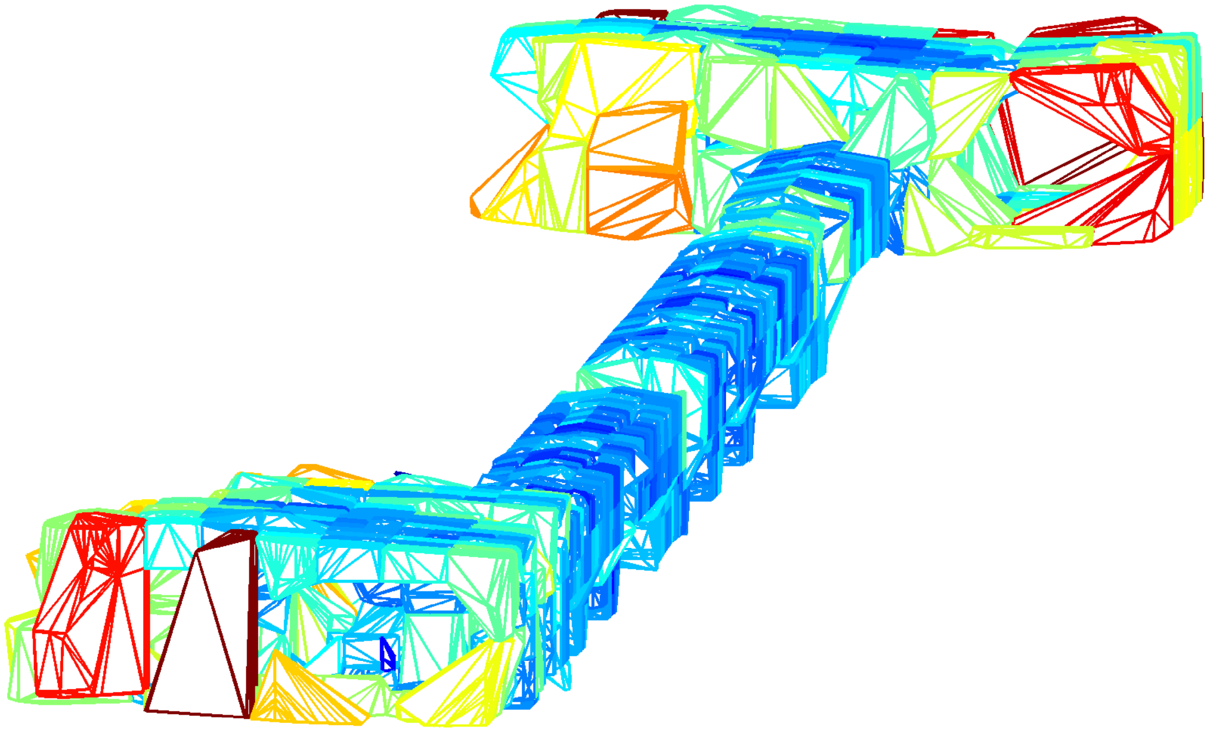

Point Cloud Simplification and Processing for Path-Planning

Master's Thesis in Engineering Mathematics and Computational Science

DAVID ERIKSSON

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2014

Abstract

Recently the area of motion planning research has been experiencing a significant resurgence of interest based on hybrid working environments that combine point and CAD models. Companies are able to work with point clouds and perform certain operations, such as path-planning, but they lack the support for fast shortest-distance computations for point clouds with more than tens of millions of points. Therefore, there is a need for handling and pre-processing massive point clouds for fast-queries.

In this thesis, algorithms have been developed that are capable of efficiently pre-processing massive point clouds for fast out-of-core queries allowing rapid computation of the exact shortest distance between a point cloud and a triangulated object. This is achieved by exploiting fast approximate distance computations between subsets of points and the triangulated object.

This approach was able to compute, on average, the shortest distance in 15 fps for a point cloud having 1 billion points, given only 8 GB of RAM. The findings and implementations will have a direct impact for the many companies that want to perform path-planning through massive point clouds since the algorithms are able to produce near real-time distance computations on a standard PC.

Acknowledgements

First of all I would like to thank Evan Shellshear for his excellent supervision and feedback throughout this thesis work. I would also like to express my gratitude to my supervisor and examiner from Chalmers, Thomas Ericsson, for his interest in this thesis. In addition, I would like to acknowledge Chalmers University of Technology and the Department of Mathematical Sciences, and everyone at FCC.

