenomena can be observed. For example it was shown in [Bey+99] that

> assuming the distance distribution behaves a certain way (...) the difference in distance between the query point and all data points becomes negligible.

In other words, if $d_{\max} = \max_{p_i,p_j \in P} \|p_i - p_j\|$ and $d_{\min} = \min_{p_i,p_j \in P} \|p_i - p_j\|$, then

$$\lim_{d\to\infty} \frac{d_{\max}}{d_{\min}} \to 1$$

for certain conditions on the distribution of the p_i and the size of P as given in [Bey+99]. This has an immediate effect on the nearest neighbor procedure as outlined in 3.1. The advantage of the procedure lies in an expectedly large

cut off of points from P, i.e. we hope not to have to consider all points of P, but only a certain subset. As the fraction $\frac{d_{\max}-d_{\min}}{d_{\max}}$ becomes smaller, less and less points from P can be cut off and have to be considered when looking for nearest neighbors.
Another effect of high dimensions is the formation of so called hubs in the point set P. A hub is a point $p_\iota \in P$ that appears as a nearest neighbor to unusually many points $x_j \in P$. For a formal description of the phenomenon see [RNI10]. It is interesting since it can be shown to be an inherent property of data distributions in high-dimensional vector space and thus also appears in real-world data. The emergence of hubs with growing dimension strongly effects the structure of the directed k-nearest-neighbor-graph and thereby has a strong impact on any algorithm for k nearest neighbor search.
Because of the presented reasons we are not able to give one analysis that once and for all settles the question for a best suited data structure for nearest neighbor computation on point sets. This explains why we have to retreat to experimental data in our reasoning about the choice of a data structure.

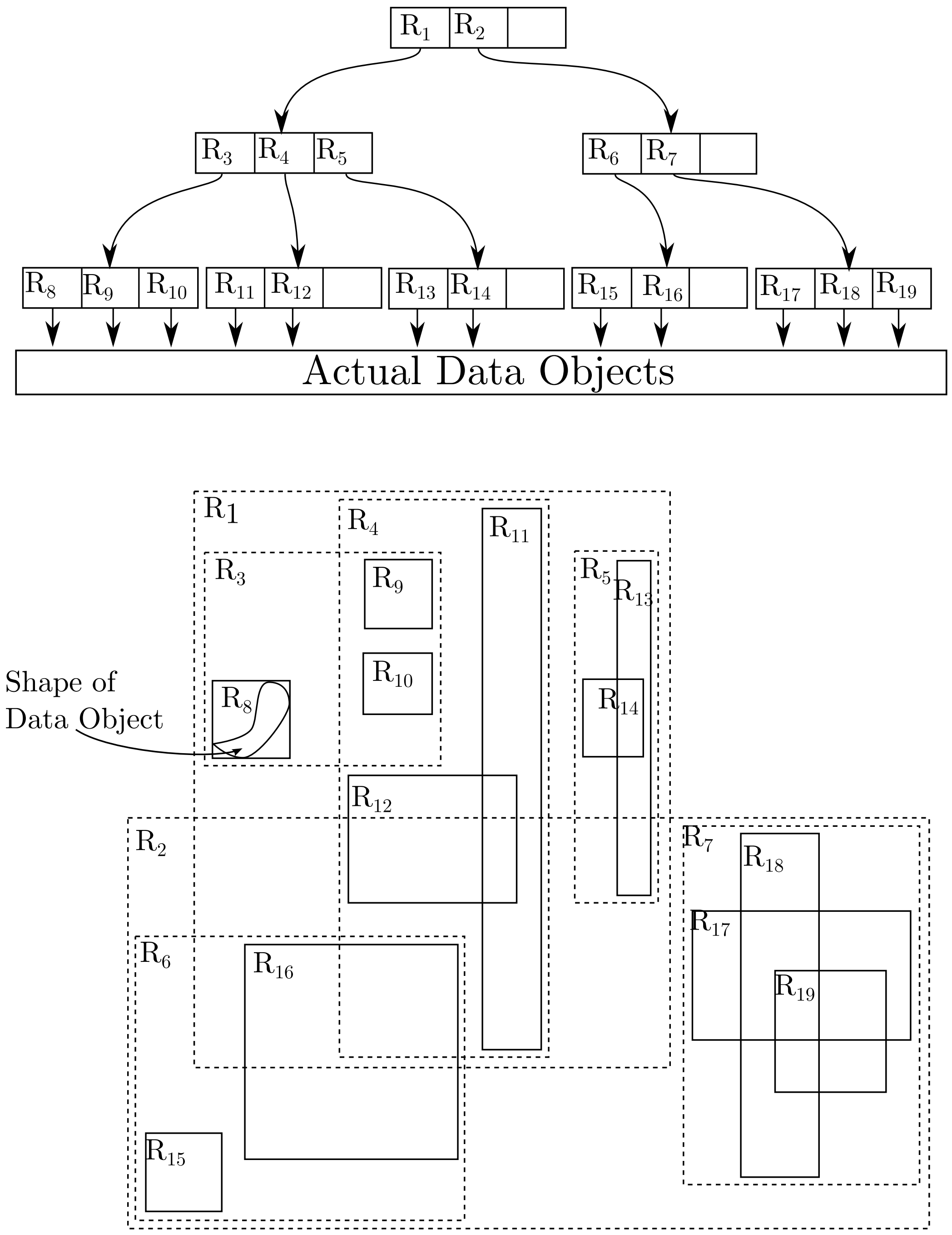


Figure 4.4: An R-Tree in $\mathbb{R}^2$ and the corresponding rectangles. The dotted drawn rectangles correspond to internal nodes of the R-Tree, while the fully drawn rectangles are rectangles storing the actual objects. In R_8 the corresponding object is shown. See [Gut84].

Chapter 5

Nearest Neighbor Search

In Chapter 3 we gave the general idea of nearest neighbor computation in Algorithm 1. In this chapter we will now use this idea, to set up a Nearest Neighbor Search algorithm on Kd-Trees. Recall that nearest neighbor computation can be naively done by just iterating over all points. This leads to the following algorithm. Note that we do not require the input point to be

Algorithm 7 Naive Nearest Neighbor

```
1: procedure NaiveNearestNeighbor($P = \{x_j \mid j = 0, \ldots, n-1\}$,$p$)
   //point set $P$, point $p$, not necessarily element of $P$
2:     min ← $x_0$
3:     for $j = 0$; $j = |P| - 1$; $j++$ do
4:         if $\|p - x_j\| <$ min then
5:             min ← $x_j$
6:         end if
7:     end for
8:     return min
9: end procedure
```

an element of the point set P. We will maintain this throughout the chapter. However, in the practical application outlined in Chapter 2 we want to query for nearest neighbors of a point $p_\iota \in P$ and we do not want p_ι to be reported as the nearest neighbor. We will describe solutions to this in Section 5.3. Finally keep in mind that Algorithm 7 has a runtime of $\mathcal{O}(n)$, that is, determining a nearest neighbor for each point x_j of the point set P takes $\mathcal{O}(n^2)$. In the remainder of the chapter we want to give two more algorithms on how to compute nearest neighbors. The first will be given in Section 5.1 and will be using Principal Component Analysis. The second will be given in Section 5.2 and will be a realization of the general idea outlined

in Algorithm 1. We will then explain how that realization is implemented in JavaView, which will be done in Section 5.3 and finally in Section 5.4 we try to modify the Kd-Tree slightly to obtain even better results for the nearest neighbor algorithm.
Before we get into the algorithms note, that these algorithms will determine $\varepsilon - k-$neighborhoods. That is, as described in Section 2.1.1, given a queried point p_ι of a point set P: The intersection of $P \cap B_\varepsilon(p_\iota)$ and the k sample points from P closest to p_ι. To find the nearest neighbor, it suffices to set $k = 1$ and ε to e.g. the diameter of the axis aligned bounding box of P.

5.1 Nearest Neighbor using PCA

The following approach to a computation of nearest neighbors uses Principal Component Analysis (PCA) as given e.g. in [Han10]. We first give a description of the procedure, than formalize it as Algorithm 8. A JavaView implementation is given in Appendix A.
Given a point set P, we will compute the eigenvalues and eigenvectors of the covariance matrix of P as defined in (2.8), except we take the sum over all points in P, not only over a neighborhood of a fixed point. The unit eigenvector a_1 to the largest eigenvalue λ_1 of the covariance matrix is the direction of largest variance in P. The unit eigenvector a_2 to the next largest eigenvalue λ_2 is the direction of largest variance in P amongst all those vectors orthogonal to a_1. In general a_i, the unit eigenvector to the ith largest eigenvalue is orthogonal to $a_1, \ldots, a_{i-1}$ and under this condition gives the direction of largest variance in P.
In our setting in $\mathbb{R}^3$, there will be three principal directions a_1, a_2, a_3. The vertices of P are now ordered in three lists L_1, L_2, L_3 according to their position regarding the principal directions. We now iterate through all vertices $p_\iota \in P$. For a start we determine the smallest index $\min_1$ such that the distance of the points $L_1(\min_1)$ and p_ι in direction a_1 is less or equal to ε. For the lists L_2 and L_3 we determine indices $\min_2$, $\min_3$, $\max_2$, $\max_3$. Indicating the range of points whose distance to p_ι on the direction a_2 and a_3 respectively is less or equal than ε.
Finally, we iterate over all indices starting at $\min_1$. If we come to an index whose point has larger distance to p_ι on the direction a_1 than ε, there are no more points to consider. For all points that we consider before running into this break, we need to check whether their corresponding index on L_2 and L_3 lies in $[\min_2, \max_2]$ and $[\min_3, \max_3]$ respectively. If so, the only thing left to do is compute the actual euclidean distance of the point to p_ι, if it is smaller than ε, the point is added to the list of neighbors. A last thing to

do, after the break condition of the iteration is reached, is to check, whether the list of neighbors contains more than k points and if so, trim it accordingly.

Algorithm 8 Nearest Neighbor using PCA

```
1:  procedure NearestNeighborPCA($P = \{x_j \mid j = 0, \ldots, n-1\}$, $\varepsilon$, $k$)
    //point set $P$, influence $\varepsilon$, max. valence $k$
2:      $M \leftarrow$computeCovariance($P$)
3:      $[a_1, a_2, a_3] \leftarrow$computeEigenvectors($M$)
4:      for $j = 0, \ldots, n-1$ do
5:          $L_i$.add($\langle x_j, a_i \rangle$), $i = 1, 2, 3$
6:      end for
7:      Sort($L_i$), $i = 1, 2, 3$//Order the points on the principal directions
8:      for $\iota = 0, \ldots, n-1$ do //Find NN for all points $p_\iota$
9:          $\min_i \leftarrow \min\{j \mid L_i(j) \leq \varepsilon\}$, $i = 1, 2, 3$
10:         $\max_i \leftarrow \max\{j \mid L_i(j) \leq \varepsilon\}$, $i = 2, 3$
11:         for $h = \min_1, \ldots, n-1$ do
12:             if $\iota < j$ and $L_1(h) - L_1(\iota) > \varepsilon$ then
13:                 Break the for loop
14:             end if
15:             if (Index of $x_j$ in $L_2 < \min_2$ or $> \max_2$) or (Index of $x_j$ in $L_3$
    $< \min_3$ or $> \max_3$) then
16:                 Continue with the next iteration in the for loop
17:             end if
18:             $d \leftarrow \|p_\iota - x_j\|$
19:             if $d > \varepsilon$ then
20:                 Continue with the next iteration in the for loop
21:             end if
22:             neigh.add($x_j$)
23:         end for
24:         if Length(neigh)$> k$ then
25:             Trim neigh to size $k$ keeping the $k$ points closest to $p_\iota$.
26:         end if
27:     end forreturn neigh
28: end procedure
```

This algorithm will serve as a benchmark in Chapter 8 for the procedures of the following section.

5.2 Nearest Neighbor Search using Kd-Trees

In Section 3.1 we already gave a general idea on how data structures can speed up the procedure of finding nearest neighbors. We will now make this explicit for the data structure of Kd-Trees. In our presentation we follow [Moo91], but divert slightly, since [Moo91] considers Kd-Trees to be Binary Trees, while we work on AVL Trees.
The algorithm works recursively. Given a query point p_ι it traverses the Kd-Tree to the leaf that would store the point p_ι, if it was stored in a leaf of the tree. Note that this leaf is uniquely determined since the leafs of the Kd-Tree partition the whole space. Already during this traversal, the points x_j from the internal nodes are stored as possible candidates for nearest neighbors of p_ι, if they satisfy $\|x_j - p_\iota\| < \varepsilon$. However, only k points are stored in an order corresponding to increasing distance to p_ι. In case of a $(k+1)$-th point being added, the point x_j that has largest distance to p_ι, amongst the possible nearest neighbor candidates, is deleted from the candidate list. Once the leaf representing the region, where p_ι lies, is reached, the point of the leaf is also stored as a possible candidate. The tree is traversed back to the root, where

- at each internal node v the following check is performed. If in the list of current candidates for nearest neighbors less than k elements are stored, or the distance of p_ι to the hyperplane represented by v is less than the distance of p_ι to the furthest point from the current candidate list. That is, the other side of the hyperplane is examined if and only if the list still needs points or the other side of the hyperplane might contain a point closer to p_ι than some element from the current list. If the check is done successfully, the subtree on the other side of the node is inspected recursively.

- at each leaf ℓ the point x_ℓ stored in ℓ is added if the current candidate list does not have k points yet, or $\|x_\ell - p_\iota\| < \|x_j - p_\iota\|$ where x_j is the point from the current candidate list that is furthest from p_ι.

Note that the recursive traversal of a subtree of the Kd-Tree is performed in the same as the whole procedure is: At first the algorithm traverses to the leaf closest to p_ι and then goes back to the root of the subtree, performing the check outlined above. We give an example of this procedure in Figures 5.1 to 5.6 for the search of one neighbor in $\mathbb{R}^2$ with an ε such that all given points are within influence radius. Figure 5.1 shows a set of seven points in $\mathbb{R}^2$ and their corresponding Kd-Tree as well as the queried point p for which we search a nearest neighbor.

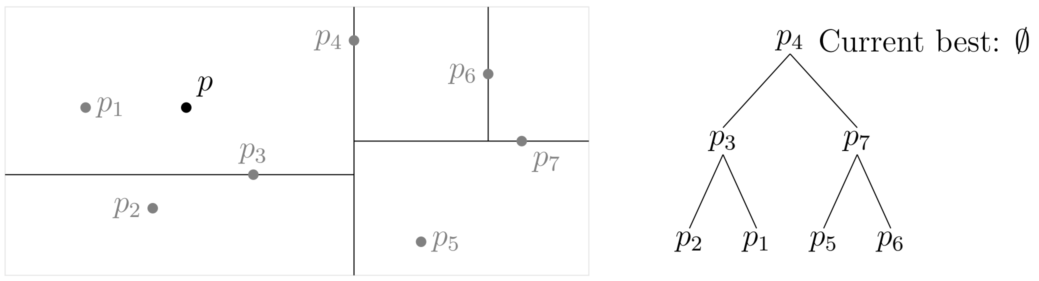


Figure 5.1: Initial point set and its Kd-Tree with the query point p.

Figure 5.2 shows how, according to our procedure, we start to traverse the Kd-Tree at its root, p_4, which is stored as currently closest point to p.

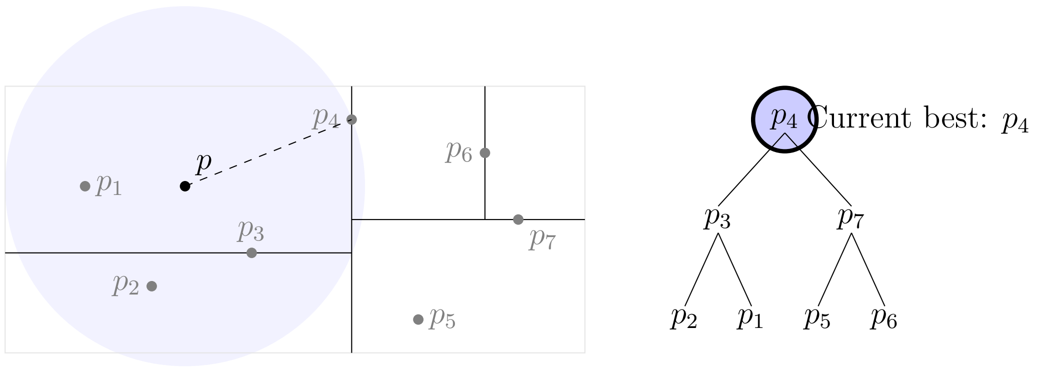


Figure 5.2: Root is examined and stored as nearest neighbor.

Figure 5.3 illustrates how the traversal continues in the direction of p, that is to the left side of the hyperplane of p_4 and considers the point p_3, which is closer to p than p_4 and is hence the new nearest neighbor.

In Figure 5.4 we see the initial traversal coming to an end at the leaf of the Kd-Tree storing p_1. The procedure considers p_1, but keeps p_3 as currently closest neighbor to p.

The traversal is now reversed and the hyperplane at p_3 is examined, see Figure 5.5. Since the distance of p to the hyperplane at p_3 is smaller than the distance of p to its currently nearest neighbor, p_3, the other side of the hyperplane is also considered. But the only node in the other subtree is p_2, which is further to p than p_3, hence p_3 is kept as nearest neighbor.

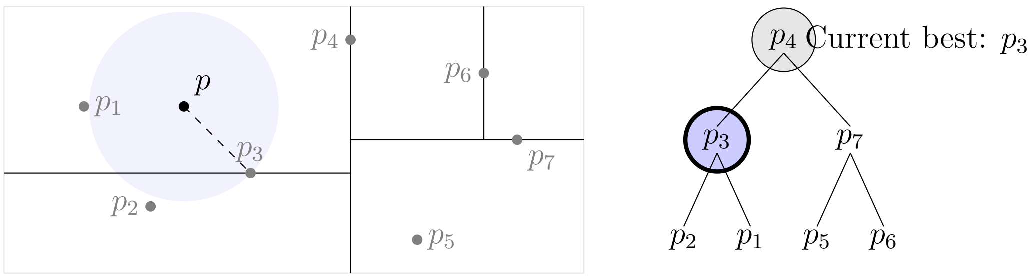


Figure 5.3: Traversal continues to p_3 which is new nearest neighbor.

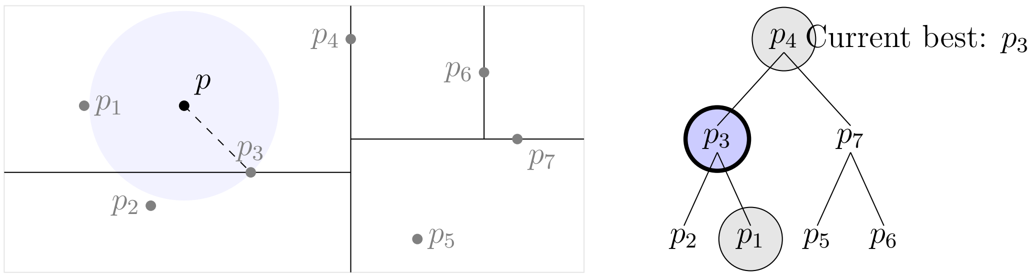


Figure 5.4: Traversal stops in leaf p_1, but keeps p_3 as nearest neighbor.