David Eriksson, Gothenburg 14/03/20

Contents

Abbreviations and Symbols

Abbreviation	Full text
AABB	Axis-aligned bounding box
$\mathcal{CH}$	Convex hull
HDD	Hard disk drive
PQP	Proximity query package
RAM	Random access memory
SSD	Solid state drive

Symbol	Name
α_i	The maximal displacement of S at iteration i
ϵ	Allowed error in the computed shortest distance
d_e	Length of the longest edge in a two-dimensional convex hull
d_j	Distance from S to the closest extreme point in convex hull number j
$d_{\min}$	The shortest distance between two geometric models
F	A face of a convex hull
P	Point cloud
$\bar{P}_\epsilon$	Simplification of the point cloud P
Q	Subset of the point cloud
$r_{\max}$	Upper bound of the distance from u to v
S	The object moved through the point cloud
T	Targeted number of points in each convex hull
u	The point in a convex hull that is closest to S
v	The extreme point in a convex hull that is closest to S
w	The extreme point in a convex hull that is closest to u
$\hat{x}$	Point that maximizes the minimum distance to the vertices in a polygon

1

Introduction

1.1 Background

High-resolution point clouds have become very important in the last 15 years after people started to exploit the advantages over triangle-based models in computer graphics applications [1], [2]. Improvements in scanning technologies make it possible to easily scan larger objects, thereby making point clouds more popular than CAD models in certain applications. The reason for this is that scanning an entire geometry is much easier than building a CAD model from scratch [1].

One area where point clouds are useful is in path-planning, where the interest is focused on collision detection and/or the computation of the shortest distance between a point cloud and an object moving through the point cloud. The processing and efficient structuring of massive point clouds is critical in order to speed-up the path-planning algorithms, [3]. The reason for this is that computing the necessary collisions and/or distances is a common bottleneck for the path-planning algorithm [4].

The point clouds used for path-planning can contain billions of points, [5], making it a challenge to process them efficiently. Volvo Cars have scanned their factories in Torslanda with the resulting point cloud containing 10 billion points, with a spacing of about 0.5 centimeters [6]. In order to illustrate how there can be billions of points in a point cloud, imagine a box with dimensions $300m \times 100m \times 10m$. If points are placed on each side as a lattice with spacing of 0.5 centimeters, the resulting point cloud will contain as many as 2.7 billion points.

Path-planning through environments consisting of triangle meshes has been studied intensively and lots of research exists on the subject [7], [8]. Although the area of path-planning with point clouds is newer, there are methods designed specifically for it, such as [9], [10]. Path-planning through hybrid environments can also be considered, where the geometry is a point cloud and the object is a triangle mesh; this combination has been studied in [1], [11], [12]. Hybrid path-planning is uncommon because of the preferred option to triangulate the point cloud and use path-planning on the resulting triangle mesh, [13]. However, this triangulation process is only of interest for point clouds smaller than those considered in this thesis, for which the algorithms used to triangulate the point cloud would be way too cumbersome to be practical [1].

This thesis will focus on path-planning with distance computation, which is still related to path-planning with collision computations for triangle meshes. A common assumption for collision computations is to assume an error bound such that false collisions may be detected but true collisions are never missed, [14]. Most modern approaches to collision detection build on this assumption since it makes it possible to simplify the point cloud or triangle mesh. The work in this thesis strives to develop a scheme that, if necessary, will be able to compute the exact shortest distance and does not rely on such a simplification scheme.

1.2 Purpose and goal

The goal of this master's thesis project is to construct a set of algorithms that are able to efficiently pre-process a massive point cloud for out-of-core proximity queries and mathematically prove that the algorithms are efficient. The algorithms should be able to compute the exact shortest distance from the pre-processed point cloud to an object, represented by a triangle mesh, that is moved through the point cloud. By allowing the distance to deviate from the true distance up to a certain limit, the point cloud may be simplified to a point cloud having fewer points.

In order to avoid computing the distance from the object to all points in the point cloud, the point cloud must be structured in such a way that facilitates deriving strict criteria for the relevance of points therefore making it possible to neglect points that are far away. The resulting shortest distances must be continuous when the object moves, disallowing any modification to the point cloud that would violate this requirement.

After constructing the algorithms and proving that they are suitable for the intended purpose, they are implemented in C++ and lastly, simulations are carried out in order to evaluate how the algorithms perform in practice.

The solution to this problem must satisfy the following criteria:

- It should be able to process point clouds having billions of points, independent of the amount of RAM available, given enough hard drive space.
- The set of computed distances must be continuous, which means that no subset of points can be completely removed if containing the point closest to the object.
- Computing the shortest distance between the massive point cloud and the object should be as fast as possible.
- For very large point clouds, the point cloud may be simplified by allowing the computed distance to deviate from the true distance as long as it never exceeds a specified limit.

- The time to compute the shortest distance should not vary significantly when the object moves. This means that waiting several seconds in order to import a large subset of the point cloud is not allowed.

1.3 Limitations

This work is based on the availability of an implementation and the efficiency of an in-core algorithm to compute the shortest distance from a small point cloud to a triangle mesh. The aim of the thesis is to process and structure a massive point cloud in such a way that makes it possible to neglect a large amount of points that are far away from the object. The remaining points can fit in-core and then make use of the existing algorithms to compute the shortest distance.

The work in this thesis also relies on the existence of an already generated path for the object passing through the point cloud and aims to verify that the correct shortest distance can be computed efficiently along this path. The algorithms developed can and will be used to generate a path, but the aim of the simulations in the thesis is to verify that the distance computation is correct and fast.

1.4 Outline

Chapter 2 will describe how to divide a massive point cloud into disjoint subsets that are small enough to fit in-core. Chapter 3 will describe how to pre-process these subsets to allow for fast-query. If the point cloud is very large and the exact shortest distance is not of interest, Chapter 4 describes how the point cloud can be simplified to a new point cloud with fewer points such that the computed shortest distance never exceeds a specified user-tolerance. Build times are analyzed in Section 4.4. Chapter 5 describes implementation choices and how to work with massive point clouds out-of-core. It also introduces PQP, which is a fast way to compute the distance between two small geometrical models. In order to test the algorithms on a real point cloud, Chapter 6 applies the theoretical results in practice. A discussion and a conclusion is provided in Chapter 7.

2

Organizing the point cloud

This chapter presents the simple structures of the point cloud and how it should be pre-processed in order to allow for fast distance computation. It is described how the point cloud can be divided into disjoint subsets for which Chapter 3 aims to derive criteria for when a subset may contain the point closest to the object. The first algorithm considered in this chapter recursively divides the axis-aligned bounding box (AABB) of the original point cloud into two AABBs until all AABBs contain a suitable number of points. The second algorithm is a clustering algorithm that can be used to recursively cluster sets of points into a fixed number of clusters. Section 2.1 contains some elementary definitions and some notations that will be used throughout this thesis. Section 2.2 describes how the point cloud can be divided into disjoint AABBs, and Section 2.3 describes how to cluster the point cloud using the k-means method.

2.1 Preparation

This section will introduce some important definitions as a necessary preparation. The first definition will be what a point cloud is:

Definition 2.1. *A* ***point cloud*** *is a set of points* $\{p^i \,|\, p^i \in \mathbb{R}^n\}$ *in a coordinate system.*

The point cloud will always be denoted by P and its corresponding cardinality (number of points) by $|P|$. The object that is moved through the point cloud is denoted by S, and it is a closed and bounded set that could be either a triangulation or a point cloud. This thesis will, however, focus on the case when S is a triangulation. This thesis will focus on $n = 2$ or $n = 3$ unless stated otherwise. Denote by p_i, $i = 1,2,\dots,n$, the coordinates of a point $p \in \mathbb{R}^n$ and let $p^1, p^2, \dots, p^m$ denote m points where $p^i \in \mathbb{R}^n$. An important property of the point cloud is its axis-aligned bounding box which will be defined next.

Definition 2.2. *The **axis-aligned bounding box** (AABB) of a point cloud is defined to be the smallest axis-aligned box in a Cartesian coordinate system such that it contains all objects of interest.*

The AABB provides the opportunity to only consider a subset of $\mathbb{R}^n$ that contains the point cloud. When a subset of P is considered it will be denoted by Q, with the implication that $Q \subseteq P$. Multiple disjoint subsets of P will be denoted by Q_i and when no particular subset is of interest the index will be dropped.

2.2 Partitioning the point cloud

This section illustrates how the original AABB of the point cloud can be successively divided into new AABBs in order to obtain a partitioning of the original point cloud. Given the original point cloud, it is straightforward to find the AABB of P. Suppose that it is of interest to divide the point cloud into subsets of at most T points and that $|P| > T$. By splitting the original AABB along one of the coordinate directions two new AABBs can be created. There are several options on where to divide the AABBs, such as along the longest side or the coordinate direction with the largest variance.

Assume for now that the point cloud is stored in an out-of-core array and that it is possible to perform swaps of its elements. It would be convenient for each AABB to have its corresponding elements of the original point cloud as a contiguous piece of this array so that each AABB only needs the first and last index of its subset of points. In addition, an AABB with data Q will need to store the six values that determines its boundaries, i.e. $\min q_i$ and $\max q_i$ for all $q \in Q$ and for $i = 1,2,3$. The algorithm for partitioning the point cloud can be seen in Algorithm 1.