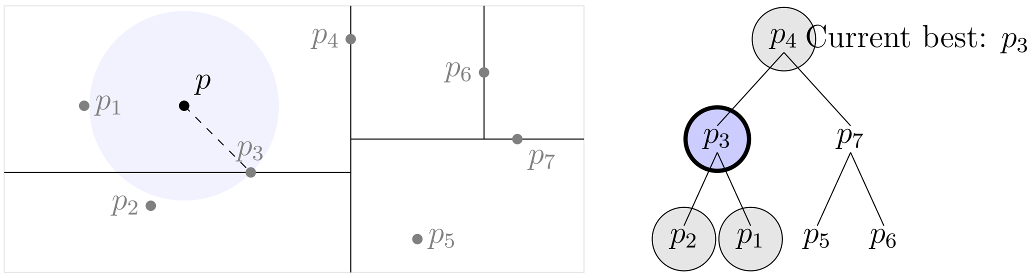


Figure 5.5: In the internal node p_3, its subtree containing p_2 is examined, but p_2 is further from p than p_3.

In the last Figure 5.6 we see that the traversal reached the root of the Kd-Tree again and examines the corresponding hyperplane at p_4. But since this hyperplane is further from p than its current nearest neighbor, the whole subtree to the right of p_4 is not examined anymore and recursion comes to a halt reporting p_3 as nearest neighbor of p.

As we saw in the example, the big advantage of the procedure outlined is that under the right circumstances a whole subtree can be discarded from

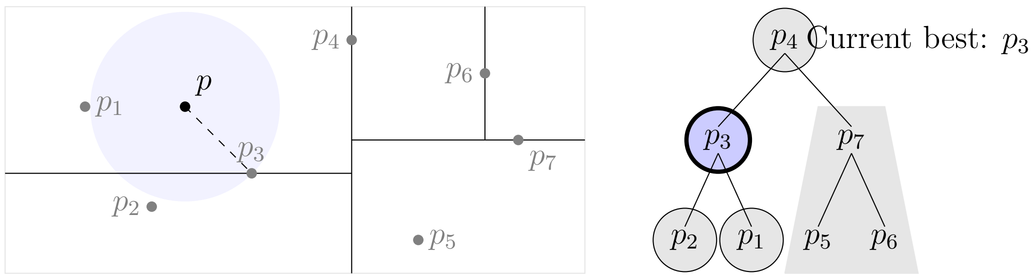


Figure 5.6: The right subtree of p_4 is rejected since the hyperplane through p_4 is further from p than the current nearest neighbor p_3.

the search. In general, if $|N_k(p_\iota)|$ is small compared to $|P|$ we expect to be able to discard many subtrees from the search. We make the description of our procedure explicit in Algorithm 9.

Note that in line 21 we do not have to check whether the hyperplane of r has distance less than ε to p, since this follows by transitivity from the condition in place and the fact that $\|L.\text{furthest} - p\| < \varepsilon$ by definition of L.
The only question remaining now is about the runtime of Algorithm 9. For this discussion we will consider the simplified case of $k = 1$ and ε large enough such that all points stored in the Kd-Tree are possible nearest neighbors. Furthermore assume that the Kd-Tree stores n points. Then at least $\mathcal{O}(\log n)$ visits of nodes in the tree are necessary, since the algorithm traverses down to leaf-level and the tree is balanced. On the other side, it can make at most n visits to nodes, since afterwards, the algorithm traversed every node in the Kd-Tree. The two important figures to look at here is the worst case run time as well as the average case run time. Figure 5.7 gives an example for a worst case example in which almost every leaf of the tree needs to be examined.

Concerning the average case runtime, we turn to [FBF77]. Within their paper they prove the following theorem.

Theorem 4. *The expected search time for the k nearest neighbors of a prespecified query point p in d-dimensional space is proportional to $\log n$, where n is the number of points stored in the Kd-Tree.*
In particular, the expected search time is independent of the distribution $\rho(P)$ of points in space.

Algorithm 9 Nearest Neighbor Kd-Trees

```
procedure NNKdTree($p$, $r$, $\varepsilon$, $k$) //Query point $p$, Root $r$ of the tree, influence $\varepsilon$, max. valence $k$
    $L \leftarrow$ empty list //List to store the neighbor candidates, gives furthest neighbor so far.
    return NNKdTreeRek($p$,$r$,$\varepsilon$,$k$,$L$)
end procedure

procedure NNKdTreeRek($p$,$r$,$\varepsilon$,$k$,$L$)//Query point $p$, current position $r$, influence $\varepsilon$, max. valence $k$, current best neighbors $L$
    if r == null then
        //The currentPosition is null, nothing can be done here, return the currently known nearest neighbors
        return $L$
    else
        //The current Position contains a point, since it is either a leaf or an internal node
        Extract the point $x_j$ from $r$ and store it in $L$. If $L$ becomes to large, delete furthest point in $L$.
        if $r$ is a leaf then
            return $L$
        else
            //If the currentPosition is not a Leaf, we can apply recursion to (possibly) both sides of the hyperplane
            NNKdTreeRek($p$,subtree of $r$, $\varepsilon$,$k$,$L$)
            if $|L| < k$ and $\|p - r.\text{hyperplane}\| < \varepsilon$ then
                //There are still neighbors missing and the other side of the hyperplane is still within influence radius
                NNKdTreeRek($p$,other subtree of $r$, $\varepsilon$,$k$,$L$)
            else if $\|L.\text{furthest} - p\| > \|p - r.\text{hyperplane}\|$ then
                //There are no neighbors missing, but we might find closer points to $p$ on the other side of the hyperplane
                NNKdTreeRek($p$,other subtree of $r$, $\varepsilon$,$k$,$L$)
            end if
            return $L$
        end if
    end if
end procedure
```

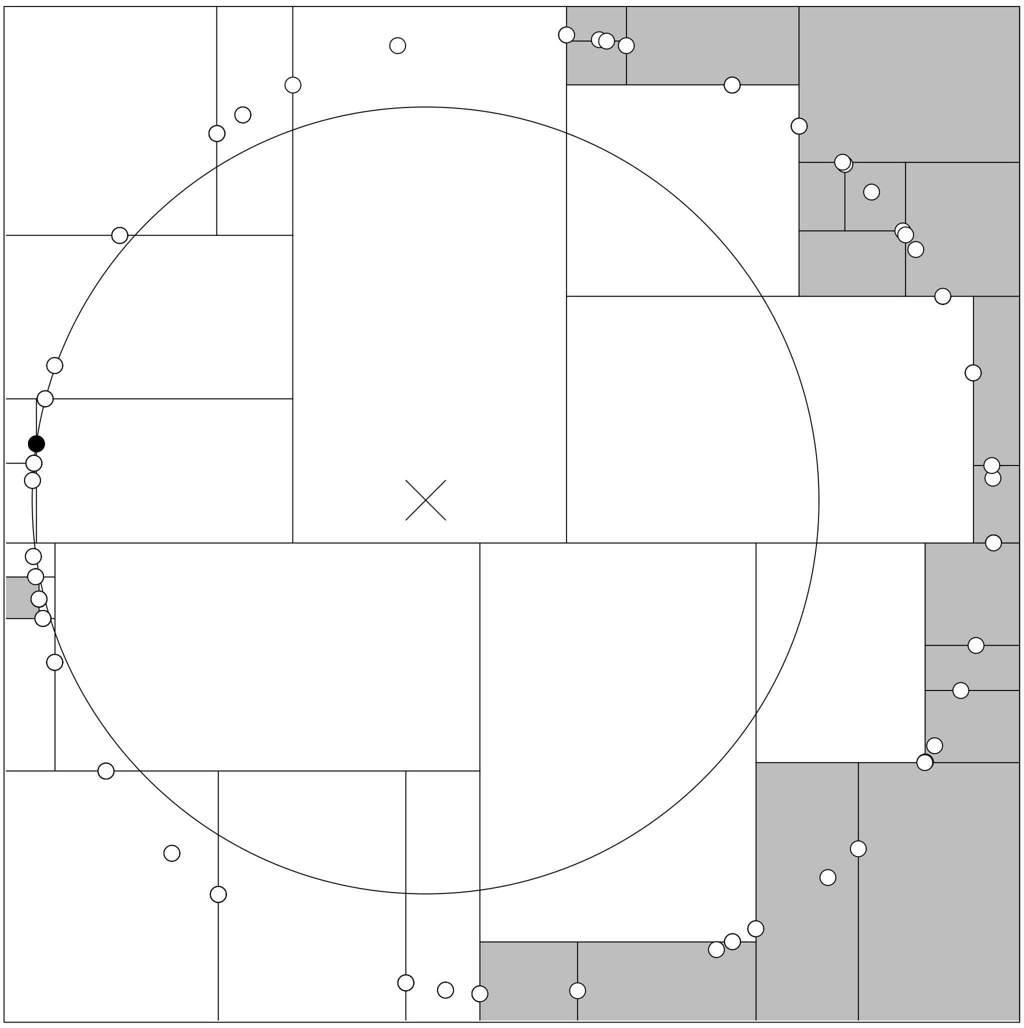


Figure 5.7: A figure from [Moo91], showing how a bad distribution of points can, for certain query points, lead to the necessity of examining almost all nodes of the Kd-Tree.

The general idea of the proof is to use the estimated size of the regions represented by the leafs of the Kd-Tree. The size of the region is derived from the number of points in P and the regions are fit into small d-dimensional hypercubes. When searching for nearest neighbors of the query point p, the algorithm traverses to the leaf representing the region that contains p. This takes $\mathcal{O}(\log n)$. Now the number of those hypercubes, which have to be investigated is estimated. It comes to be independent of the file size N and the probability distribution of the points in P, but only depends on k and d. Note at this point, that for large values of k and d, compared to n, the theorem becomes meaningless. For a thorough proof see [FBF77].

5.3 Implementation in JavaView

We will now briefly present the implementation of Algorithm 9 within the JavaView framework. The nearest neighbor computations are performed by the abstract Kd-Tree class as introduced in Section 3.3.2. Apart from the method given in Listing 3.1 the abstract Kd-Tree provides the methods given in Listing 5.1.

```java
//Nearest Neighbor for given k
public PriorityQueue<PdVector>
    getNearestNeighborsQueueByNumber(PdVector input, int
    count, Boolean includeInput);
public PiVector getNearestNeighborsVectorByNumber(PdVector
    input, int count, Boolean includeInput);
protected PriorityQueue<PdVector>
    getNumberOfNearestNeighbors(PdVector input, int count,
    PriorityQueue<PdVector> currentBest, Node
    currentPosition, Boolean includeInput);
//Nearest Neighbor Stack for given influence radius
public Stack<PdVector> getNearestNeighborsStackByInfluence(
    PdVector input, double influence, Boolean includeInput)
    ;
protected Stack<PdVector> getStackOfNearestNeighbors(
    PdVector input, double influence, Stack<PdVector> stack
    , Node currentPosition, Boolean includeInput);
//Nearest Neighbor Vector for given influence radius
public PiVector getNearestNeighborsVectorByInfluence(
    PdVector input, double influence, Boolean includeInput)
    ;
protected PiVector getVectorOfNearestNeighbors(PdVector
    input, double influence, PiVector vector, Node
    currentPosition, Boolean includeInput);
//Nearest Neighbor for given k and influence radius
public PriorityQueue<PdVector>
    getNearestNeighborsQueueByNumberAndInfluence(PdVector
    input, int count, double influence, Boolean
    includeInput);
public PiVector
    getNearestNeighborsVectorByNumberAndInfluence(PdVector
    input, int count, double influence, Boolean
    includeInput);
protected PriorityQueue<PdVector>
    getInfluencedNumberOfNearestNeighbors(PdVector input,
    int count, double influence, PriorityQueue<PdVector>
    currentBest, Node currentPosition,Boolean includeInput)
    ;
```

Listing 5.1: Methods provided by the abstract Kd-Tree class

All methods in Listing 5.1 compute nearest neighbors for an input point p, always given as a `PdVector`. However, three different ways to set up the neighborhood $N_k(p)$ are treated separately.

1. Case: The number k of neighbors is given. In the terms of Chapter 2, the influence radius ε is larger than $\max_{p_\iota, p_j \in P} \|p_\iota - p_j\|$. This means no point is neglected as possible nearest neighbor, because it is has distance larger ε to the input point.

2. Case: The influence radius ε is given. This means that all points $x_j \in P$ are reported as nearest neighbors of the input point p, if $\|x_j - p\| \leq \varepsilon$.

3. Case: Both a number k of neighbors and an influence radius ε is given. In this case, the neighborhood of the input point p is given by the intersection of the neighborhoods of Case 1 and Case 2.

For all three cases we furthermore provide two methods each. The first method provides a data structure which actually contains the nearest neighbors as `PdVectors`. This is either a `PriorityQueue<PdVector>`, provided with an appropriate comparator, or a `Stack<PdVector>`. For the documentation of these data structures, see [Ora]. The second method provided does not give the actual points, but a set of indices corresponding to these points. The result is provided in form of a `PiVector` from the JavaView framework. Although all six methods generally work in the same way, i.e. they construct the necessary data structure and call a **protected** recursive function to fill the data structure, we decided to rather provide six methods than one. This is mainly because the different used data structure can hardly be accessed by one common interface as they provide different functionality. Furthermore, the `PiVectors` could be obtained from the corresponding `PriorityQueue` or `Stack`, but this would involve iteration over the whole data structure. By providing a method that immediately returns a `PiVector`, we save this time. Since the last method `getInfluencedNumberOfNearestNeighbors` realizes the third case from above, i.e. incorporates the first and second case, we provide this method in Appendix E. The other methods are implemented similarly.

5.4 Alternative Pivot Policies for the Kd-Tree

The tree building algorithms from Section 3.3 require the median of the point set within a certain dimension to be found by either sorting or a more elaborate algorithm. Using the median, the point set is split, while the dimensions are iterated in a cyclic manner. The benefit of this method is the fact that the resulting Kd-Tree is always a balanced tree. That is, if n points are stored in the tree, traversing to a leaf takes time $\mathcal{O}(n \log(n))$. However, this pivot policy of splitting at the median and iterate through dimensions does not necessarily provide well shaped regions for nearest neighbor search. In the context of nearest neighbor search we aim for regions that are approximately of cubical form, which in Algorithm 9 minimizes the regions to be considered. However, the use of perfectly cubical regions degenerates the Kd-Tree to a Quad-, respectively Octree, which was dismissed in Section 4.1. In order to maintain a balanced tree, but still obtain well shaped regions, [Omo87] suggests to split the point set at the median, but not iterate through dimensions. The dimension is rather chosen in each recursion step as the dimension within which the spread of the point set is maximal. Although this seems to be a good approach, [Moo91] found that the obtained regions are still far from optimal, see Figure 5.8.

We saw that neither Quad-, respectively Octrees, nor balanced trees, even with the more sophisticated pivot policy of [Omo87] are a perfect choice for the nearest neighbor procedures. Note that this is mainly, because the data cannot be assumed to be uniformly distributed. If it was, the policy of [Omo87] would perform much better. As a consequence, [Moo91] suggests a different pivot policy that lies between the "median of the most spread dimension" policy and the usage of Quad-, respectively Octrees.
The proposed pivot policy splits the point set on the most split dimension, just as [Omo87] does. However, it is not split at the median, but at the point closest to the middle of the range along this dimension. That is the pivot element a is chosen from the point set P as

$$\begin{aligned} i &= \arg\max_{j=1,\dots,d} \left(\max_{p,q\in P} |p_i - q_i| \right), \\ m &= \left(\min_{p\in P} p_i + \max_{q\in P} q_i \right) /2, \\ a &= \arg\min_{p\in P} |p_i - m| . \end{aligned} \tag{5.1}$$

This pivot policy, according to [Moo91] does create only slightly unbalanced trees, while the regions of the Kd-Tree seem to be more cubical. The effect of this pivot policy is illustrated in Figure 5.9. An implementation is given in Appendix F.

In Chapter 8 we will compare the presented pivot policies.

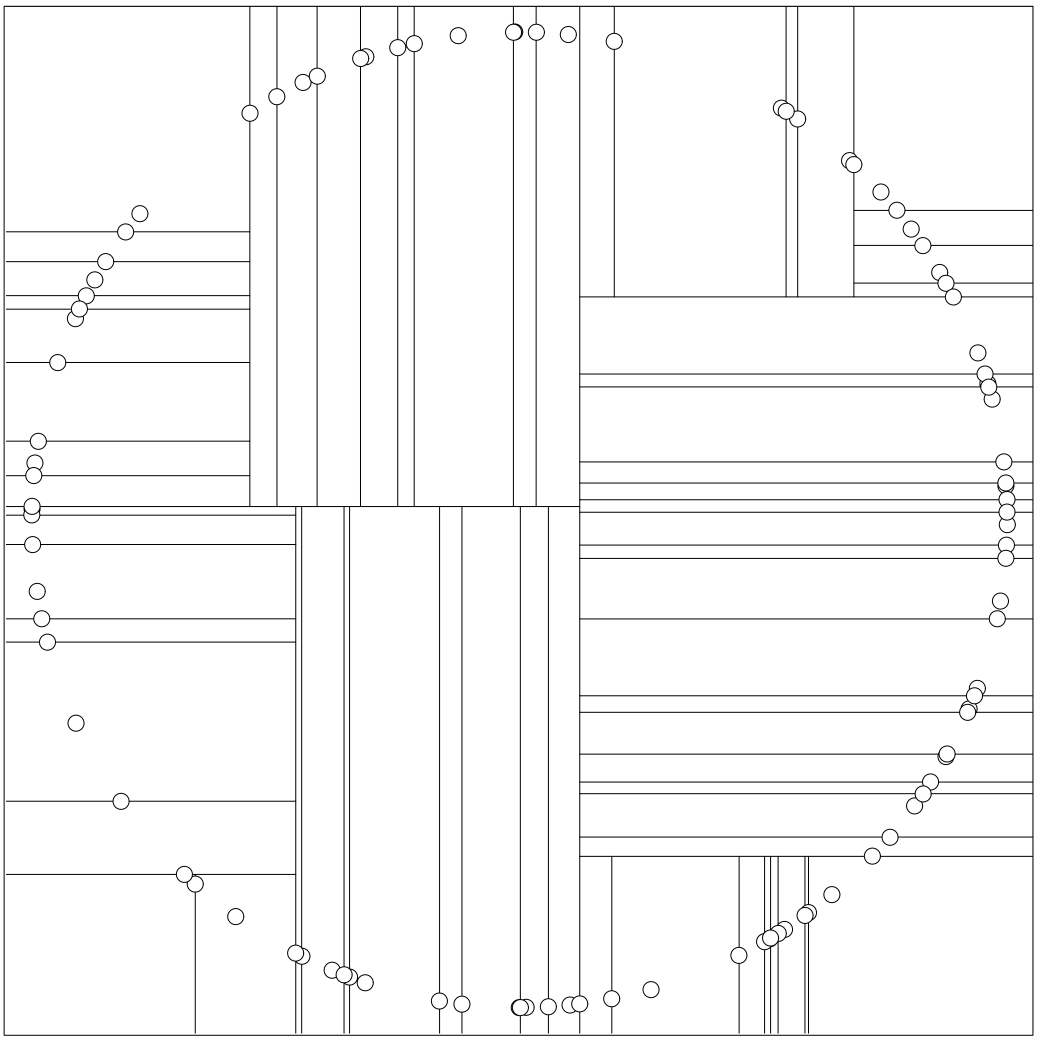


Figure 5.8: A figure taken from [Moo91], illustrating the short-comings of the ”median of the most spread dimension” pivot policy of [Omo87], namely the creation of many slim regions.