Algorithm 1 Partitioning of the point cloud

```
Create an empty list $L$ of AABBs
Compute the AABB of the original point cloud $P$
Add this AABB to the back of $L$
Initialize $i \leftarrow 0$
while $i <$number of elements in $L$ do
    if $|L(i)| > T$ then
        Divide $L(i)$ into two disjoint sets
        Create two new AABBs
        Perform swaps in $P$ to make the subsets of the new AABBs contiguous in $P$
        Add the two AABBs to $L$ after the last element
        Flag as deleted, but do not remove, $L(i)$
    end if
    $i \leftarrow i + 1$
end while
Create a new list without the AABBs that are flagged as deleted
```

2.3 The k-means method

This section will describe how to use k-means clustering in order to partition the point cloud. The k-means method is one of the oldest clustering methods whose aim is to divide a set of points into k clusters. The algorithm is based on work by Lloyd, [15], and is often referred to as Lloyd's algorithm. The clusters $C_1, \ldots, C_k$ have the properties $C_i \subseteq Q$, $C_i \cap C_j = \emptyset$ for $i \neq j$, and $\bigcup_{i=1}^{k} C_i = Q$. Each cluster has a centroid μ^i which is computed as the the mean value of all points belonging to cluster C_i. A point is assigned to the cluster of the centroid closest to the point. The k-means method for a set of points Q can be written as the optimization problem:

$$\underset{C_1,\ldots,C_k}{\text{minimize}} \sum_{j=1}^{k} \sum_{i \in C_j} \|q^i - \mu^j\|^2, \; \forall q^i \in Q. \tag{2.1}$$

By choosing to minimize Equation 2.1 in the Euclidean norm so that points are assigned to the closest centroid in the L_2-sense, the resulting convex hulls of the generated clusters will be disjoint [16]. The definition of convex hulls is given on page 8 in Definition 3.3

The k-means method relies on prior knowledge of how many clusters should be created, rather than how many points are allowed to be in each cluster. Since the clustering process may produce an uneven distribution of points between the clusters, it is hard to choose k such that no cluster contains more than T points. A recursive variation of k-means is more suitable where subsets that contain more than T points are clustered with a small value of k until all subsets contain no more than T points. A variation of Algorithm 1 that uses k-means instead of the AABBs can be seen in Algorithm 2.

Algorithm 2 Clustering the point cloud P using recursive k-means

```
Create an empty list $L$ of subsets (clusters)
Add $P$ to the back of $L$
Initialize $i \leftarrow 0$
while $i <$ number of elements in $L$ do
    if $|L(i)| > T$ then
        Initialize the centroids $\mu^1, \ldots, \mu^k$
        while repeat until convergence or maximal number of iterations exceeded do
            Assign the points to the cluster with the closest centroid in $L_2$
            Compute the new centroid $\mu^i$ as the mean value of the points in cluster $C_i$
        end while
        Add the $k$ new subsets to $L$ after the last element
        Flag as deleted, but do not remove, $L(i)$
    end if
    $i \leftarrow i + 1$
end while
Create a new list without the subsets that are flagged as deleted
```

The most critical part of the algorithm is choosing the initial centroids $\mu^1, \ldots, \mu^k$

since a poor initial configuration can lead to slow convergence or a suboptimal solution to the k-means problem. There are many different strategies for this problem, such as choosing k random points from the set of points. One recent approach is k-means++ [17] which guarantees an $O(\log k)$ competitive solution. The idea is based on the fact that convergence is expected to be faster if the initial centroids are far apart. Thus the k-means++ method chooses the first centroid at random and then for each point $q \in Q$ the distance to the closest centroid, denoted by $D(q)$, is computed. The next centroid is then picked with a probability proportional to $D(q)^2$, giving points that are further away from their closest centroid a much higher probability to be chosen. Note that this makes it impossible for a preexisting centroid to be chosen again, as long as $k \leq |Q|$, since $D(q) = 0$ in this case.

By using the k-means method, the point cloud can recursively be divided into disjoint subsets with at most T points by re-running k-means on each new cluster that has more than T points. This will be done for a small value of k such as $k = 2$ or $k = 3$. Figure 2.1 shows an example of the k-means algorithm on two-dimensional simulated data using $k = 5$ and the k-means++ initialization. It is clear that after a few iterations that the positions of the centroids have started to converge.