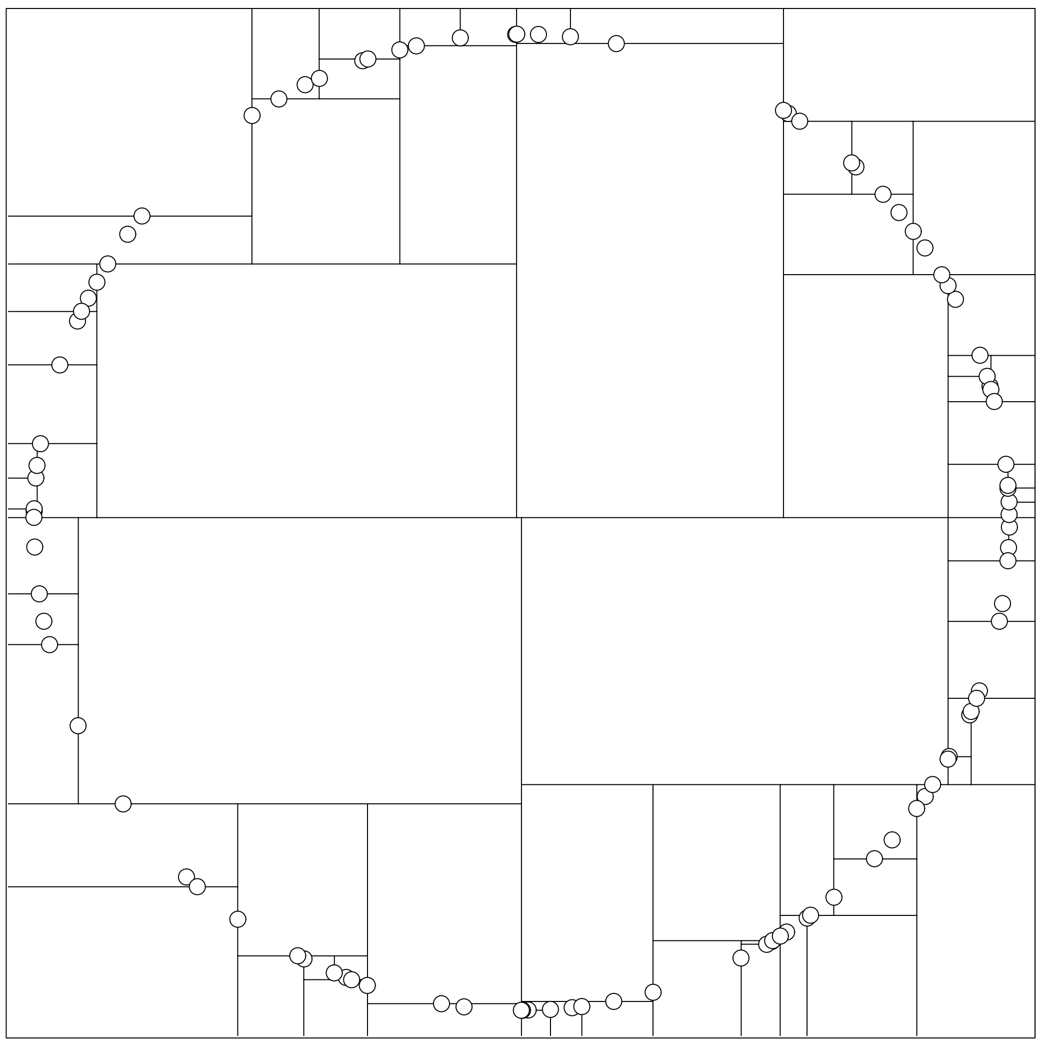


Figure 5.9: A figure taken from [Moo91], illustrating the alternative pivot policy "closest to the center of the most spread dimension", compare to Figure 5.8.

Chapter 6

Median

For a set of points P, we defined the median of P in Definition 3. The building process of Kd-Trees as given in Section 3.2 makes heavy use of the median. We already mentioned in Section 3.3.4 that finding the median in an ordered sequence is trivial. However, sorting a set P containing n elements comes with the cost of $\mathcal{O}(n \log(n))$. In this chapter, we will investigate alternatives to sorting.

6.1 Randomization

A good class of algorithms for finding the median are randomized algorithms. They are in general based on the idea of the quicksort algorithm. That is, a pivot element p is chosen from the set P and we partition P according to p by placing all elements smaller than p to the left of p and all elements larger than p to its right. Now it can be determined on which side of p the element lies and the algorithm recursively continues on that side of the set. Obviously this algorithm depends heavily on the choice of the pivot element p. Assume for example that p is chosen to be the minimum in every step, then we need to perform $(n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$ operations during the partition, i.e. the worst case runtime is $\mathcal{O}(n^2)$, even worse than sorting the set. A simple version of this idea is given in [SW11] and presented as Algorithm 10.

It is shown in e.g. [Cor+13] that Algorithm 10 has an expected runtime of $\mathcal{O}(n)$, that is an element of rank k, in particular the median, can be found in expected linear time. The algorithm can actually be made faster by not only picking one element at random, but three, and using the median of them as

Algorithm 10 Randomized Select

```
 1: procedure RandSelect($a_0, \dots, a_{n-1}$,$k$) //Sequence of element $a_i$, $k$th element is sought
 2:     if n=1 then
 3:         return $a_0$
 4:     else
 5:         Pick $x$ from the $a_i$ randomly
 6:         Partition the $a_i$ into $a_0, \dots, a_{q-1}$,$a_q, \dots, a_{r-1}$, and $a_r, \dots, a_{n-1}$ such that $a_i < x$ for $i = 0, \dots, q-1$, $a_i = x$ for $i = q, \dots, r-1$, and $a_i > x$ for $i = r, \dots, n-1$
 7:         if $k < q$ then
 8:             return RandSelect($a_0, \dots, a_{q-1}$,$k$)
 9:         else if $k < r$ then
10:             return $x$
11:         else
12:             return RandSelect($a_r, \dots, a_{n-1}$,$k - r$)
13:         end if
14:     end if
15: end procedure
```

pivot element. This is known as ”Median-of-Three” technique and can be found e.g. in [SW11].
The algorithm presented in [FR75] is structure-wise equivalent to Algorithm 10. The only difference here is the choice of the pivot element, which is chosen from a random sample. The algorithm from [FR75] was shown to have an optimal number of comparisons within lower-order terms, namely it finds the kth largest of n elements within

$$n + \min\{k, n-k\} + \mathcal{O}(\sqrt{n}).$$

However, the algorithm as given by [FR75] utilizes certain constants that have to be optimized depending on the machine that the algorithm is performed on. This might have been a good idea at the time when the algorithm was introduced, but nowadays an algorithm should perform well on any machine.

6.2 Deterministic Algorithm

We saw in the previous section that the median of a set can be determined in expected linear time. In this section we will present an algorithm with worst case behavior of $\mathcal{O}(n)$, see [Blu+73]. The idea is to use Algorithm 10, but pick the pivot element in a correct way. It turns out that the median of medians of five elements each does the job. Computing the $n/5$ medians of blocks of five elements from the original point set can be done in linear time, since finding the median of 5 points can be performed in constant time. Finding the median of these medians is now a recursive call to the method on a set with $n/5$ points. The obtained median of medians is then used to partition the set and the algorithm is recursively applied to the side of the median. See Algorithm 11.

Algorithm 11 Deterministic Select

```
procedure DetSelect($a_0, \dots, a_{n-1}$,$k$) //Sequence of element $a_i$, $k$th element is sought
    if $n < 140$ then
        Sort the sequence and return the median
    else
        Compute the $\lceil n/5 \rceil$ medians $m_0, \dots, m_{\lceil n/5 \rceil -1}$ of blocks containing five consecutive $a_i$ each
        $m$ ←DetSelect($m_0, \dots, m_{\lceil n/5 \rceil -1}$,$\lfloor \lceil n/5 \rceil /2 \rfloor$) //Recursively compute the median of the medians
        Partition the $a_i$ into $a_0, \dots, a_{q-1}$,$a_q, \dots, a_{r-1}$, and $a_r, \dots, a_{n-1}$ such that $a_i < m$ for $i = 0, \dots, q-1$, $a_i = m$ for $i = q, \dots, r-1$, and $a_i > m$ for $i = r, \dots, n-1$
        if $k < q$ then
            return DetSelect($a_0, \dots, a_{q-1}$,$k$)
        else if $k < r$ then
            return $m$
        else
            return DetSelect($a_r, \dots, a_{n-1}$,$k - r$)
        end if
    end if
end procedure
```

Call the median of medians a. To compute the runtime of this algorithm, we first give a lower bound on the number of elements larger than a. At least half of the $\lceil n/5 \rceil$ medians initially computed are larger or equal to a, therefore at least half of the $\lceil n/5 \rceil$ groups contributes three elements larger or equal to a, except a group with less than 5 elements (if n is not divisible by 5) and the group of the a itself. Therefore at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

elements is larger than a. Similarly at least $\frac{3n}{10} - 6$ elements are smaller than a. Therefore, after partitioning, the algorithm is called on at most $\frac{7n}{10} + 6$ elements. See Figure 6.1 for an illustration.

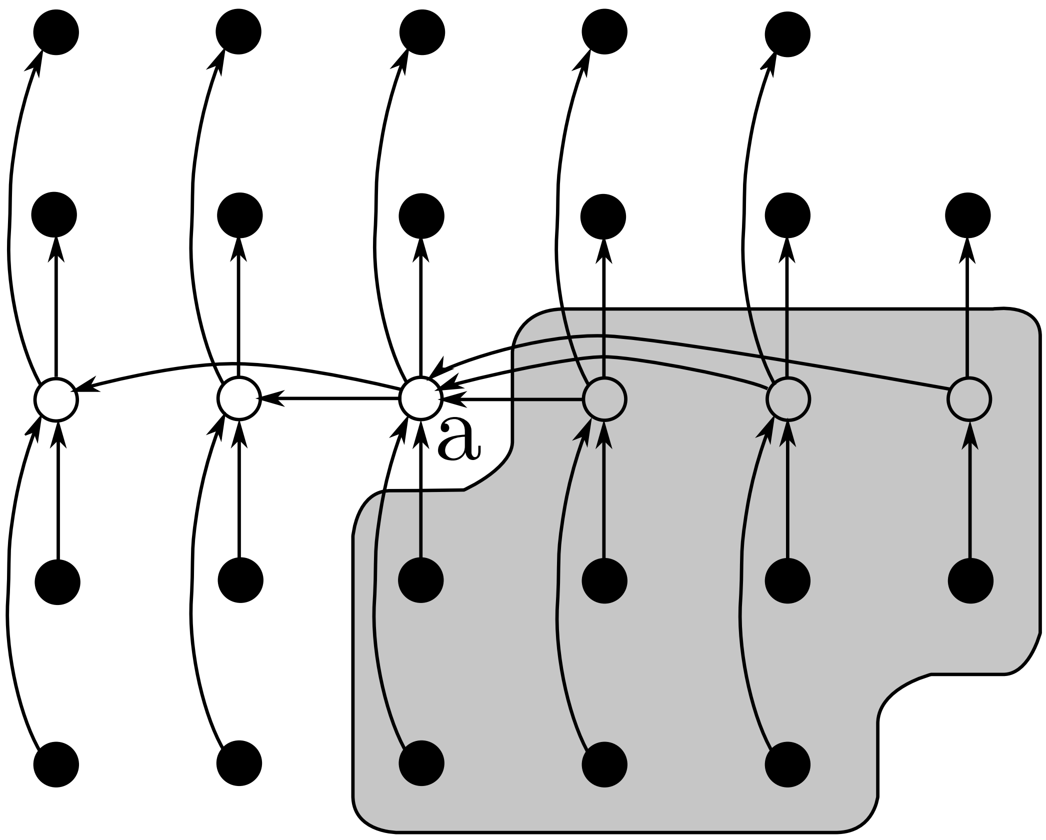


Figure 6.1: A set of 28 points. An arrow from point p to q indicates that p was found to be smaller than q. The marked point a is the median of medians. Note that all points in the gray region are necessarily smaller than a, which was established directly while computing the medians, or which is given by transitivity.

Determining the medians of the $\lceil n/5 \rceil$ groups and partitioning the point set takes $\mathcal{O}(n)$ time. If we denote the runtime of the algorithm by $T(n)$, then the recursive call to determine the median of medians takes $T(\lceil n/5 \rceil)$ and in recursion step on the larger partitioned part of the point set the algorithm takes at most time $T(7n/10+6)$. We now assume that the median of any set with $n < 140$ can be determined in $\mathcal{O}(1)$. This assumption seems rather arbitrary now, but the particular choice will make later calculation easier. Under the assumption, the runtime $T(n)$ satisfies the following recursion

$$T(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n < 140, \\ T(\lceil n/5 \rceil) + T(7n/10+6) + \mathcal{O}(n) & \text{if } n \geq 140. \end{cases} \tag{6.1}$$

To solve this recursion assume that $T(n) \leq c \cdot n$ for some large constant c and $n < 140$. This is indeed the case for c large enough. Furthermore we pick a constant a such that the non-recursive term in (6.1) for all $n > 0$ is bounded from above by $a \cdot n$. If we plug these assumptions into the right side of (6.1) we get

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10+6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an), \end{aligned}$$

which solves to be at most cn, if the inequality

$$-cn/10 + 7c + an \leq 0$$

is satisfied. For $n > 70$ this inequality is equivalent to $c \geq 10a(n/(n-70))$. Since we assume $n \geq 140$, it is $n/(n-70) \leq 2$. Picking $c \geq 20a$ therefore satisfies the inequality. Hence $T(n) \leq cn$.
Note that the choice of 140 is not necessary. Any integer larger than 70 can be chosen here. The presented calculations are taken from a thorough proof in [Cor+13]. Furthermore, the constant c in this algorithm is fairly large. For example, [Epp] computes it to be $c = 24$. But as final remark, [DZ99] proves that the median of n elements can be found using at most $2.95n$ comparisons.

6.3 Approximation

To close this chapter we will briefly mention the possibility of approximating the median. Using just any random element from the set P as approximation of the median will result in a fairly bad result. However, using e.g. the median of a small fixed sample is a better idea. Both approaches take only a constant number of steps and are therefore considerably faster than the methods given above. However, good results can only be expected, if the

sample size is bound to the size of the input set and therefore, constant time approximations will not perform very well.
A more elaborate way of approximating the median is presented in [RB90], where the median is approximated using a sequence of arrays. Data is sorted into the first array. As soon as it is filled, the median of this array is determined, saved in a second array and the first array is emptied. This procedure is iterated until all data has been processed. See Figure 6.2 for an illustration.

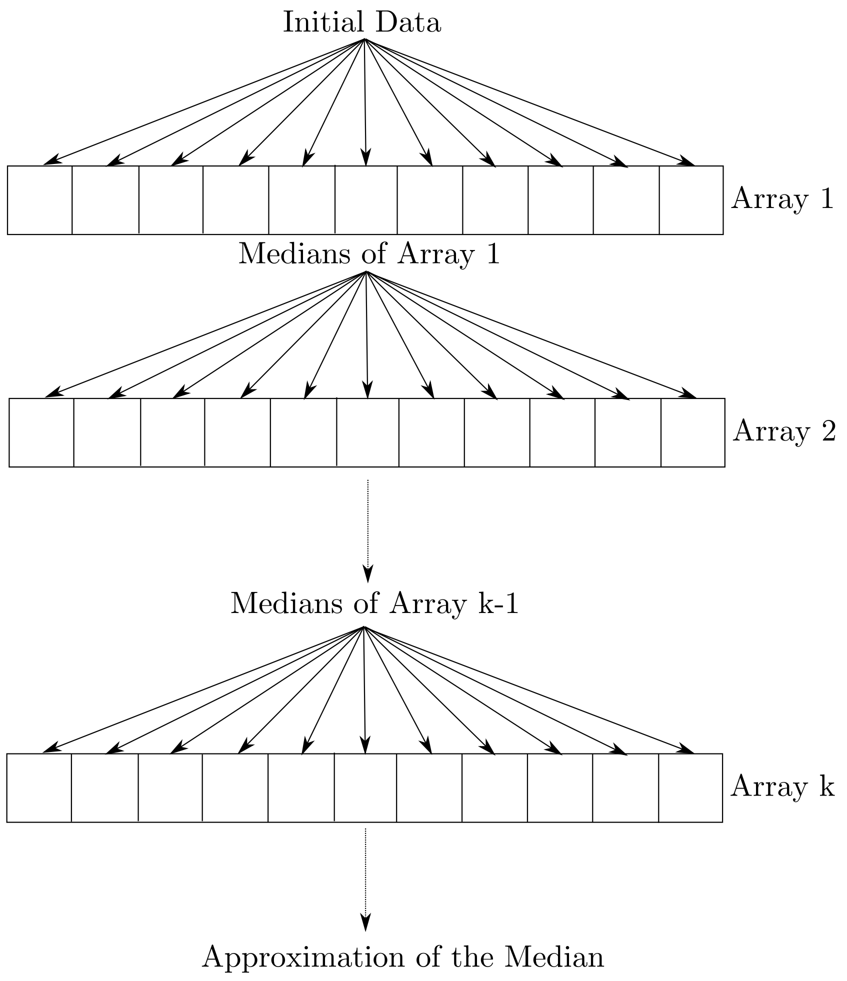


Figure 6.2: The process of computing the "Remedian" median approximation. The initial data is written into Array 1, until it is filled. Its median is stored in Array 2 and Array 1 is emptied, more initial data is written into it. This process is iterated over all Arrays and all initial data, until finally the median of the last used Array gives the approximation of the median.

An obvious downside of approximating the median within our application is the loss of balance of the Kd-Tree. Since in general an approximation of the median will have different numbers of elements smaller and larger than itself, the Kd-Tree resulting from the usage of median approximations will not be balanced. This will effect further applications on the Kd-tree as the nearest neighbor search. Hence, despite runtime advantages, we will not utilize approximation of the median.

Chapter 7

The Program

In this chapter we will briefly describe the program used in Chapter 8. It implements the techniques presented in Chapter 2 and Chapter 5. Implementation is done in Java using the JavaView Framework [Pol+]. We will here explain the general procedure of how to use the program, present an example and give details for all possible settings.

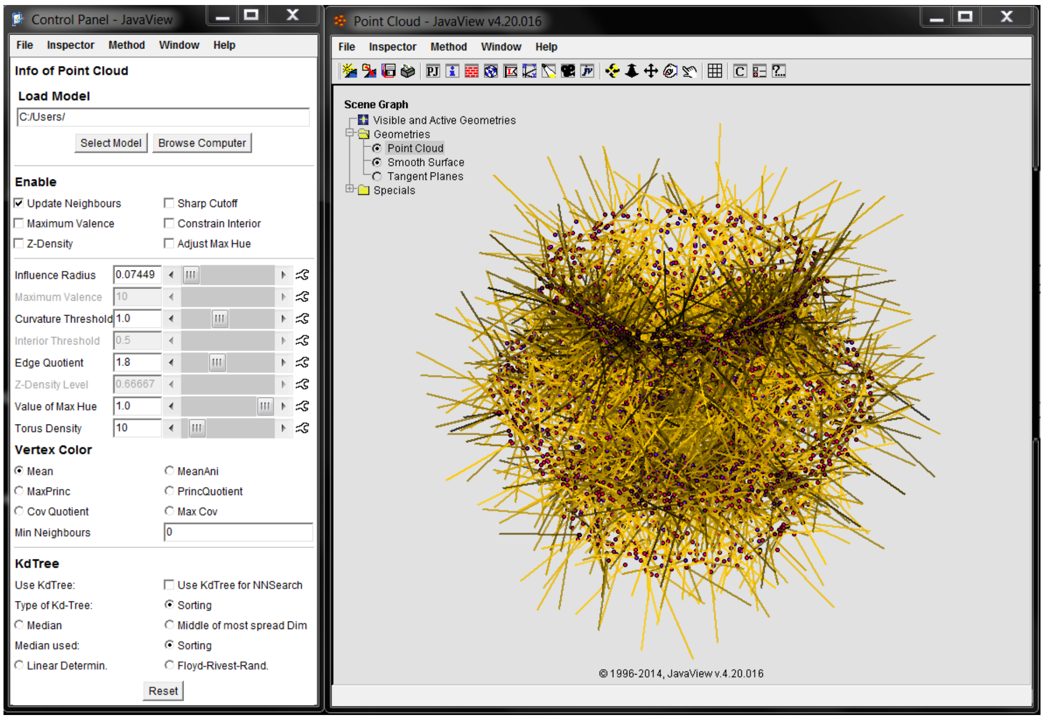
7.1 General Procedure

So far the program is available as a Java file `PaPointCloud` with a corresponding launch configuration `DevApps_PaPointCloud`. In Eclipse run the file using its launch configuration. The initial start up screen is shown in Figure 7.1.