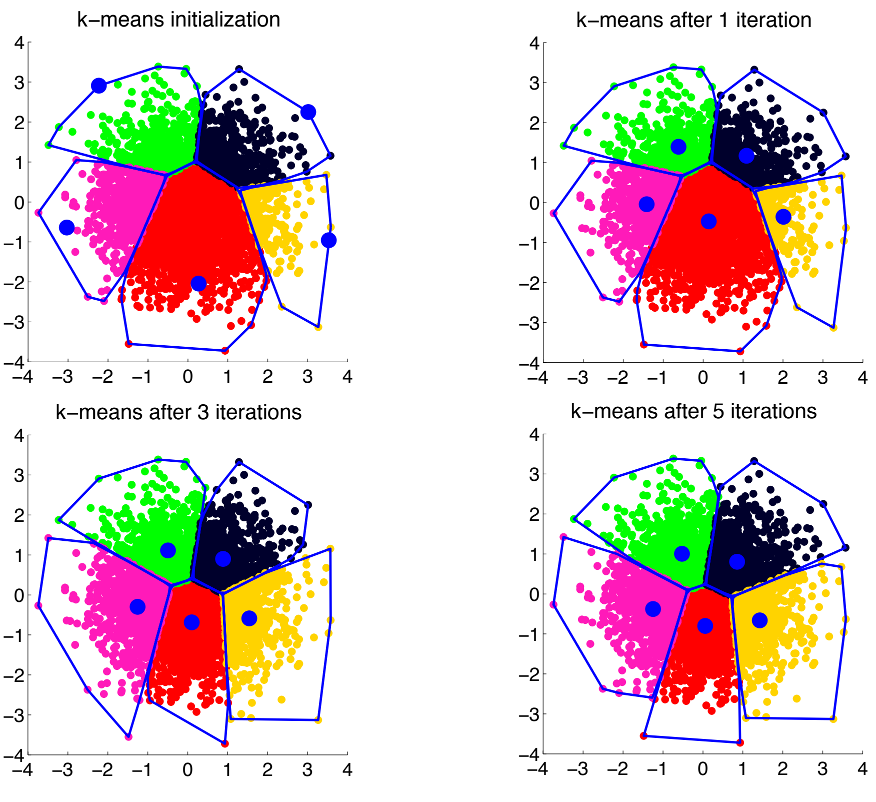

Figure 2.1: *The k-means method with k-means++ initialization for $k = 5$.*

3

Pre-processing of the point cloud for fast-query

This chapter describes how the organized point cloud from Chapter 2 can be pre-processed to allow for fast distance computation. Section 3.1 introduces the concept of convex hulls and some elementary results from optimization that will be necessary throughout this chapter. In Section 3.2 a lower bound on the distance from a set of points $Q \subseteq P$ to an object S, such that the convex hull of Q does not intersect with S, will be derived in 2D. The lower bound is only based on the extreme points of Q to allow a fast way to estimate how close the closest point in Q can be from S. This result is then generalized to 3D in Section 3.3. Section 3.4 shows how to use information from the last known position of the object in order to directly rule out subsets of P. Section 3.5 will demonstrate a case where the results from Section 3.3 perform poorly. In Section 3.6 it is investigated how a simple AABB would compare to using the convex hulls. Finally, Section 3.7 explains how to treat the case where S is located within a convex hull, for which the results from Section 3.3 do not apply.

3.1 Convex hulls and optimization

As preparation, the concepts of convex sets, convex combinations, and convex hulls are introduced, [18], which will frequently be used from now on.

Definition 3.1. *A set $C \subset \mathbb{R}^n$ is said to be **convex** if for every $x^1, x^2 \in C$ and every real number α, $0 < \alpha < 1$, the point $\alpha x^1 + (1-\alpha)x^2 \in C$.*

Definition 3.2. *A **convex combination** of a set $C \subset \mathbb{R}^n$ is of the form $\alpha_1 x^1 + \ldots + \alpha_n x^n$ where $\alpha_i \in \mathbb{R}^n$, $\alpha_i \geq 0$, $\sum_{i=1}^{n} \alpha_i = 1$ and where $\{x^1, \ldots, x^n\} \in C$.*

Definition 3.3. *Let $C \subset \mathbb{R}^n$. The **convex hull** of C is the set of all convex combinations of C:*

$$\mathcal{CH}(C) = \left\{ \alpha_1 x^1 + \ldots + \alpha_n x^n \,\middle|\, \alpha_1, \ldots, \alpha_n \geq 0; \sum_{i=1}^{n} \alpha_i = 1 \right\}, \tag{3.1}$$

where $\{x^1, \ldots, x^n\} \in C$.