Now start by selecting the entry "Smooth Surface" in the Scene Graph. Via the Load Model Dialog, browse the computer to load a model. Adjust the look of the geometries as intended. This can be done using the

$$\text{Inspector} \rightarrow \text{Geometry} \rightarrow \text{Material}$$

dialog. By loading a smooth surface the program immediately creates a point cloud. Select the corresponding entry "Point Cloud" in the Scene Graph. Now the smoothing process as given in Chapter 2 is controlled by several parameters. These can be set in the project panel. See Section 7.3 for details on these settings.
Obviously, one can already load a noised surface into the program. Assume that the loaded "Smooth Surface" is indeed smooth. To demonstrate the algorithm we now want to add noise to the surface. This can be done by


Control Panel - JavaView
File Inspector Method Window Help
Select Model
Browse Computer
Reset
Point Cloud - JavaView v4.20.016
File Inspector Method Window Help
Scene Graph
Visible and Active Geometries
Geometries
Point Cloud
Smooth Surface
Tangent Planes
Specials
© 1996-2014, JavaView v.4.20.016

Info of Point Cloud

Load Model: C:/Users/

Enable

- [✓] Update Neighbours
- [ ] Sharp Cutoff
- [ ] Maximum Valence
- [ ] Constrain Interior
- [ ] Z-Density
- [ ] Adjust Max Hue

Influence Radius	0.07449
Maximum Valence	10
Curvature Threshold	1.0
Interior Threshold	0.5
Edge Quotient	1.8
Z-Density Level	0.66667
Value of Max Hue	1.0
Torus Density	10

Vertex Color: [✓ Mean] MeanAni MaxPrinc PrincQuotient Cov Quotient Max Cov

Min Neighbours: 0

KdTree

Use KdTree: [ ] Use KdTree for NNSearch

Type of Kd-Tree: [✓ Sorting] Median Middle of most spread Dim

Median used: [✓ Sorting] Linear Determin. Floyd-Rivest-Rand.

Figure 7.1: The initial start screen of the program.

browsing to

$$\text{Method} \to \text{Effect} \to \text{Noise}$$

and applying noise to the smooth set using this dialog. To actually start the smoothing process, browse to

$$\text{Method} \to \text{Vector Field} \to \text{Evolve.}$$

In the shown dialog select the "Vertex Vector" option as well as the "Flip Direction" box. Finally hit the "Animate" button to see the smoothing procedure.

7.2 Example

We will now go through the program alongside an example. Assume that we want to apply the procedure as given in Section 7.1 to the geometry shown in Figure 7.2 using the settings as given in Listing 7.1.

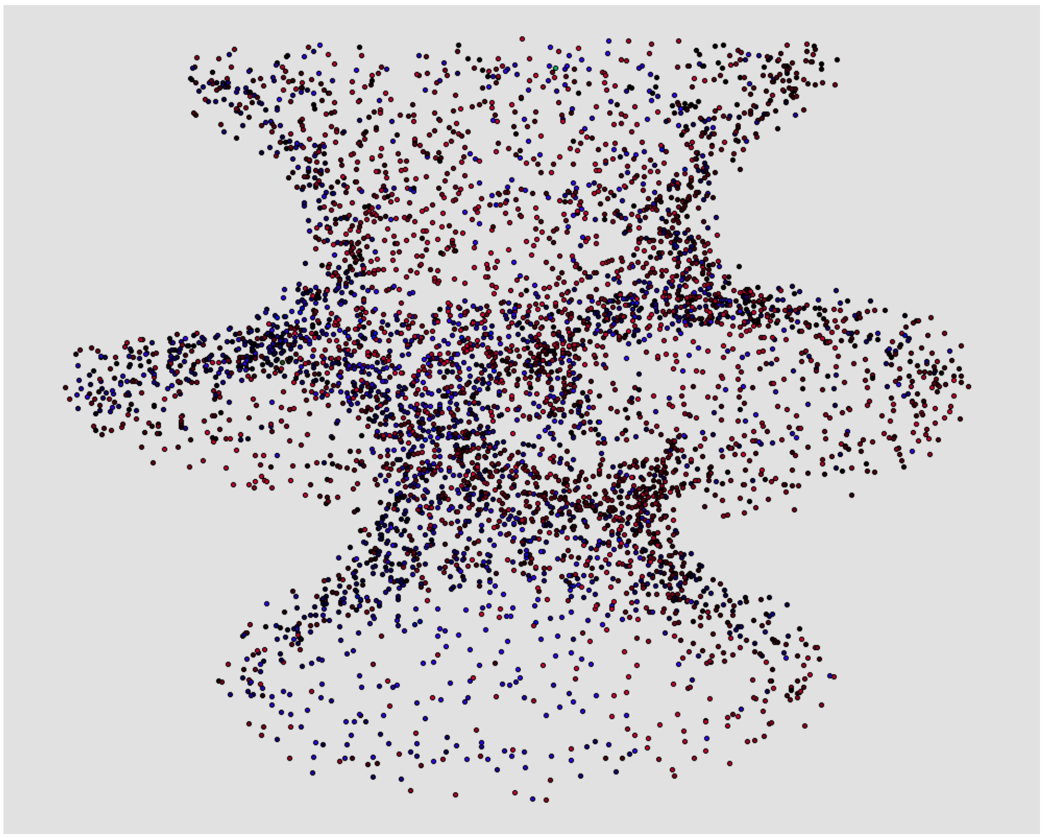


Figure 7.2: An example geometry to illustrate the program's abilities, namely a point set sample of the Costa surface.

```
Original neighbourhood of smooth surface
Vector Len = 0.5
Vector Siz = 2.

Noise:
- Nor,Tan
- Amplitude = 0.3

Evolve:
- Vector
- Offset 0. bis -0.49

Project:
Enabled:
  Max Valence

Density: 10
Influence: 1.0
Max Valence: 10
Curvature Thre: 1.0
Edge Quotient: 30.
Density Level: 0.71
```

```
  Value of Max Hue: 0.57533
```

Listing 7.1: A list of examplary settings to illustrate the program's abillities.

In Eclipse we start the `PaPointCloud` file using the launch configuration `DevApps_PaPointCloud`. Select the entry "Smooth Surface" in the Scene Graph. Via the Load Model Dialog, browse the computer to load the model "Costa4918Noise03OrigNeigh.jvx". To obtain the image shown in Figure 7.2, go to

$$\text{Inspector} \rightarrow \text{Geometry} \rightarrow \text{Material},$$

and deselect the checkbox "Vector Fields". Leave the Material dialog open. Select the entry "Point Cloud" in the Scene Graph. In the Material dialog adjust the Vector Length and the Vector Size to 0.5 and 2.0 respectively, as given above. Now display the Project Panel by selecting Inspector $\Rightarrow$ Project. According to the given description in Listing 7.1, mark the "Maximum Valence" field and set the "Influence Radius", "Maximum Valence", "Curvature Threshold", "Edge Quotient" and "Value of Max Hue" to the given values. Note that after entering the value into the text field, by pressing Shift+Enter, you can enter values that would be out of bounds for this field.
To apply noise open

$$\text{Method} \rightarrow \text{Effect} \rightarrow \text{Noise}.$$

As stated in the description, select both "Normal" and "Tangential". Set the Amplitude to 0.3. The Point Set now has noise. Leave the Noise dialog by clicking "ok" and open

$$\text{Method} \rightarrow \text{Vector Field} \rightarrow \text{Evolve}.$$

Switch to the direction "Vertex Vector". Mark the "Flip Direction" box. Press "Animate" to start the smoothing process. Different stages of the animation are shown in Figure 7.3.

7.3 Setting Details

In this section we will briefly explain the different setting possibilities of the program. First the six checkboxes on top of the project inspector, shown on the left of Figure 7.1.

- `Update Neighbors`: Every change of the point set geometry triggers a new computation of the neighborhoods.

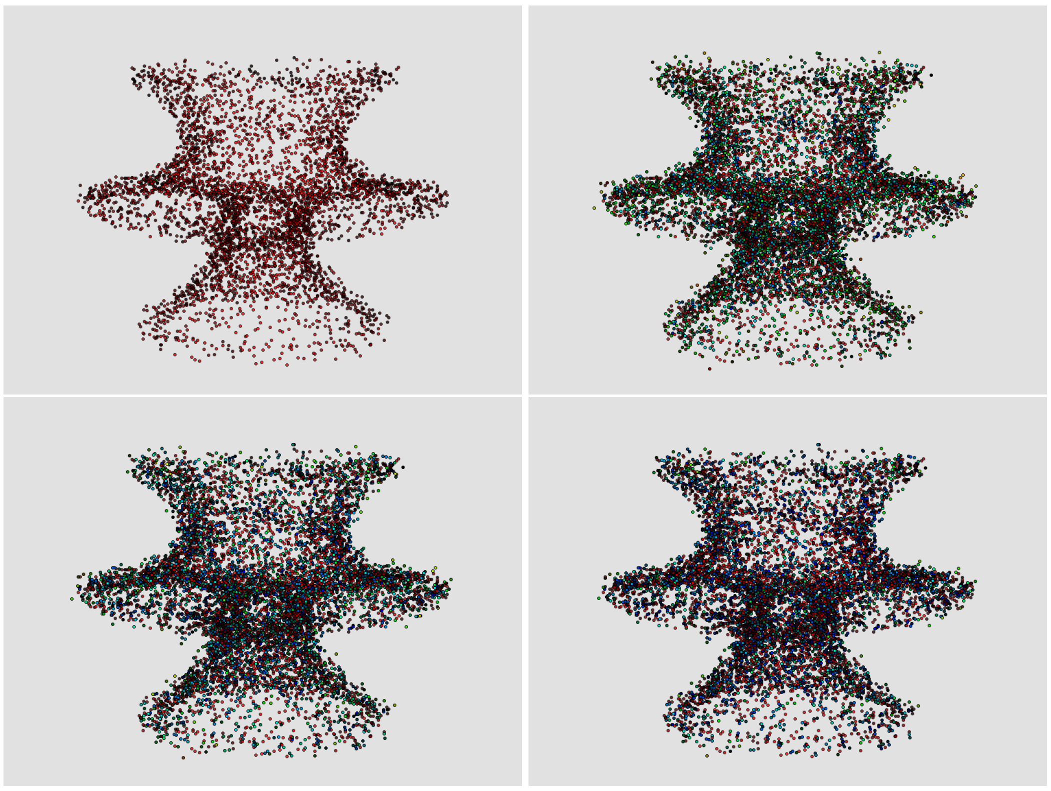


Figure 7.3: The images show from left to right and top to bottom: The original point cloud, the point cloud with noise applied, the smoothing process with animation offset 0.2, and the smoothing process with animation offset 0.4. Note how the points become darker during the animation, indicating that their position becomes more and more fixed.

- `Maximum Valence`: Not all points within influence radius of the input contribute to the neighborhood, but only as many as the parameter maximum valence is set to (see below).

- `Z-Density`: The point set is dissolved along the z-axis according to the `Z-Density Level` set (see below).

- `Sharp Cutoff`: Utilize the function $g_{\iota j}^{\text{sharp}}$ as defined in (2.31) within the anisotropic Laplacian. $g_{\iota j}$ then only depends on the `Edge Quotient` (see below).

- `Constrain Interior`: Depending on the `Interior Threshold`, certain points are considered to be point on the interior of a flat surface and are not moved by the smoothing procedure.

- `Adjust Max Hue`: Automatically set the `Value of Max Hue` parameter.

Apart from these options, several parameters are to be chosen using sliders. The parameters are

- `Influence Radius`: This is the parameter called ε in Chapter 2. It determines the maximum distance of neighbors.
- `Maximum Valence`: This is the parameter called k in Chapter 2. It is only active if the `Maximum Valence` checkbox is set. It limits the size of each neighborhood.
- `Curvature Threshold`: This is the parameter called λ in Chapter 2. It determines when a point is considered to be a feature which is not to be smoothed as in (2.31) and (2.32).
- `Interior Threshold`: If the `Constrain Interior` checkbox is selected, this parameter determines when a point is considered to be in the interior of a flat surface.
- `Edge Quotient`: This is the parameter called Q in Chapter 2. As defined in (2.33), it incorporates principal curvature directions into the anisotropic smoothing process.
- `Z-Density Level`: Density of the point cloud along the z-axis. Effect is illustrated in Figure 7.4.
- `Values of Max Hue`: Maximum Hue of the different coloring schemes as listed below.
- `Torus Density`: Changing this parameter creates a Torus as smooth surface of set density.

Next, the vertex color can be chosen. The program offers the following possibilities. They are illustrated in Figure 7.5.

- `Mean`: Colors the vertices according to the mean curvature.
- `Max Princ`: Colors the vertices according to the maximum principal curvature.
- `CovQuotient`: Colors the vertices according to the covariance quotient.
- `MeanAni`: Colors the vertices according to the mean curvature, considering the anisotropic cut-off.

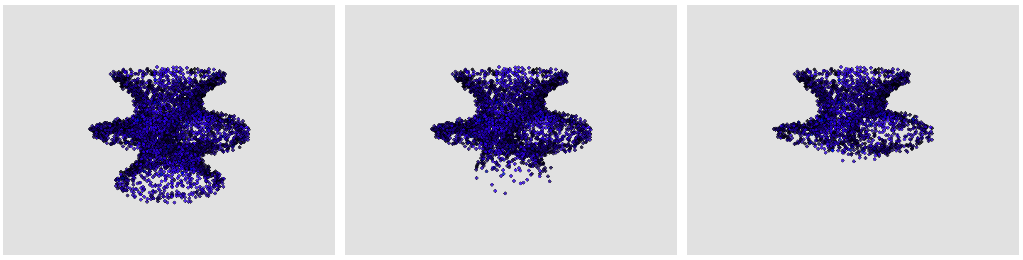


Figure 7.4: The effect of the Z-Density Level parameter on the point cloud of the Costa surface.

- `PrincQuotient`: Colors the vertices according to the principal curvature quotient.
- `MaxCov`: Colors the vertices according to maximum covariance.

Below the vertex color, the program shows one text field.

- Min Neighbors: This field shows the smallest neighborhood according to the given parameters, i.e. all neighborhoods are computed according to the `Influence Radius` and `Maximum Valence`. The size of the smallest neighborhood is shown in the textfield. This enables easy set up of good parameters for a given geometry.

The last settings concern the use of Kd-Trees.

- `Use Kd-Tree for NNSearch`: This enables the use of Kd-Trees for nearest neighbor search rather than the covariance method given in Section 5.1.

Three types of Kd-Trees can be chosen.

- `Sorting`: The Kd-Tree initially sorts the points and passes the sorting on during its recursive building process. See Section 3.3.3.
- `Median`: The Kd-Tree computes the median in each step of the recursive building process. The method for median computation is specified below. See Section 3.3.4.
- `Middle of most spread Dim`: The Kd-Tree splits at the point closest to the middle of the most spread dimension as given in Section 5.4.

In case of the Median-Kd-Tree being utilized, the type of median finding algorithm can be specified from the following options.

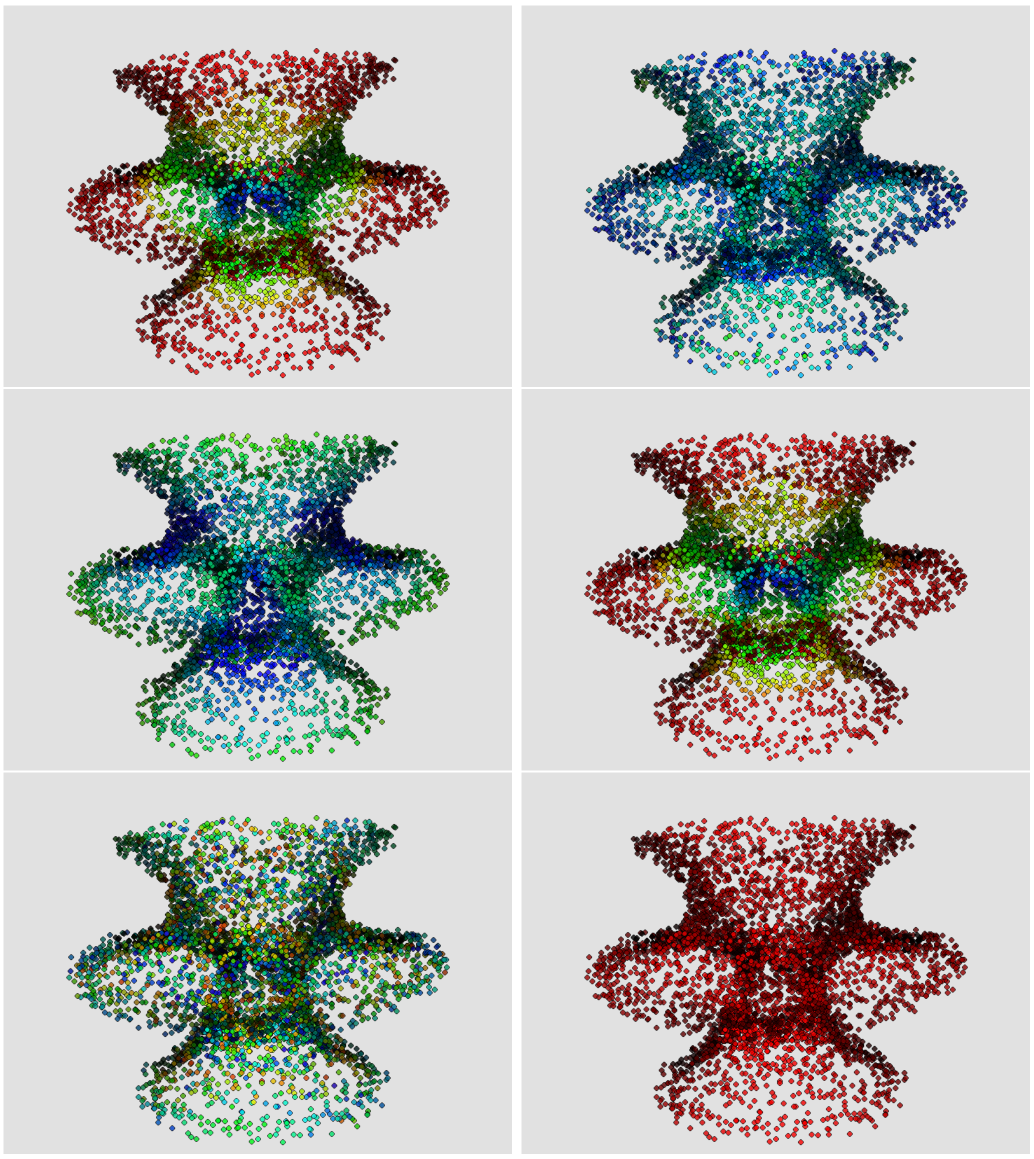


Figure 7.5: The effect of the different vertex colorings, from left to right and top to bottom: Mean, MaxPrinc, CovQuotient, MeanAni, PrincQuotient, MaxCov.

- `Sorting`: The set is sorted an the $\lfloor n/2 \rfloor$-th element is given as median.
- `Linear Determin.`: The median is computed using the deterministic linear-time algorithm presented in Section 6.2.
- `Floyd-Rivest-Rand.`: The median is computed using the randomized algorithm of Floyd and Rivest as presented in Section 6.1.

Chapter 8

Computational Results

In the previous chapters we presented several techniques for nearest neighbor search. In this chapter we would like to test the presented methods within the JavaView framework. The test will be performed on three point set geometries, referred to as Costa, Cylinder and Torus. The geometries are shown in Figure 8.1. By subdivision using the algorithm of Catmull-Clark, the number and complexity of the geometries is increased. Figure 8.2 shows the different complexities used in the benchmarks.

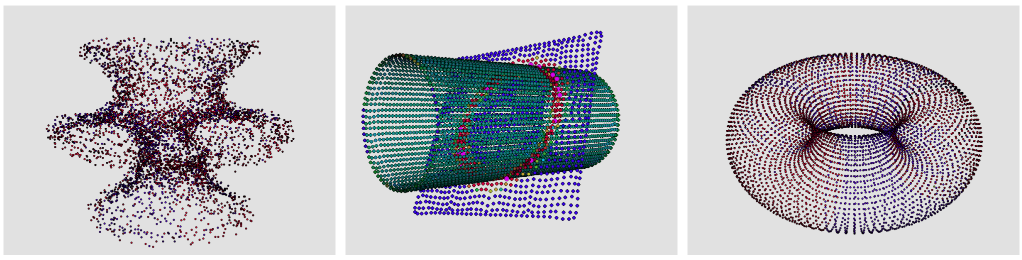


Figure 8.1: From left to right: A sampled version of the Costa surface, a cylinder intersected with a plane segment, and a torus.

Geometry	#Vertices	1 subd.	2 subd.
Costa	4918	28311	112041
Cylinder	5249	30977	123393
Torus	4851	19404	77616

Figure 8.2: The different geometries and their number of vertices originally and after one, respectively two subdivisions.

In [SS] it was shown that the benefit, for the building times of a Kd-Tree and nearest neighbor computation times, of storing a general point p' as defined in (3.3) is marginal compared to computing it on the fly from the original vertex p of the point set. Since our setup focuses on building Kd-Trees for nearest neighbor searches, we therefore compute all composite coordinates from (3.2) when needed. This allows us to store only the d-dimensional points $p \in P$ and not $d \times d$-matrices p'. Although it does not affect our setup, note that computing composite coordinates for each point during the building process did prove to be beneficial in the application of region queries, see [SS]. The corresponding experimental results, taken from [SS], are given in Figure 8.3.

Task	Type	Data	Duration	Result
Build	Store	Average	3070ms	On average the Storing
		Median	3202ms	classes took 114ms
	Compute	Average	3184ms	longer to be built, i.e.
		Median	3307ms	3.7% longer.
Region query	Store	Average	4669ms	On average the Computing
		Median	4867ms	classes took 3975ms
	Compute	Average	8644ms	longer to perform the
		Median	7570ms	query, i.e. 85.1% longer.
Nearest Neighbor Query	Store	Average	273379ms	On average the Computing
		Median	198103ms	classes took 5408ms
	Compute	Average	267971ms	longer to perform the 2%
		Median	194846ms	query, i.e. 2% longer.

Figure 8.3: The shown data is taken from [SS] and was established by performing 100 build operations, 1000 region queries and 1000 nearest neighbor searches on a geometry with 69649 vertices. The two compared Kd-Tree implementations either determine all composite coordinates from Definition 4 during the building process and save them (Store) or determine the coordinates on the fly when needed in queries (Compute).

8.1 Building Times and Tree Depths

At first we will compare the times for the building process of the different Kd-Tree implementations developed throughout this thesis. We will refer to the Kd-Tree as presented in Section 3.3.3 by "Sorting". The Kd-Trees as presented in Section 3.3.4 will be denoted by "Median", followed by a specification of the used algorithm as presented in Chapter 6. Namely these will

be "Median (Sorting)", "Median (Lin.Det)" (see Section 6.2), and "Median (Floyd-Rivest)" (see Section 6.1).
To avoid side effects from other processes on the computer, the trees were build 1000 times each on all three geometries. The obtained data can be found in Figure 8.8. Furthermore, the building times for the original geometries are plotted in Figure 8.4, while the times for the once and twice subdivided geometries are plotted in Figure 8.5 and Figure 8.6 respectively.

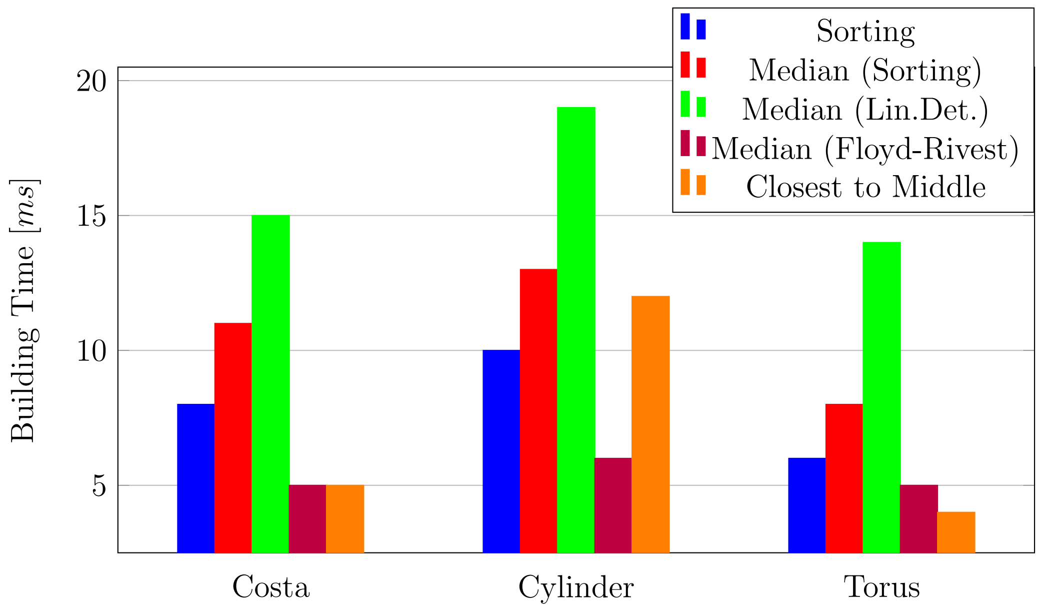


Figure 8.4: Building time of the different Kd-Tree implementations on the original geometries as shown in Figure 8.1.

We see that the Kd-Tree implementation, which determines the median using the randomized algorithm of Floyd and Rivest has the shortest building time. Furthermore note that the construction of a Kd-Tree that uses the "Closes to middle of most spread dimension" pivot rule (see Section 5.4) takes up to ten times as long as the construction of the "Median (Floyd-Rivest)" Kd-Tree.
We know that all Kd-Trees set up by "Sorting" or one of the "Median" implementations have the same height on each geometry, since these implementations always create a balanced AVL Tree. However, the "Closest to Middle" implementation does not necessarily create a balanced tree. Therefore we monitored the depth of the resulting Kd-Tree on the different geometries. The numbers are given in Figure 8.7. They already suggest that any operation on these trees will be costly due to the high depths of the trees.

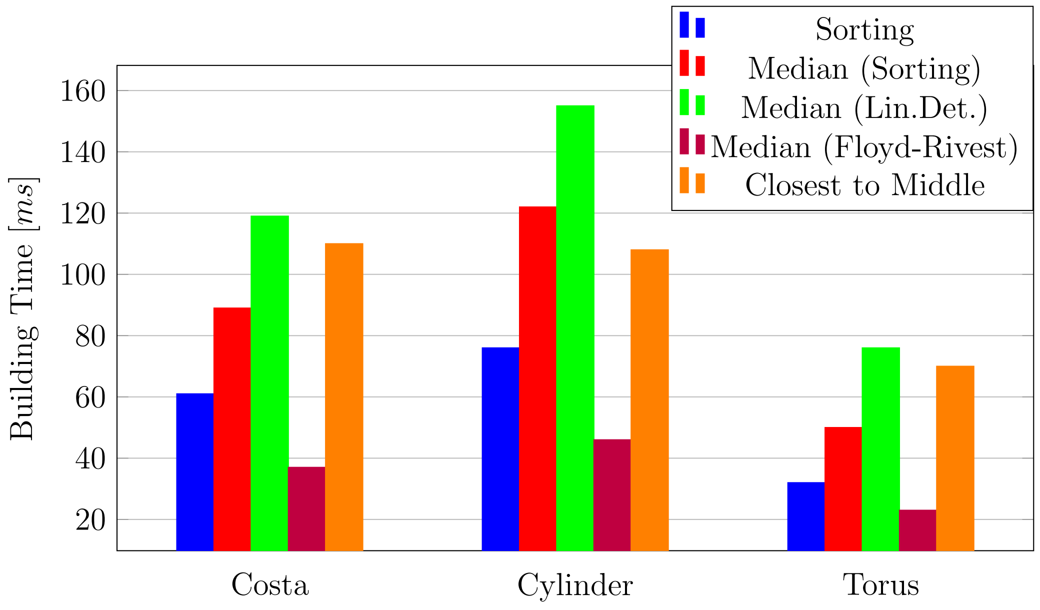


Figure 8.5: Building time of the different Kd-Tree implementations on the geometries, each subdivided once using Catmull-Clark.

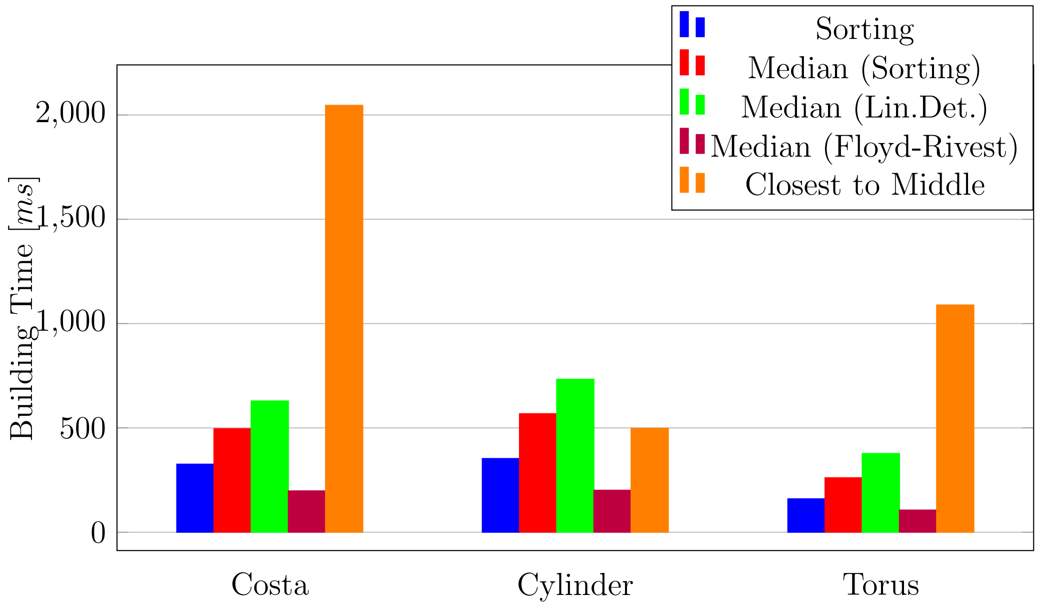
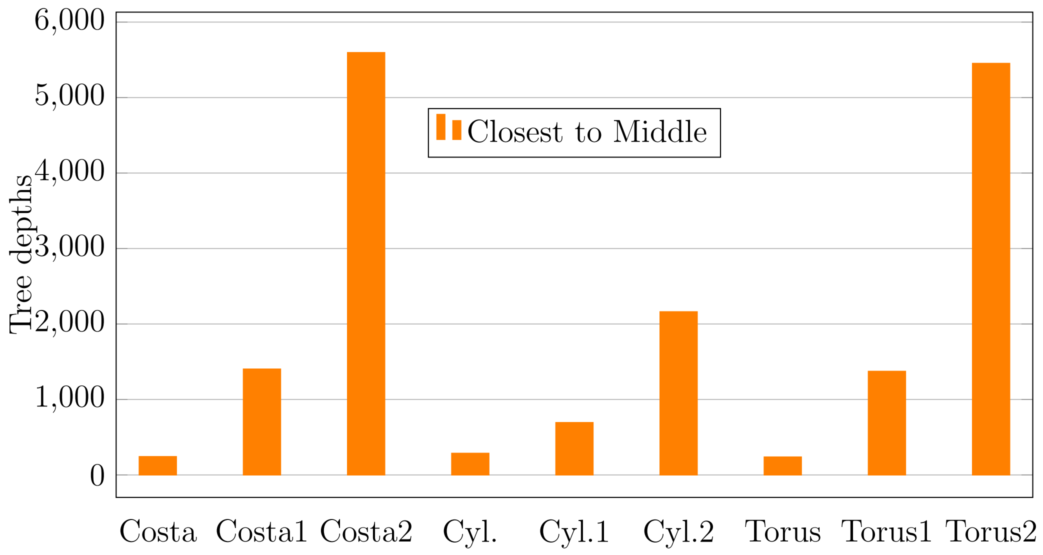


Figure 8.6: Building time of the different Kd-Tree implementations on the geometries, each subdivided twice using Catmull-Clark.



	Costa	Costa1	Costa2
Balanced	12	14	16
Closest to Middle	243	1403	5596

	Cylinder	Cylinder1	Cylinder2
Balanced	12	14	16
Closest to Middle	287	694	2162

	Torus	Torus1	Torus2
Balanced	12	14	16
Closest to Middle	238	1373	5453

Figure 8.7: Development of the depth of the Kd-Trees obtained using the "Closest to middle of most spread dimension" pivot strategy on the original geometries as well as on their respective subdivisions.

8.2 Kd-Tree vs. PCA

We will now turn to nearest neighbor search. First we will compare the performance of our Kd-Tree implementation with the nearest neighbor algorithm using PCA as given in Section 5.1. In order to do so, we let both, the Kd-Tree "Sorting" and the PCA algorithm run on the original Cylinder

	Costa		Costa 1		Costa 2	
	Average	Median	Average	Median	Average	Median
Sorting	9.21	8	62.50	61	328.61	326
Median (Sorting)	10.73	11	90.57	89	498.18	496
Median (Linear Det.)	15.78	15	119.91	119	630.47	629
Median (Floyd-Rivest)	5.41	5	37.33	37	200.32	198
Closest to Middle	5.07	5	110.53	110	2057.34	2046
	Cylinder		Cylinder 1		Cylinder 2	
	Average	Median	Average	Median	Average	Median
Sorting	10.36	10	77.46	76	355.84	353
Median (Sorting)	13.08	13	123.64	122	588.76	586
Median (Linear Det.)	19.81	19	155.64	155	735.98	733
Median (Floyd-Rivest)	6.46	6	46.20	46	203.67	201
Closest to Middle	11.66	12	109.31	108	499.23	498
	Torus		Torus 1		Torus 2	
	Average	Median	Average	Median	Average	Median
Sorting	6.62	6	32.71	32	162.29	160
Median (Sorting)	8.17	8	50.82	50	263.01	261
Median (Linear Det.)	14.90	14	76.97	76	377.20	377
Median (Floyd-Rivest)	5.45	5	23.77	23	107.40	106
Closest to Middle	4.57	4	71.11	70	1092.47	1089

Figure 8.8: Building times of the different Kd-Tree implementations. All times are given in ms. The columns Costa, Cylinder, Torus give the times for the original geometries, while the other columns give the times for the once, respectively twice subdivided geometries. The numbers are plotted in Figures 8.4, 8.5, and 8.6.

geometry. We choose some influence radius ε in such a way that all points of the geometry are considered in every neighborhood. Then we vary the `Max Valence` parameter from $k = 1$ to 400. That is, in each run, both the Kd-Tree and the PCA method try to find k nearest neighbors to each point p in the point set. Results are given in Figure 8.9.

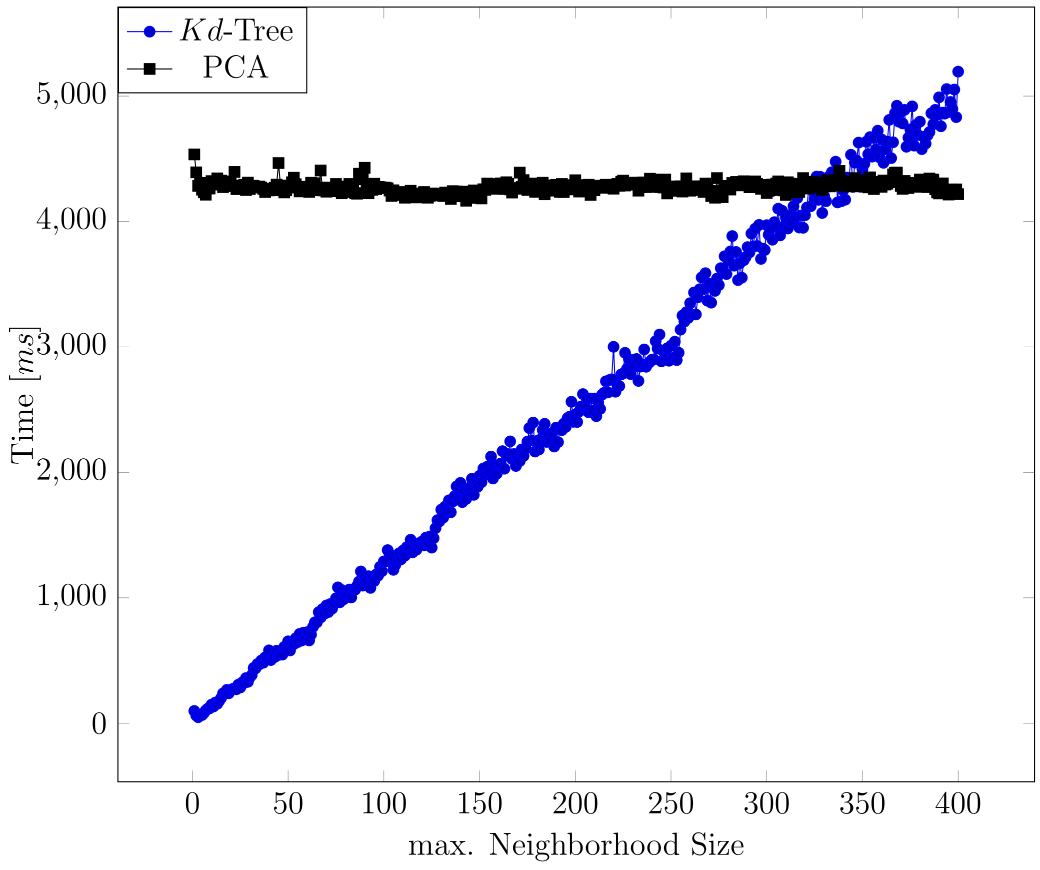


Figure 8.9: Neighborhood computation times for different maximal neighborhood sizes and a fixed (large) influence radius. The Kd-Tree implementation shown is "Sorting".

Recall that the geometry used in Figure 8.9 has 5249 vertices, hence our neighborhood computation includes neighborhoods of size up to 7.62% of the whole geometry. In any practical application, the neighborhoods will be a lot smaller, i.e. a small two-digit constant. Therefore from Figure 8.9 we already see that the Kd-Tree method improves neighborhood computation times drastically. Up to a maximum neighborhood size of 100, the Kd-Tree is more than 3.3 times faster than the PCA method.

We will now alter the setup of our experiment slightly. While before we fixed an influence radius and varied the maximum neighborhood size, we will now do the opposite. There will be no restrictions on the size of the neighborhood and we will vary the influence radius. The cylinder geometry has a diameter of 2.09, which is the length of the diagonal of the smallest axis-parallel bounding box of the geometry. We will vary the influence radius from 0.002 up to 0.33, which is 15% of the diameter of the geometry. Again, in any practical application, the influence radius will be chosen smaller. During the experiment, in each run, both the Kd-Tree and the PCA method try to find all neighbors to each point p in the point set, that lie within influence radius. Results are given in Figure 8.10.