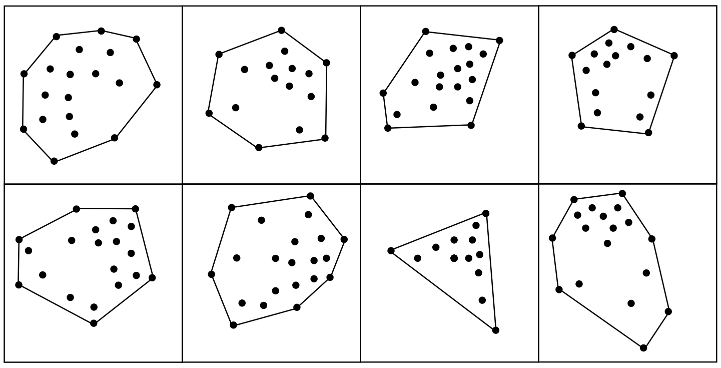

Figure 3.1: *Convex hulls in 2D constructed from disjoint subsets.*

The concept of the extreme points of a convex hull is very important and is introduced next:

Definition 3.4. *A point x in a convex set C is said the be an **extreme point** of C if there are no two distinct points x^1 and x^2 in C such that $x = \alpha x^1 + (1 - \alpha)x^2$ for some α, $0 < \alpha < 1$.*

Denote by $E(Q)$ the set of extreme points of $\mathcal{CH}(Q)$, where $E(Q) \subseteq Q$. The use of extreme points is very important because they are all that is needed to determine the convex hull of a set of points. This follows from the following theorem [18]:

Theorem 3.1. *A closed and bounded convex set in $\mathbb{R}^n$ is equal to the closed convex hull of its extreme points.*

This theorem is important because it means that the extreme points of the convex hull, made up by a subset of points from the point cloud, will be enough to determine the convex hull. The extreme points will be very important throughout this thesis since they can be used to bound the distance to a subset of points. The distance between two points, from a point to a set, and from a set to a set is defined according to:

Definition 3.5. *Let Q, $R \subset \mathbb{R}^n$ and $q, r \in \mathbb{R}^n$. Define:*

$$\begin{aligned} d(q,r) &= \|q - r\|_2 \\ d(q,R) &= \inf_{r \in R} d(q,r) \\ d(Q,R) &= \inf_{q \in Q,\, r \in R} d(q,r) \end{aligned} \tag{3.2}$$

As a necessary preparation, a theorem stating that there exists a nonempty set of optimal solutions to the problem of minimizing a continuous function over a non-empty, closed, and bounded set is needed. This theorem is well-known as Weierstrass' theorem, for which a proof and all necessary definitions can be found in [19].

Theorem 3.2. *Let D be a non-empty, closed, and bounded subset of $\mathbb{R}^n$ and let $f : D \to \mathbb{R}$ be a continuous function on D. Then the problem*

$$\begin{aligned} &\text{minimize}\, f(x) \\ &\text{subject to}\, x \in D \end{aligned} \tag{3.3}$$

has a global minimum.

This theorem guarantees the existence of a bounded set of global minimas that is also non-empty. Before ending this section, the aim is to prove that the point belonging to the convex hull that is closest to the object S must lie on the boundary, given that $\mathcal{CH}(Q) \cap S = \emptyset$. This proof will only be carried out for full-dimensional polytopes, which is defined next.

Definition 3.6. *$K \subset \mathbb{R}^n$ is a **polytope** if it is the convex hull of finitely many points in $\mathbb{R}^n$.*

Definition 3.7. *A polytope is called **full-dimensional** in $\mathbb{R}^n$ if $n + 1$ of its extreme points do not lie in a common hyperplane.*

Note that according to this definition, a polytope with less than $n+1$ points can never be full-dimensional in $\mathbb{R}^n$. The work in this thesis will only consider full-dimensional polytopes and it will be assumed from now on that all polytopes are full-dimensional. This is also known as assuming general position in computational geometry [20]. Degenerate cases are not of interest and can be made full-dimensional in $\mathbb{R}^n$, if the subset has at least $n + 1$ points, by perturbing the points. In order to define what the boundary of a full-dimensional convex hull is, the terms half-space and face are necessary to define.