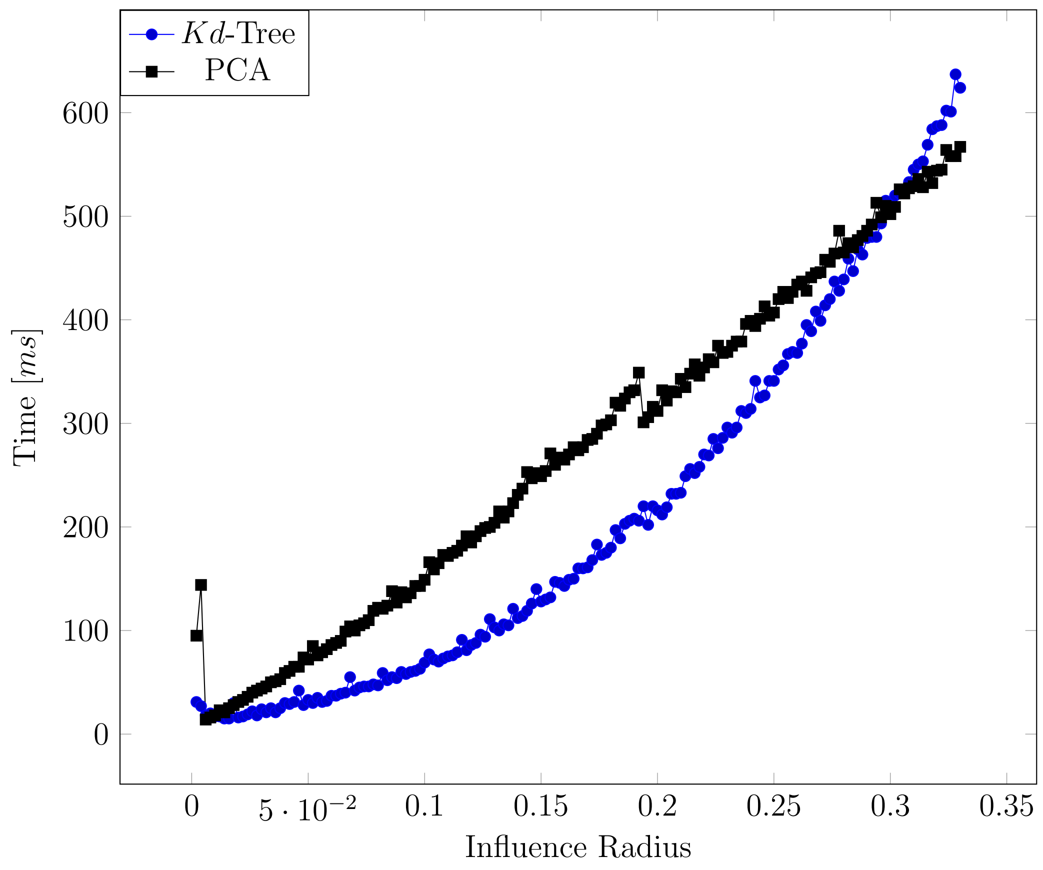


Figure 8.10: Neighborhood computation times for different influence radii and no restriction on the neighborhood size. The Kd-Tree implementation shown is "Sorting".

As for the different valences, also when varying the influence radius, our Kd-

Tree implementation performs better than the PCA-approach within the given range. In the following we will therefore restrict our experiments to a maximum neighborhood size of $k \in [0, 300]$ and an influence radius ε of up to 15% of geometry's diameter.

8.3 "Middle of most spread Dim." vs. Median

Recall that the implementations "Sorting" and "Median" (independent of the used median algorithm) produce the same balanced Kd-Tree. Solely the Kd-Tree implementation utilizing the "Closest to middle of most spread dimension" pivot rule does produce a different Kd-Tree. In this section we will perform several nearest neighbor queries on these two types of Kd-Trees to determine which approach provides faster results.
At first we perform the same experiment as we did on the Kd-Tree and the PCA method. That is, we use both the "Sorting" and the "Closest to Middle" implementation to determine neighborhoods of different size on the Cylinder. Neighborhood sizes are varied from $k = 1$ to 100. As above, we set the influence radius to a high value that ensure all points to qualify for possible neighborhood membership. The results of this experiment are plotted in Figure 8.11.

We repeat the experiment in a similar fashion, except this time, we run it on the original Costa surface and we repeat it five times to exclude any interference with other processes. Figure 8.12 shows for each neighborhood size the median of the obtained running times.

To finish the comparison between the balanced Kd-Trees and the trees obtained by the "Closest to middle of most spread dimension" pivot rule, we repeat the second experiment from Section 8.2. On the original torus geometry and its once subdivided form, we do not impose any restrictions on the size of the neighborhood and vary the influence radius in 100 steps from one per mill of the geometry's diameter to 10% of the geometry's diameter. Results are given in Figure 8.13.

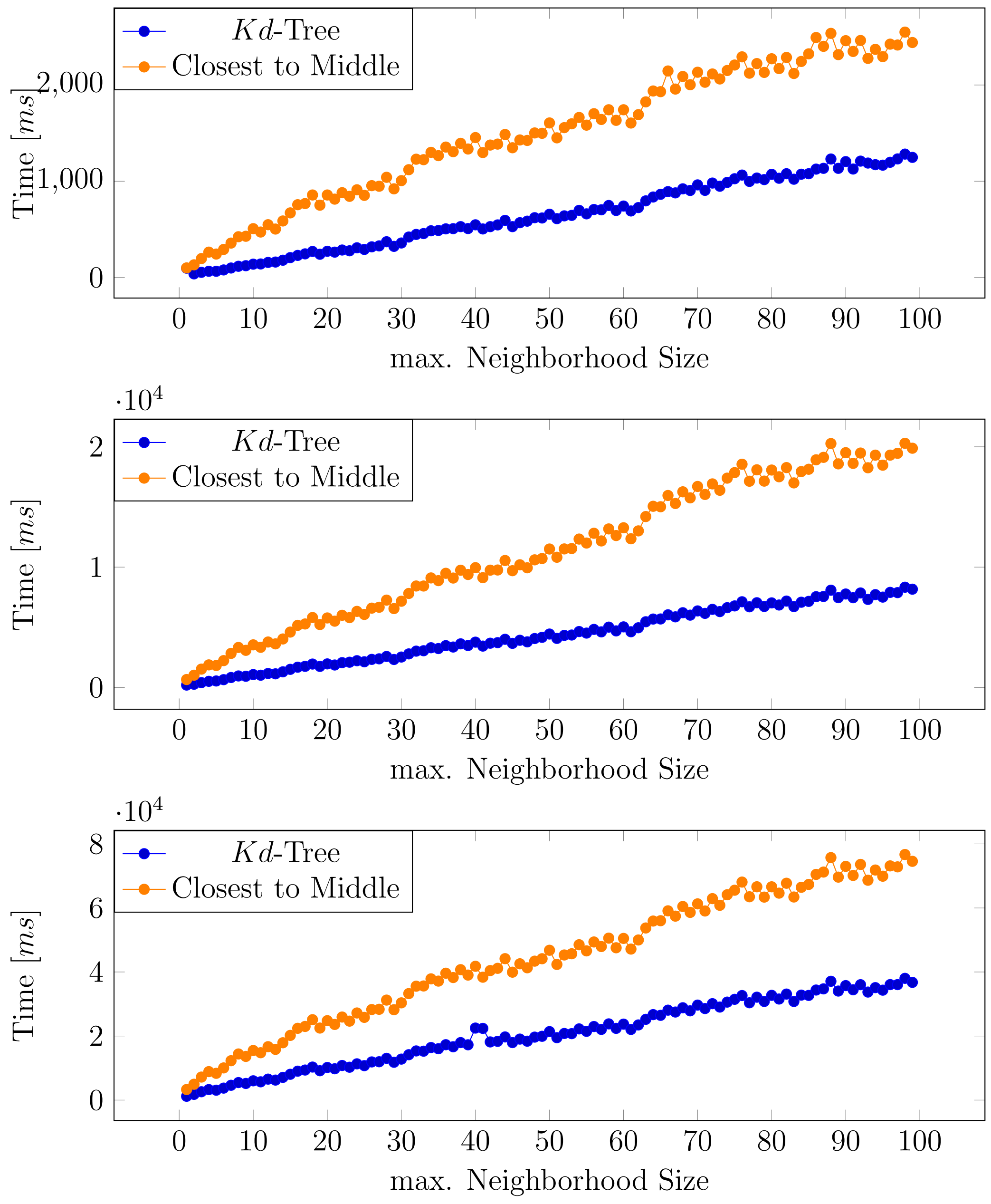


Figure 8.11: Times for the computation of different sized neighborhoods using Kd-Tree "Sorting" and the "Closest to Middle" implementation. On the top the times on the original cylinder geometry, in the middle the times on the geometry after one subdivision, and on the bottom after two subdivisions.

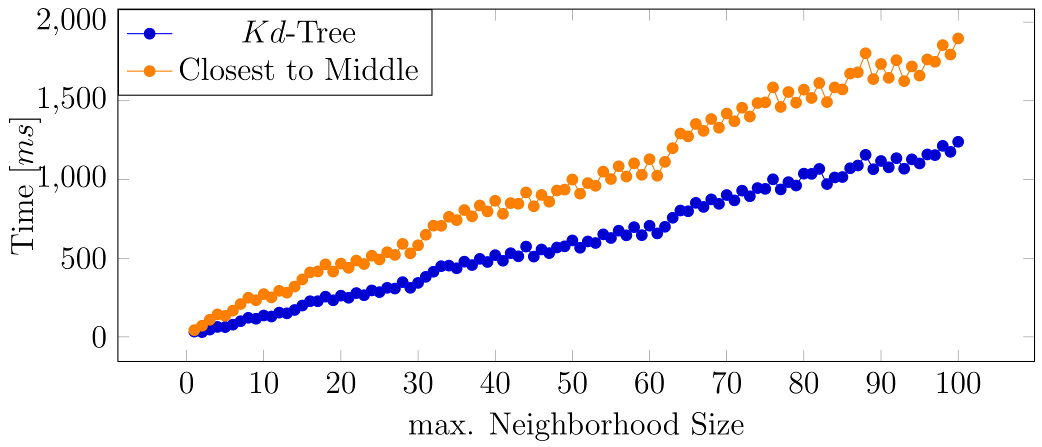
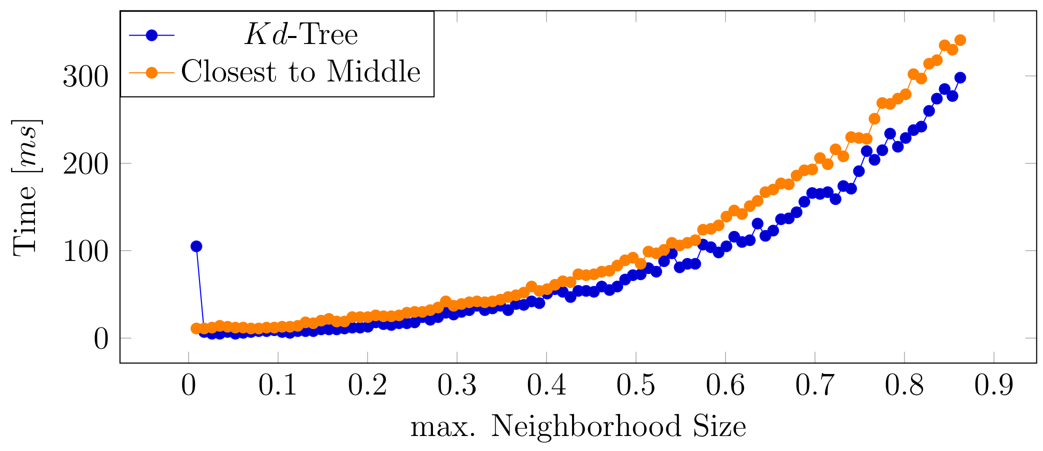


Figure 8.12: Times for the computation of different sized neighborhoods using Kd-Tree "Sorting" and the "Closest to Middle" implementation. Times are taken on the original Costa surface without subdivision and the experiment was repeat five times. The plot shows the median.

8.4 Conclusion from computational results

We saw in Section 8.2 that the usage of Kd-Trees can speed up the process of nearest neighbor computations drastically, compared to the method outlined in Section 5.1 and therefore also compared to any naive method. Hence, Kd-Trees should be used for neighborhood computations in point cloud settings. Concerning the different implementations of Kd-Trees we saw that building a balanced Kd-Tree by using the median as pivot element and determining the median using a randomized algorithm is faster than any other shown implementation. This coincides with the general observation that, although theoretically faster, deterministic algorithms are in practice often slower than randomized algorithms.
Finally, we can not reproduce the results from [Moo91]. In all three tested applications, the "Closest to Middle" implementation of Kd-Trees performed worse or on par with the Kd-Trees producing a balanced Kd-Tree.



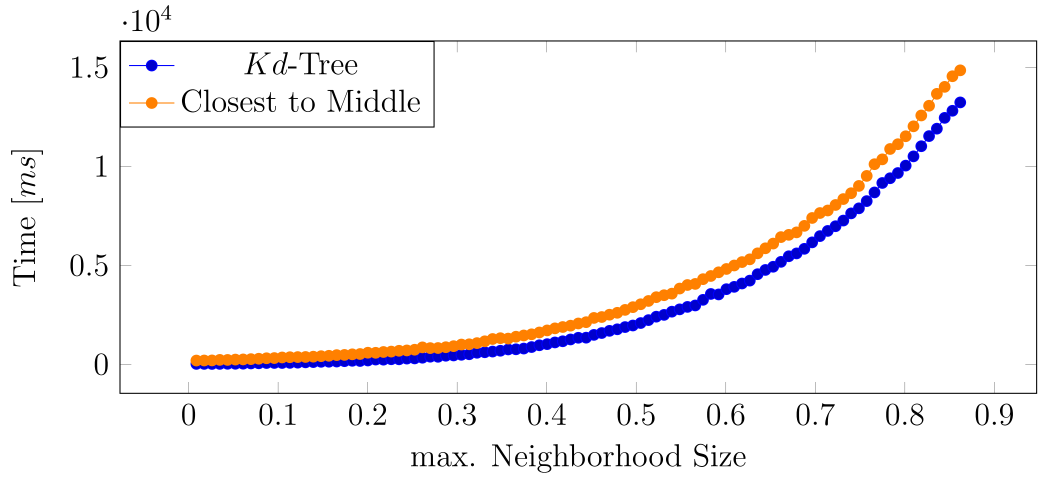


Figure 8.13: Neighborhood computation times for different influence radii and no restriction on the neighborhood size using the "Sorting" and "Closest to Middle" implementations.

Chapter 9

Conclusion and Further Research

In this thesis we presented an application for neighborhood computation in a point cloud setting, namely the smoothing of point clouds. We gave different approaches on how to implement the presented nearest neighbor methods and experimentally showed that the fastest choice from the presented implementations is a randomized median algorithm in a balanced Kd-Tree for both fast building and neighborhood computation.

Nonetheless certain aspects are left for further research. For example note that the smoothing algorithm as presented in [LP05] does not include automatic feature detection. Further research for compatible feature detection mechanisms should be conducted and a suitable procedure should be included in the presented program.

It was already mentioned in [LP05] that smoothing behaves amazingly well if the neighborhood is given from the final smooth object already. This poses an interesting question on how well neighborhoods can be obtained from point clouds at all and whether the presented methods can be improved.

Already in the Introduction we dismissed, for the course of this thesis, smoothing approaches that use meshes. The main disadvantage of meshes is the time used to actually compute the mesh. However, meshes do have certain advantages. Assume that the point cloud is a scan of a very thin surface. Using the presented nearest neighbor search techniques, points from one side of the surface might, depending on the thickness of the surface, the influence radius, and the maximum valence, be considered as neighbors of the other side of the surface. Meshes do not pose these problems and they are to be taken seriously in further research.

In Chapter 4 we dismissed Quadtrees and R-Trees in favor of Kd-Trees. Although this decision stands, an interesting question is the following: Can

a probability distribution be found that mimics point clouds obtained from surface scans? And can such a probability distribution be used to make any statements on the efficiency of different data structures, beside computational results?
In [Moo91], the pivot strategy "Closest to middle of most spread dimension" was introduced as beneficial for the nearest neighbor search. We have not been able to reproduce this results. Further tests could reveal the reasons for this phenomenon.
Finally, in a Quadtree when looking for nearest neighbors it is possible to search in four directions. Our Kd-Tree implementation only offers to traverse the tree. By introducing links on the leaf-level which connect the leafs of the tree horizontally, one would be able to peek into neighboring subtrees. Also the process could be parallelized into traversing the tree in three directions. However, not too many links apart from the suggested ones can be introduced, before the Kd-Tree actually collapses into a Quadtree.