Definition 3.8. *A closed **half-space** in $\mathbb{R}^n$ is a set $\{x \in \mathbb{R}^n : a \cdot x \geq b\}$ for some $a \in \mathbb{R}^n \backslash \{0\}$ and $b \in \mathbb{R}^n$; the hyperplane $\{x \in \mathbb{R}^n : a \cdot x = b\}$ is its boundary.*

Definition 3.9. *A **face** of a convex polytope is any intersection of the polytope with a half-space such that none of the interior points of the polytope lie on the boundary of the half-space.*

Let $\partial \mathcal{CH}(Q)$ denote the boundary of the convex hull and define this as the union of the faces of the polytope. The next theorem will show that if $\mathcal{CH}(Q) \cap S = \emptyset$ and $\mathcal{CH}(Q)$ is a full-dimensional polytope, the point on $\mathcal{CH}(Q)$ that is closest to S must lie on $\partial \mathcal{CH}(Q)$.

Theorem 3.3. *Let $S \subset \mathbb{R}^n$ be a closed and bounded set and let $\mathcal{CH}(Q) \subset \mathbb{R}^n$ be a closed and bounded convex hull that is a full-dimensional polytope. If $\mathcal{CH}(Q) \cap S = \emptyset$, the point in $\mathcal{CH}(Q)$ closest to S must lie on the boundary.*

Proof. Let $V = (S\,, \mathcal{CH}(Q)) \subseteq \mathbb{R}^n \times \mathbb{R}^n \cong \mathbb{R}^{2n}$ and define the function

$$\begin{aligned} &f : \mathbb{R}^{2n} \to \mathbb{R} \\ &f(x) = \left\| x_S - x_{\mathcal{CH}(Q)} \right\|_2 \end{aligned} \tag{3.4}$$

where $x_S \in S$ and $x_{\mathcal{CH}(Q)} \in \mathcal{CH}(Q)$. From Theorem 3.2

$$\exists y = \left(y_S\,, y_{\mathcal{CH}(Q)}\right) \text{ such that } y = \underset{x\in V}{\arg\min}\, f(x). \tag{3.5}$$

If $y_{\mathcal{CH}(Q)}$ lies on the boundary of $\mathcal{CH}(Q)$, the proof is completed. If $y_{\mathcal{CH}(Q)}$ is an interior point of $\mathcal{CH}(Q)$, construct the line

$$L = \left\{ y_{\mathcal{CH}(Q)} + \left(y_S - y_{\mathcal{CH}(Q)}\right) t \,|\, 0 \le t \le 1,\, t \in \mathbb{R} \right\} \tag{3.6}$$

between the two points. Since $y_{\mathcal{CH}(Q)}$ is an interior point of a closed set, take $\delta > 0, \delta \in \mathbb{R}$ small enough so that $\hat{y} = y_{\mathcal{CH}(Q)} + (y_S - y_{\mathcal{CH}(Q)})\delta \in \mathcal{CH}(Q)$. This implies that

$$\|\hat{y} - y_S\|_2 < \|x - y_S\|_2 \tag{3.7}$$

This contradicts the fact that $y_{\mathcal{CH}(Q)}$ was the point in $\mathcal{CH}(Q)$ closest to y_S. Hence the optimal point $y_{\mathcal{CH}(Q)}$ must lie on the boundary of $\mathcal{CH}(Q)$. □

3.2 Distance from the object to the convex hull in 2D

This section aims to derive a strict criteria based on only the extreme points of Q, that is fast to evaluate and can be used to determine whether a point of Q can be the point closest to S. Such a criteria will allow neglecting subsets Q that are far away from S, without computing the distance from these points to S. All results will assume that $\mathcal{CH}(Q) \cap S = \emptyset$ and Section 3.7 will explain how to treat the situation where this may not hold.

It will be assumed in this section that S and Q are objects in 2D, which will later be extended to 3D. Some notations for a few important points of the convex hull that will be used throughout this chapter will be defined next, where the definition for adjacent extreme points can be found in [21].

Definition 3.10. *Define $u \in \mathcal{CH}(Q)$ to be the point that is closest to S,*

$$u = \underset{x\in \mathcal{CH}(Q)}{\arg\min}\, d(x, S). \tag{3.8}$$

Definition 3.11. *Let $v \in E(Q)$ be the extreme point closest to S,*

$$v = \underset{x\in E(Q)}{\arg\min}\, d(x, S). \tag{3.9}$$

Definition 3.12. *Let $w \in E(Q)$ to be the extreme point that is closest to u,*

$$w = \arg\min_{x \in E(Q)} d(x, u). \tag{3.10}$$

Definition 3.13. *Define by $d_e \in \mathbb{R}$ the length of the longest edge in $\mathcal{CH}(Q)$ between any two adjacent extreme points.*

Note that $u \neq v$ can hold since u may not be an extreme point. It is also possible that $u = v = w$ in which the closest point is an extreme point. A situation where all points are distinct is illustrated in Figure 3.2.