Appendices

Appendix A

PCA Nearest Neighbor JavaView Implementation

```
protected void computeNeighbourByCovariance(PgPointSet geom
    ,
      double influence, int nov) {
    int dim            = geom.getDimOfVertices();
    PdVector [] vertex     = geom.getVertices();
    PiVector  neigh    = new PiVector(nov);
    double influence2    = influence*influence;

    PdVector [] bndbox   = geom.getBounds();
    if (bndbox == null) {
      // if (PsDebug.WARNING) PsDebug.warning("missing
           bndbox of geometry = "+geom.getName());
      return;
    }
    PdMatrix covMat    = computeCovariance(null, vertex);
    if (covMat == null) {
      if (PsDebug.WARNING) PsDebug.warning("missing
           vertices to compute covariance matrix, geom = "+
           geom.getName());
      return;
    }
    PdVector [] eVector = PdVector.realloc(null, dim, dim);
    PdVector eValue    = new PdVector(dim);
    int [] spec         = computeCovarianceSpectrum(covMat,
        eVector, eValue);

    // Sort eigenvalues by magnitude, princ[0] is eVector
        if the largest eValue.
    PdVector [] princ    = new PdVector[dim];
    for (int j=0; j<dim; j++)
      princ[j]   = eVector[spec[dim-1-j]];

    PdVector diag      = princ[0];
    PdVector diag2       = princ[1];
    PdVector diag3       = princ[2];

    // Use index vector to identify position along
        direction
    int [] index        = new int[nov];
    double [] height    = new double[nov];
    for (int i=0; i<nov; i++)
      height[i] = PdVector.dot(diag, vertex[i]);
    PuMath.heapsort(nov, height, index);
    // For each vertex store its position in the height
        vector
    int [] indexInv       = new int[nov];
    for (int i=0; i<nov; i++)
      indexInv[index[i]] = i;

    int [] index2       = new int[nov];
    double [] height2   = new double[nov];
    for (int i=0; i<nov; i++)
      height2[i] = PdVector.dot(diag2, vertex[i]);
    PuMath.heapsort(nov, height2, index2);
    // For each vertex store its position in the height
        vector
    int [] indexInv2       = new int[nov];
    for (int i=0; i<nov; i++)
      indexInv2[index2[i]] = i;

    int [] index3       = new int[nov];
    double [] height3   = new double[nov];
    for (int i=0; i<nov; i++)
      height3[i] = PdVector.dot(diag3, vertex[i]);
    PuMath.heapsort(nov, height3, index3);
    // For each vertex store its position in the height
        vector
    int [] indexInv3       = new int[nov];
    for (int i=0; i<nov; i++)
      indexInv3[index3[i]] = i;

    int maxValence        = m_maxValence.getValue();
    int [] indexNeigh     = new int[nov];
    double [] distNeigh = new double[nov];

    // Largest index smaller than i such that dist(vertex[
        indMin], vertex[i])<influence.
    int currMin    = 0;
    m_minNeighCnt = Integer.MAX_VALUE;
    for (int i=0; i<nov; i++) {
      while (height[index[i]]-height[index[currMin]] >
          influence)
        currMin++;

      // In direction 2 and 3 we compute integer bounds
          currMinI and currMaxI
      // which enclose all vertices of the influence
          interval of the current
      // vertex index[i] (currMinI and currMaxI both belong
          to the influence
      // interval too). The values currMinI and currMaxI
          are indices of
      // the index2 and index3 array.

      // Find in 2 direction currMin2 and currMax2
      int currMin2 = indexInv2[index[i]]-1;
      while (currMin2>=0 &&
            height2[index[i]]-height2[index2[currMin2]]<
                influence) {
        currMin2--;
      }
      currMin2++;

      int currMax2 = indexInv2[index[i]];
      while (currMax2<nov &&
            height2[index2[currMax2]]-height2[index[i]] <
                influence) {
        currMax2++;
      }
      currMax2--;

      // Find in 3 direction currMin3 and currMax3
      int currMin3 = indexInv3[index[i]];
      while (currMin3>=0 &&
            height3[index[i]]-height3[index3[currMin3]] <
                influence) {
        currMin3--;
      }
      currMin3++;

      int currMax3 = indexInv3[index[i]];
      while (currMax3<nov &&
            height3[index3[currMax3]]-height3[index[i]] <
                influence) {
        currMax3++;
      }
      currMax3--;

      int numNeigh = 0;
      for (int j=currMin; j<nov; j++) {
        // Must first check if we are still in the 1-
            interval
        // since this is the only place when we break.
        if (i<j && height[index[j]]-height[index[i]]>
            influence)
          break;
        // Check of index[j]-vertex lies in 2-influence
            interval of index[i]
        if (indexInv2[index[i]]<currMin2 || currMax2<
            indexInv2[index[i]])
          continue;
        if (indexInv3[index[i]]<currMin3 || currMax3<
            indexInv3[index[i]])
          continue;
        if (j == i)
          continue;

        double dist = PdVector.sqrDist(vertex[index[i]],
            vertex[index[j]]);
        if (dist > influence2)
          continue;

        neigh.m_data[numNeigh]  = index[j];
        distNeigh[numNeigh]   = dist;
```

```
          numNeigh++;
        }
        if (numNeigh < m_minNeighCnt)
          m_minNeighCnt = numNeigh;
        if (!m_bEnableMaxValence) {
          m_neigh[index[i]].setSize(numNeigh);
          m_neigh[index[i]].copy(0, neigh, 0, numNeigh);
        } else {
          // Get the nearest neighbours
          PuMath.heapsort(numNeigh, distNeigh, indexNeigh);

          numNeigh = Math.min(numNeigh, maxValence);
          m_neigh[index[i]].setSize(numNeigh);
          for (int j=0; j<numNeigh; j++)
            m_neigh[index[i]].m_data[j] = neigh.m_data[
                indexNeigh[j]];
        }
      }
      for (int i=0; i<nov; i++) {
        vertex[i].setName(String.valueOf(m_neigh[i].getSize()
          ));
      }
    }
```

Listing A.1: Implementation of the technique outlined in Section 5.1.

Appendix B

PCA Nearest Neighbor JavaView Implementation

```java
import java.security.InvalidParameterException;
import java.util.Comparator;

import jv.vecmath.PdVector;

/**
 * This class realizes a Comparator for PdVectors of
      arbitrary dimension. Within its compare method, two
 * vectors are compared lexicographically with respect to
      the startDimension set in the comparator.
 * For example, if the dimension is 3 and startDimension is
      1, the vectors are compared in their
 * components y, z, x.
 * @author Martin Skrodzki
 * @see Comparator
 * @see PdVector
 * @version   30.09.14, 1.0.0 created (ms)
 */
public class LexicographicalComparator implements
    Comparator<PdVector> {

  /**
   * The dimension to start the lexicographical comparison
       at.
   */
  protected int startDimension;

  /**
   * The dimension of the vectors compared using the
       comparator
   */
  protected int dimension;

  /**
   * Creates a new Lexicographical Comparator to compare
       two PdVectors lexicographically.
   * @param startDimension The dimension where the
       lexicographical comparison is started at,
   *  some value >=0 and <= dimension-1.
   * @param dimension The dimension of the Vectors that are compared, some value >0.
   * @throws InvalidParameterException If either startDimension<0, dimension<=0 or startDimension >= dimension.
   */
  public LexicographicalComparator(int startDimension, int dimension) throws InvalidParameterException {
    super();
    if (startDimension < 0 || dimension <= 0) {
      throw new InvalidParameterException("Can only initialize a Lexicographical Comparator on startDimension"
          + "greater equal to 0 and dimension greater than 0, but was given startDimension "+startDimension
          +" and dimension "+dimension+".");
    }
    if (startDimension >= dimension)   {
      throw new InvalidParameterException("Can only initialize a Lexicographical Comparator on a startDimension "
          + "less than dimension, but was given dimension "+dimension+" and startDimension "
          +startDimension+".");
    }
    this.startDimension = startDimension;
    this.dimension = dimension;
  }

  /* (non-Javadoc)
   * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
   */
  @Override
  public int compare(PdVector arg0, PdVector arg1) {
    if (arg0.getSize() != arg1.getSize()) {
      throw new ClassCastException("LexicographicalComparator can only compare two vectors of size same size, "
          + "but vectors have sizes "+arg0.getSize()+" and "+arg1.getSize()+".");
    }
    if (arg0.getSize() != this.dimension) {
      throw new InvalidParameterException("LexicographicalComparator of dimension "+this.dimension+
          " can only compare vectors of according dimension, but was given vectors of dimension "+
          arg0.getSize()+".");
    }
    int d = startDimension;
    //Compare the entries of the Vectors according to the given startDimension of this comparator
    if (arg0.getEntry(d) < arg1.getEntry(d)) {
      return -1;
    } else
    if (arg0.getEntry(d) > arg1.getEntry(d)) {
      return 1;
    } else {
      d = (d+1) % dimension;
      //Iterate cyclically through all other dimensions
          until the startDimension is reached again
      while (d != startDimension) {
        if (arg0.getEntry(d) < arg1.getEntry(d)) {
          return -1;
        } else
        if (arg0.getEntry(d) > arg1.getEntry(d)) {
          return 1;
        }
        d = (d+1) % dimension;
      }
    //The vectors have the same entries, tell them apart by
          their hash codes.
    if (arg0.hashCode() > arg1.hashCode()) {
      return 1;
    } else if (arg0.hashCode() < arg1.hashCode()){
      return -1;
    }
    return 0;
    }
  }

  /**
   * @return the startDimension, i.e. the dimension where
       the lexicographical comparison is started at,
   *  some value >=0 and <dimension.
   */
  public int getStartDimension() {
    return startDimension;
  }

  /**
   * @param startDimension set the dimension to start the
       lexicographical comparison at. Possible values lie
       in the
   *  range [0,dimension-1].
   * @throws InvalidParameterException If the given
       startDimension parameter is < 0 or greater equal to
       the set dimension.
   */
  public void setStartDimension(int startDimension) throws
      InvalidParameterException {
    if (startDimension < 0) {
      throw new InvalidParameterException("Can only set
          startDimension to a value greater equal than 0, "
          + "but was given value "+startDimension+".");
    }
    if (startDimension >= dimension) {
      throw new InvalidParameterException("Can only set a
          startDimension strictly less to the dimension, "
          + "but was set dimension "+dimension+" and given
              startDimension "+startDimension+".");
    }
    this.startDimension = startDimension;
  }

  /**
   * @return the dimension, i.e. the dimension of the
       vectors that can be compared using this comparator.
   */
  public int getDimension() {
    return dimension;
  }

  /**
   * @param dimension Set the dimension of the vectors that
        are to be compared by this lexicographical
       comparator,
   *  possible values must be strictly larger than 0.
   * @throws InvalidParameterException If the given
       dimension parameter is <= 0 or strictly less than
       the set startDimension.
   */
  public void setDimension(int dimension) throws
      InvalidParameterException {
    if (dimension <= 0) {
      throw new InvalidParameterException("Can only set
          dimension to a value strictly greater than 0,"
          + " but was given value "+dimension+".");
    }
    if (startDimension >= dimension) {
      throw new InvalidParameterException("Can only set a
          dimension strictly greater to the startDimension,
          "
          + " but was given dimension "+dimension+" and set
              startDimension "+startDimension+".");
    }
    this.dimension = dimension;
  }

}
```

Listing B.1: Implementation of the Lexicographical Order as described in Section 3.3.1.

Appendix C

Kd-Tree Sorting Class

```java
package devMS.kdTree;

import java.security.InvalidParameterException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

import devMS.comparators.LexicographicalComparator;

import jv.geom.PgPointSet;
import jv.vecmath.PdVector;

/**
 * This class implements the abstract class {@link KdTree}.
     On top of the functionality of the abstract class,
 * this class can actually build a KdTree. This is done via
     an initial sort of the points in three lists:
 * One sorting according to each dimension of the points.
     The sortings are maintained during the building
     process
 * and thereby it is trivial to find the median.
 * @author Martin Skrodzki
 */
public class KdTree_Sorting extends KdTree {

  /**
   * Creates a KdTree from the given set of points by
       calling the method {@link #buildTree(PgPointSet)}.
   * @param points The set of points that are to be
       represented by this KdTree.
   * @throws InvalidParameterException if the given point
       set is NULL.
   */
  public KdTree_Sorting(PgPointSet points) throws
      InvalidParameterException{
    super(points);
    buildTree(points);
  }

  /* (non-Javadoc)
```

```java
   * @see devMS.kdTree.KdTree#createTree(jv.geom.PgPointSet)
   */
  protected void buildTree(PgPointSet points) {
    int length = points.getNumVertices();
    //Create ArrayLists that can be sorted later on using the Collections.sort() method
    ArrayList<ArrayList<PdVector>> sortings = new ArrayList<ArrayList<PdVector>>(dimension);
    for (int i=0; i<dimension; i++) {
      sortings.add(new ArrayList<PdVector>(length));
    }
    //Fill the ArrayLists with the points from the base geometry
    for (int j=0; j<length; j++) {
      for (int i=0; i<dimension; i++) {
        sortings.get(i).add(points.getVertex(j));
      }
    }

    //Sort the lists
    for (int i=0; i<dimension; i++) {
      Collections.sort(sortings.get(i), new LexicographicalComparator(i,dimension));
    }

    //TODO Is sorted insertion faster than insertion and sorting?

    this.root = recursiveBuild(sortings,0,0,length-1);
  }

  /**
   * Recursively builds a KdTree from a set of points. The points are given to the method in three sorted lists.
   * The method finds the median in the current split dimension, splits all three lists according to the median
   * and passes the lists on recursively, while stating on what part of the lists the recursion should act.
   * @param sortings A list of sorted lists, where each lists realizes a sorting along a certain dimension.
   * @param splitDim The dimension in which to search for the median.
   * @param start The starting index from where to work on the lists.
   * @param end The ending index where to end working on the lists.
   * @return A node representing the root of a KdTree storing all points between start and end in the given lists.
   */
  private Node recursiveBuild(ArrayList<ArrayList<PdVector>> sortings, int splitDim, int start, int end) {

    //TODO Rewrite this not to resort in the same place, but use an array twice as long and sort in the first and second half alternating.

    //No element has been passed to the method, return null
    if (end < start) {
      return null;
    }
    //Only one element has been passed to the method,
        create a leaf containing this element
    if (start == end) {
      Node result = new Node(null, null, Node.
          noHyperplaneValue, splitDim, sortings.get(0).get(
          start), true,
           indexTable.get(sortings.get(0).get(start)));
      return result;
    } else {
    //There is more than one element, apply recursion
      PdVector median = null;
      double splitValue = 0;
      int medianIndex = (start+end)/2;
      //Find the median according to the given
          splitDimension
      median = sortings.get(splitDim).get(medianIndex);
      splitValue = median.getEntry(splitDim);

      //Partition the other two lists accordingly
      for (int i=0; i<dimension; i++) {
        if (i != splitDim) {
          partition(sortings.get(i), median, splitDim,
              start, end, medianIndex);
        }
      }

      //Create a new Node with two children being the
          recursive processing of the rest of the elements
      Node left = recursiveBuild(sortings,(splitDim+1) %
          dimension,start,medianIndex-1);
      Node right = recursiveBuild(sortings,(splitDim+1) %
          dimension,medianIndex+1,end);
      Node result = new Node(left, right, splitValue,
          splitDim, median,false,indexTable.get(median));

      return result;
    }
  }

  /**
   * Partitions a part of a given list, defined by the
       start and end index around a pivot element. The
       sorting
   * of the list is kept intact while partitioning around
       the pivot element.
   * @param sortedList The sorted list which part is to be
       partitioned.
   * @param pivot The pivot element around which to
       partition the indicated part of the given list.
   * @param splitDim The dimension according to which the
       comparator acts to keep the sorting intact
   * @param start The starting index indicating where the
       part of the list starts that is to be partitioned.
   * @param end The ending index indicating where the part
       of the list ends that is to be partitioned.
   * @param medianIndex The index of the median within the
       relevant part of the list.
   */
  protected void partition( ArrayList<PdVector> sortedList,
              PdVector pivot,
              int splitDim,
              int start,
              int end,
              int medianIndex) {

    //Create a new Comparator to compare the elements in
        the sorted list
    Comparator<PdVector> comparator = new
        LexicographicalComparator(splitDim,dimension);

    //Create a Queue to store the elements in
    LinkedList<PdVector> queue = new LinkedList<PdVector>()
        ;
    //store the first position that is considered or known
        to be empty
    int smallestEmpty = start;
    //Iterate through the sorted List up to the medianIndex
    for (int i=start; i<=medianIndex; i++) {
      PdVector current = sortedList.get(i);
      int comparison = comparator.compare(current, pivot);
      if (comparison < 0) {
        //The element is smaller than the pivot element and
            has to be placed on the left
        sortedList.set(smallestEmpty, current);
        smallestEmpty++;
      }
      if (comparison > 0) {
        //Add the element to the queue
        queue.add(current);
      }
    }
    //Reached the position of the median, store it here
    sortedList.set(medianIndex, pivot);
    //Process the right side of the median, add all
        previously queued items here
    for (int i=medianIndex+1; i<=end; i++) {
      PdVector current = sortedList.get(i);
      int comparison = comparator.compare(current, pivot);
      if (comparison < 0) {
        //The element is smaller than the pivot element and
            has to be placed on the left
        sortedList.set(smallestEmpty, current);
        smallestEmpty++;
      } else {
        if (comparison > 0) {
          //Add the element in the queue
          queue.add(current);
        }
      }
      // In any case, place an element from the queue at
          the current position
      sortedList.set(i, queue.pop());
    }
  }
```

```
}
```

Listing C.1: Implementation of a Kd-Tree as outlined in Section 3.3.3.

Appendix D

Kd-Tree Median Class

```java
package devMS.kdTree;

import java.security.InvalidParameterException;

import devMS.comparators.LexicographicalComparator;
import devMS.median.IMedianAlgorithm;

import jv.geom.PgPointSet;
import jv.vecmath.PdVector;

/**
 * This class implements the abstract class {@link KdTree}.
     On top of the functionality of the abstract class,
 * this class can actually build a KdTree. This is done via
     a recursive method that finds the median of a point
 * set and partitions the point set around the median.
 * @author Martin Skrodzki
 */
public class KdTree_Median extends KdTree {

  /**
   * An algorithm to determine the median of a set.
   */
  protected IMedianAlgorithm medianAlgorithm;

  /**
   * Several Lexicographical Comparators that are used in
       the recursive run of
   * {@link #recursiveBuild(PdVector[], int, int, int)}
       which are stored as a field of the class to not
   * have to initialize them in every run of the recursion.
   */
  protected LexicographicalComparator[] dimComparator;
  protected LexicographicalComparator comparator;

  /**
   * Creates a KdTree from the given set of points by
       calling the method {@link #buildTree(PgPointSet)}.
       During
   * the building process it is necessary to determine the
       Median of a set. This is done by the given
   * median Algorithm.
   * @param points The set of points which are to be represented by this KdTree.
   * @param medianAlgorithm An algorithm to determine the median of a set.
   * @throws InvalidParameterException If the given dimension is strictly less than 1.
   */
  public KdTree_Median(PgPointSet points, IMedianAlgorithm medianAlgorithm) throws InvalidParameterException{
    super(points);
    if (dimension <= 0) {
      throw new InvalidParameterException("Can not build a KdTree on dimension < 1, "
        + "but was given dimension "+dimension+".");
    }
    this.medianAlgorithm = medianAlgorithm;

    //Set up a Lexicographical Comparator for each dimension
    for (int i=0; i<dimension; i++) {
      this.dimComparator[i] = new LexicographicalComparator(i,dimension);
    }

    //Build the actual tree
    buildTree(points);
  }

  /* (non-Javadoc)
   * @see devMS.kdTree.KdTree#buildTree(jv.geom.PgPointSet)
   */
  protected void buildTree(PgPointSet points) {
    this.root = recursiveBuild(points.getVertices(),0,0, points.getNumVertices()-1);
  }

  /**
   * Recursively builds a KdTree from a set of points. The method finds the median in the current split
   * dimension and partitions the set accordingly. It is specified by the indices left and right on what part
   * of the set the method acts.
   * @param points The set of points to be represented by the KdTree this method builds.
   * @param splitDim The dimension in which to split the set, i.e. to find the median; a value >=0 and < dimension.
   * @param left The leftmost index of the part of the point set on which the method currently works.
   * @param right The rightmost index of the part of the point set on which the method currently works.
   * @return The root of a KdTree representing the given points.
   */
  private Node recursiveBuild(PdVector[] points, int splitDim, int left, int right) {
    //No point to be represented, i.e. root stays NULL.
    if (left > right) {
      return null;
    } else
    //Only one point to be represented, i.e. create a leaf containing the single point.
    if (left == right) {
      return new Node(null, null, Node.noHyperplaneValue, splitDim, points[left], true, indexTable.get(points[left]));
    } else {
      //Switch to the correct comparator that is used to compare the points to the found median.
      comparator = dimComparator[splitDim];
      int i = left;
      int j = right;
      int m = (left+right)/2;
      PdVector median = medianAlgorithm.median(points, splitDim);
      //Proceed from left to right and from right to left simultaneously. If points on the left are larger
      //than the median or points on the right are smaller than the median, exchange them.
      while (i < j) {
        while (comparator.compare(points[i], median) < 0) { i++; }
        while (comparator.compare(points[j], median) > 0) { j++; }
        PdVector temp = points[i];
        points[i] = points[j];
        points[j] = temp;
      }
      //Recursively create a node storing the median with a left and a right subtree holding the points
      //which have been partitioned to the left or right of the median respectively.
      return new Node(recursiveBuild(points, (splitDim+1) % dimension, left, m-1),
              recursiveBuild(points, (splitDim+1) % dimension, m+1, right),
              median.getEntry(splitDim), splitDim, median, false, indexTable.get(median));
    }
  }
}
```

Listing D.1: Implementation of a *Kd*-Tree as outlined in Section 3.3.4.

Appendix E

Nearest Neighbor Computation in abstract Kd-Tree

```
/**
   * A recursive search for the input point within the
       KdTree is performed. It ends in a leaf
   * of the KdTree representing the position into which the
        input point would have been stored.
   * From here the tree is traversed backwards up to the
       root, where every not yet accessed subtree
   * is considered if and only if the subtree might still
       contain points closer to the input
   * point than the points found so far and the subtree is
       still within influence radius.
   * @param input A point around which to search for
       neighbors.
   * @param count Number of neighbors to be found.
   * @param influence Influence radius around the input
       point to be considered.
   * @param currentBest List of currently closest points
       found.
   * @param currentPosition Node in the KdTree that is
       currently examined.
   * @return A queue containing the found neighbors, where
       the head of the queue is the one with the largest
       distance.
   */
  protected PriorityQueue<PdVector>
      getInfluencedNumberOfNearestNeighbors(
       PdVector input, int count, double influence,
           PriorityQueue<PdVector> currentBest, Node
           currentPosition,
       Boolean includeInput){
     if (currentPosition == null) {
       //The currentPosition is null, nothing can be done
           here, return the currently known nearest
           neighbors
       return currentBest;
     } else {
       //The current Position contains a point, add it to
           the list of currently known neighbors,
      //if it fits the influence radius. If the list
         becomes to large, trim it
      if (includeInput || (currentPosition.getPoint().
         hashCode() != input.hashCode())) {
        if (currentPosition.getPoint().dist(input) <=
            influence) {
          currentBest.add(currentPosition.getPoint());
        }
        if (currentBest.size() > count) {
          currentBest.poll();
        }
      }
      //If the currentPosition is a Leaf, recursion comes
         to an end and the currently known
      //nearest neighbors are reported
      if (currentPosition.isLeaf()) {
        return currentBest;
      } else {

        int splitDim = currentPosition.getSplitDim();
        double inputSplitDimCoordinate = input.getEntry(
            splitDim);
        double currentPositionSplitDimCoordinate =
            currentPosition.getPoint().getEntry(splitDim);

        //If the currentPosition is not a Leaf, we can
            apply recursion to both sides of the hyperplane
        if (inputSplitDimCoordinate <=
            currentPositionSplitDimCoordinate) {

          //Examine the side where the input point lies
          currentBest =
              getInfluencedNumberOfNearestNeighbors(
               input,count,influence, currentBest,
                  currentPosition.getLeft(),includeInput);

          if ((currentBest.size() < count)
               && (influence >=  (Math.abs(
                  inputSplitDimCoordinate -
                  currentPositionSplitDimCoordinate)))) {
            //If there are still neighbors missing and they
                still lie within influence radius,
            //examine the other side, too
            currentBest =
                getInfluencedNumberOfNearestNeighbors(
                 input,count,influence,currentBest,
                    currentPosition.getRight(),includeInput
                    );
          } else {
            //In case the needed amount of neighbors has
                already been found, the other side is only
                examined
            //if points might be closer to the input than
                the points found so far
            PdVector currentWorstPoint = currentBest.peek()
                ;
            if ((currentWorstPoint.dist(input) > (Math.abs(
                inputSplitDimCoordinate -
                currentPositionSplitDimCoordinate)))
```

```
                                && (influence >=  (Math.abs(inputSplitDimCoordinate - currentPositionSplitDimCoordinate)))){
                            currentBest = getInfluencedNumberOfNearestNeighbors(
                                    input,count,influence,currentBest, currentPosition.getRight(), includeInput);
                        }
                    }
                } else {

                    //Examine the side where the input point lies
                    currentBest = getInfluencedNumberOfNearestNeighbors(
                            input,count,influence,currentBest, currentPosition.getRight(),includeInput);

                    if ((currentBest.size() < count)
                            && (influence >=  (Math.abs(inputSplitDimCoordinate - currentPositionSplitDimCoordinate)))){
                        //If there are still neighbors missing, examine the other side, too
                        currentBest = getInfluencedNumberOfNearestNeighbors(
                                input,count,influence,currentBest, currentPosition.getLeft(),includeInput);
                    } else {
                        //In case the needed amount of neighbors has already been found, the other side is only examined
                        //if points might be closer to the input than the points found so far
                        PdVector currentWorstPoint = currentBest.peek();
                        if ((currentWorstPoint.dist(input) > (Math.abs(inputSplitDimCoordinate - currentPositionSplitDimCoordinate)))
                                && (influence >=  (Math.abs(inputSplitDimCoordinate - currentPositionSplitDimCoordinate)))) {
                            currentBest = getInfluencedNumberOfNearestNeighbors(
                                    input,count,influence,currentBest, currentPosition.getLeft(), includeInput);
                        }
                    }
                }
                return currentBest;
            }
        }
    }
```

Listing E.1: Implementation of the nearest neighbor algorithm as given in Algorithm 9.

Appendix F

Implementation of alternative Pivot Rules

```
import java.security.InvalidParameterException;

import jv.geom.PgPointSet;
import jv.vecmath.PdVector;

import devMS.comparators.LexicographicalComparator;

/**
 * This class implements the abstract class {@link KdTree}.
      On top of the functionality of the abstract class,
 * this class can actually build a KdTree. This is done via
      a recursive building method. The Tree is build
 * using the point closest to the middle of the largest
     spread dimension as pivot element.
 * @author Martin Skrodzki
 */
public class KdTree_ClosestToMiddle extends KdTree {

  public KdTree_ClosestToMiddle(PgPointSet points)
      throws InvalidParameterException {
    super(points);
    if (dimension <= 0) {
      throw new InvalidParameterException("Can not build a
          KdTree on dimension < 1, "
          + "but was given dimension "+dimension+".");
    }
    //Build the actual tree
    buildTree(points);
  }

  /* (non-Javadoc)
   * @see devMS.kdTree.KdTree#buildTree(jv.geom.PgPointSet)
   */
  protected void buildTree(PgPointSet points) {
    PdVector minBound = new PdVector(points.
        getDimOfVertices());
    PdVector maxBound = new PdVector(points.getDimOfVertices());
    PdVector[] pointArray = new PdVector[points.getNumVertices()];
    for (int j=0; j<points.getDimOfVertices(); j++) {
      minBound.setEntry(j, points.getVertex(0).getEntry(j));
      maxBound.setEntry(j, points.getVertex(0).getEntry(j));
    }
    for (int i=0; i<points.getNumVertices(); i++) {
      pointArray[i] = points.getVertex(i);
      for (int j=0; j<points.getDimOfVertices(); j++) {
        if (pointArray[i].getEntry(j) < minBound.getEntry(j)) {
          minBound.setEntry(j, pointArray[i].getEntry(j));
        }
        if (pointArray[i].getEntry(j) > maxBound.getEntry(j)) {
          maxBound.setEntry(j, pointArray[i].getEntry(j));
        }
      }
    }
    this.root = recursiveBuild(pointArray,minBound,maxBound,0,points.getNumVertices()-1);
  }

  /**
   * Recursively builds a KdTree from a set of points. The points are given to the method in an array.
   * The method finds the point closest to the middle of the largest spread dimension, splits the array
   * according to this point and passes the lists on recursively, while stating on what part of the array
   * the recursion should act.
   * @param points Points to be stored in the tree
   * @param minBound One of the defining points of the bounding box of the point set.
   * @param maxBound The second defining point of the bounding box of the point set.
   * @param left Leftmost index of the points array to be included.
   * @param right Rightmost index of the points array to be included.
   * @return Root of a Kd-Tree storing all points from the given array.
   */
  private Node recursiveBuild(PdVector[] points, PdVector minBound, PdVector maxBound, int left, int right) {
    //No point to be represented, i.e. root stays NULL.
    if (left > right) {
      return null;
    } else
    //Only one point to be represented, i.e. create a leaf containing the single point.
    if (left == right) {
      return new Node(null, null, Node.noHyperplaneValue, Node.noHyperplaneDim, points[left], true);
    } else {
```

```java
    //Find the largest spread dimension
    int largestSpreadDim = 0;
    for (int i=1; i<minBound.getSize(); i++) {
      if (Math.abs(maxBound.getEntry(i)-minBound.getEntry(i))
        > Math.abs(maxBound.getEntry(largestSpreadDim)-minBound.getEntry(largestSpreadDim))) {
        largestSpreadDim = i;
      }
    }

    //Determine the splitValue, i.e. middle of the largest spread dimension
    double splitValue = maxBound.getEntry(largestSpreadDim)+minBound.getEntry(largestSpreadDim)/2;

    //Find the point closest to the middle of the largest spread dimension
    PdVector pivot = points[left];
    int pivotIndex = left;
    for (int i=left+1; i<=right; i++) {
      double currentDist  = points[i].getEntry(largestSpreadDim)-splitValue;
      double pivotDist  = pivot.getEntry(largestSpreadDim)-splitValue;
      if(Math.abs(currentDist) < Math.abs(pivotDist)){
        pivot = points[i];
        pivotIndex = i;
      }
    }
  //Partition the set according to the pivot element
    int i = left+1;
    int j = right;
    LexicographicalComparator comparator = new LexicographicalComparator(largestSpreadDim, minBound.getSize());
    //Place the pivot element in the first place
    points[pivotIndex] = points[left];
    points[left] = pivot;
    //Proceed from left to right and from right to left simultaneously. If points on the left are larger
    //than the median or points on the right are smaller than the median, exchange them.
    while (i < j) {
      while ((i<j) && (comparator.compare(points[i], pivot) <= 0)) { i++; }
      while ((i<j) && (comparator.compare(points[j], pivot) > 0)) { j--; }
      PdVector temp = points[i];
      points[i] = points[j];
      points[j] = temp;
    }
    points[left] = points[j];
    points[j] = pivot;
    PdVector rightMinBound = PdVector.copyNew(minBound);
    PdVector rightMaxBound = PdVector.copyNew(maxBound);
    maxBound.setEntry(largestSpreadDim, splitValue);
```

```
        rightMinBound.setEntry(largestSpreadDim, splitValue);

        //Recursively create a node storing the median with a
             left and a right subtree holding the points
        //which have been partitioned to the left or right of
             the median respectively.
        return new Node(recursiveBuild(points, minBound,
            maxBound, left, j-1),
                recursiveBuild(points, rightMinBound,
                    rightMaxBound, j+1, right),
                pivot.getEntry(largestSpreadDim),
                    largestSpreadDim, pivot, false);
      }
   }
}
```

Listing F.1: Implementation of a Kd-Tree utilizing the alternative pivot rule from [Moo91].

List of Figures

List of Algorithms

Listings

Bibliography

[ABK98] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. "A new Voronoi-based surface reconstruction algorithm". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM. 1998, pp. 415–421.

[Ale+03] Marc Alexa et al. "Computing and rendering point set surfaces". In: *Visualization and Computer Graphics, IEEE Transactions on* 9.1 (2003), pp. 3–15.

[Bär10] Christian Bär. *Elementare Differentialgeometrie*. German. 2nd. De Gruyter, 2010. ISBN: 978-3-11-022458-0.

[Ber+08] Mark de Berg et al. *Computational Geometry. Algorithms and Applications*. English. 3rd. Springer, 2008. ISBN: 978-3-540-77974-2.

[Bey+99] Kevin Beyer et al. "When is ?nearest neighbor? meaningful?" In: *Database Theory?ICDT?99*. Springer, 1999, pp. 217–235.

[BF05] Chris Boehnen and Patrick Flynn. "Accuracy of 3D scanning technologies in a face scanning scenario". In: *3-D Digital Imaging and Modeling, 2005. 3DIM 2005. Fifth International Conference on*. IEEE. 2005, pp. 310–317.

[Blu+73] Manuel Blum et al. "Time bounds for selection". In: *Journal of computer and system sciences* 7.4 (1973), pp. 448–461.

[Bri05] Thomas Brinkhoff. *Geodatenbanksysteme in Theorie und Praxis. Einführung in objektrelationale Geodatenbanken unter besonderer Berücksichtigung von Oracle Spatial*. German. 1st. Herbert Wichmann, 2005. ISBN: 3-87907-433-X.

[Buc+07] Ursula Buck et al. "Application of 3D documentation and geometric reconstruction methods in traffic accident analysis: with high resolution surface scanning, radiological MSCT/MRI scanning and real data based animation". In: *Forensic science international* 170.1 (2007), pp. 20–28.

[CDR00] Ulrich Clarenz, Udo Diewald, and Martin Rumpf. “Anisotropic geometric diffusion in surface processing”. In: *Proceedings of the conference on Visualization’00*. IEEE Computer Society Press. 2000, pp. 397–405.

[Com79] Douglas Comer. “Ubiquitous B-tree”. In: *ACM Computing Surveys (CSUR)* 11.2 (1979), pp. 121–137.

[Cor+13] Thomas H. Cormen et al. *Algorithmen - eine Einführung*. German. 4th. Oldenbourg, 2013. ISBN: 9783486748611.

[CZ05] Volker Coors and Alexander Zipf. *3D-Geoinformationssysteme. Grundlagen und Anwendungen*. German. 1st. Herbert Wichmann, 2005. ISBN: 3-87907-411-9.

[Des+99] Mathieu Desbrun et al. “Implicit fairing of irregular meshes using diffusion and curvature flow”. In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1999, pp. 317–324.

[DZ99] Dorit Dor and Uri Zwick. “Selecting the median”. In: *SIAM Journal on Computing* 28.5 (1999), pp. 1722–1758.

[Els+12] Jan Elseberg et al. “Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration”. In: *Journal of Software Engineering for Robotics* 3.1 (2012), pp. 2–12.

[Epp] David Eppstein. *Deterministic selection*. URL: http://www.ics.uci.edu/~eppstein/161/960130.html.

[Eri05] Christer Ericson. *Real-Time Collision Detection*. English. 1st. Elsevier, 2005. ISBN: 1-55860-732-3.

[FBF77] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. “An algorithm for finding best matches in logarithmic expected time”. In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pp. 209–226.

[Fla98] David Flanagan. *Java. In a nutshell*. German. 2nd. O’Reilly, 1998. ISBN: 3-89721-100-9.

[FR01] Michael S Floater and Martin Reimers. “Meshless parameterization and surface reconstruction”. In: *Computer Aided Geometric Design* 18.2 (2001), pp. 77–92.

[FR75] Robert W. Floyd and Ronald L. Rivest. “Algorithm 489: The Algorithm SELECT for Finding the Ith Smallest of N Elements [M1]”. In: *Commun. ACM* 18.3 (Mar. 1975), pp. 173–. ISSN: 0001-0782. DOI: 10.1145/360680.360694. URL: http://doi.acm.org/10.1145/360680.360694.

[Gam+94] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[GP11] Markus Gross and Hanspeter Pfister. *Point-based graphics*. Morgan Kaufmann, 2011.

[Gut84] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*. Vol. 14. 2. ACM, 1984.

[GUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems. The complete book*. English. 1st. Prentice Hall, 2002. ISBN: 0-13-031995-3.

[Han10] Andreas Handl. *Multivariate Analysemethoden. Theorie und Praxis multivariater Verfahren unter besonderer Berücksichtigung von S-PLUS*. German. 2nd. Springer, 2010. ISBN: 978-3-3642-14987-0.

[Her12] Philipp Herholz. “General discrete Laplace operators on polygonal meshes”. Diploma Thesis. Humboldt-University Berlin, 2012.

[JDD03] Thouis R Jones, Frédo Durand, and Mathieu Desbrun. “Non-iterative, feature-preserving mesh smoothing”. In: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, pp. 943–949.

[Jun11] Christopher Jung. “Vergleich von Quadtree, kd-tree und r-tree für statische und dynamische Geodaten”. MA thesis. Cologne University of Applied Sciences, 2011.

[Lam+02] Christopher Xu Fu Lam et al. “Scaffold development using 3D printing with a starch-based polymer”. In: *Materials Science and Engineering: C* 20.1 (2002), pp. 49–56.

[Leu+05] Barbara Leukers et al. “Hydroxyapatite scaffolds for bone tissue engineering made by 3D printing”. In: *Journal of Materials Science: Materials in Medicine* 16.12 (2005), pp. 1121–1124.

[Lev+00] Marc Levoy et al. “The digital Michelangelo project: 3D scanning of large statues”. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques.* ACM Press/Addison-Wesley Publishing Co. 2000, pp. 131–144.

[Lev04] David Levin. “Mesh-independent surface interpolation”. In: *Geometric modeling for scientific visualization.* Springer, 2004, pp. 37–49.

[LP05] Carsten Lange and Konrad Polthier. “Anisotropic smoothing of point sets”. In: *Computer Aided Geometric Design* 22.7 (2005), pp. 680–692.

[Moo91] Andrew Moore. *An introductory tutorial on kd-trees.* Tech. rep. Technical Report No. 209, Computer Laboratory, University of Cambridge. Pittsburgh, PA: Robotics Institute, Carnegie Mellon University, 1991.

[Omo87] S.M. Omohundro. “Efficient Algorithms with Neural Network Behavior”. In: *Journal of Complex Systems* 1.2 (1987), pp. 273–347.

[Ora] Oracle. *Java Platform, Standard Edition 7 API Specification.* URL: http://docs.oracle.com/javase/7/docs/api/overview-summary.html.

[OW02] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen.* Spektrum, Akad. Verlag, 2002.

[Pau+02] Mark Pauly et al. *Multiresolution modeling of point-sampled geometry.* Swiss Federal Institute of Technology, Computer Science Department,[Institute of Visual Computing], Computer Graphics Lab CGL, 2002.

[Pol+] Konrad Polthier et al. *A 3D Geometry Viewer and a Geometric Software Library written in Java.* URL: http://www.javaview.de.

[PR99] Tobias Preußer and Martin Rumpf. “Anisotropic nonlinear diffusion in flow visualization”. In: *Visualization’99. Proceedings.* IEEE. 1999, pp. 325–539.

[Pro] Mathematical Geometry Processing. *3D Scanner.* URL: http://www.mi.fu-berlin.de/en/math/groups/ag-geom/3D_print/index.html.

[Pro+03] Octavian Procopiuc et al. “Bkd-tree: A dynamic scalable kd-tree”. In: *Advances in Spatial and Temporal Databases.* Springer, 2003, pp. 46–65.

[RB90] Peter J Rousseeuw and Gilbert W Bassett Jr. “The remedian: A robust averaging method for large data sets”. In: *Journal of the American Statistical Association* 85.409 (1990), pp. 97–104.

[Ren+10] Fabian Rengier et al. “3D printing based on imaging data: review of medical applications”. In: *International journal of computer assisted radiology and surgery* 5.4 (2010), pp. 335–341.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. “Nearest neighbor queries”. In: *ACM sigmod record.* Vol. 24. 2. ACM. 1995, pp. 71–79.

[RNI10] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. “Hubs in space: Popular nearest neighbors in high-dimensional data”. In: *The Journal of Machine Learning Research* 11 (2010), pp. 2487–2531.

[SC03] Shashi Shekhar and Sanjay Chawla. *Spatial Databases. A Tour.* English. 1st. Pearson Education, 2003. ISBN: 0-13-017480-7.

[Sed92] Robert Sedgewick. *Algorithmen.* German. Addison-Wesley, 1992.

[SS] Thomas Stollin and Martin Skrodzki. “Documentation of the Project Orthogonal Range Searching”. Project Documentation for the course Scientific Visualization by Konrad Polthier at FU Berlin in 2013/14.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms.* English. 4th. Addison-Wesley, 2011. ISBN: 978-0-321-57351-3.

[Tau95] Gabriel Taubin. “Estimating the tensor of curvature of a surface from a polyhedral approximation”. In: *Computer Vision, 1995. Proceedings., Fifth International Conference on.* IEEE. 1995, pp. 902–907.

[Tho79] John A Thorpe. *Elementary topics in differential geometry.* Springer, 1979.

[Vai89] Pravin M Vaidya. “An O(n logn) algorithm for the all-nearest-neighbors problem”. In: *Discrete & Computational Geometry* 4.1 (1989), pp. 101–115.

[VMM99] Jörg Vollmer, Robert Mencl, and Heinrich Mueller. “Improved laplacian smoothing of noisy surface meshes”. In: *Computer Graphics Forum*. Vol. 18. 3. Wiley Online Library. 1999, pp. 131–138.