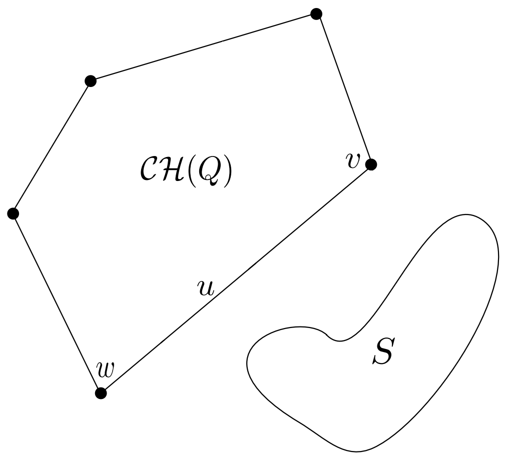

Figure 3.2: *Illustration of the points u, v and w used in Theorem 3.4.*

The following theorem shows a straightforward way to approximate the minimum distance from a convex hull $\mathcal{CH}(Q)$ to S.

Theorem 3.4. *Assume that $\mathcal{CH}(Q) \cap S = \emptyset$. Then $\frac{1}{2}d_e$ is an upper bound of $|d(u,S) - d(v,S)|$. This means that u cannot be closer to S than $d(v,S) - \frac{1}{2}d_e$, that is:*

$$d(Q,S) = d(u,S) \geq d(v,S) - \frac{1}{2}d_e. \tag{3.11}$$

Proof. From Theorem 3.3 it must hold that $u \in \partial\mathcal{CH}(Q)$. There must always be an extreme point on the edge where u is located that is within $\frac{1}{2}d_e$ from u since d_e is the length of the longest edge. The use of the triangle inequality yields,

$$d(w,S) \leq d(w,u) + d(u,S) \leq \frac{1}{2}d_e + d(u,S). \tag{3.12}$$

By the definition of v it must hold that $d(v,S) \leq d(w,S)$ and hence

$$d(u,S) \geq d(w,S) - \frac{1}{2}d_e \geq d(v,S) - \frac{1}{2}d_e, \tag{3.13}$$

which proves the claim. □

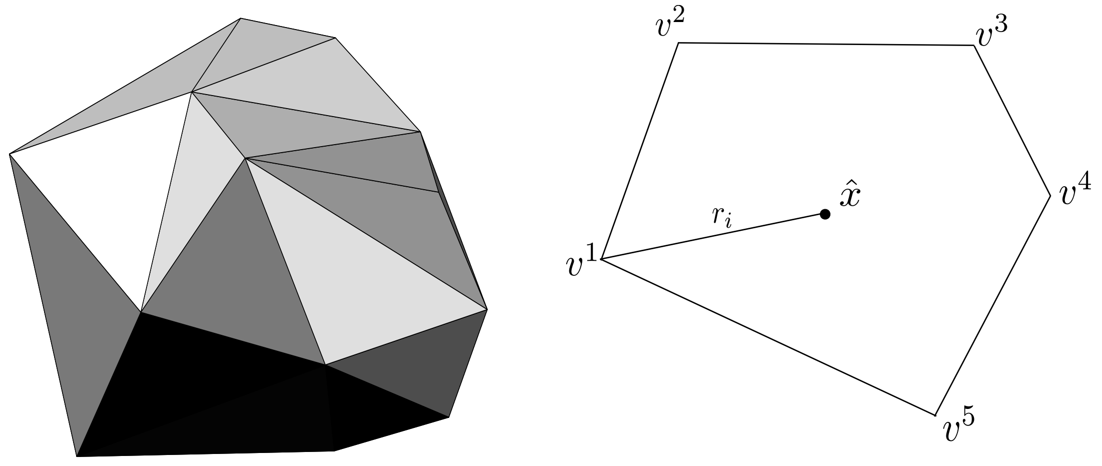

Figure 3.3: *(Left) A full-dimensional convex hull in 3D. (Right) A face with five vertices illustrating the location of $\hat{x}$ and also r_i, the distance from $\hat{x}$ to its closest vertex.*

This presents a simple algorithm to approximate the minimum distance from S to Q:

1. Compute the value of d_e for $\mathcal{CH}(Q)$.
2. Find the closest extreme point $v \in E(Q)$ to S.
3. The approximation $d(Q,S) \geq d(v,S) - \frac{1}{2}d_e$ can be used to quickly find a lower bound for how close any point in Q is to S.

3.3 Distance from the object to the convex hull in 3D

The point clouds of interest consist of points in three dimensions which is why the results from the previous section have to be extended to 3D. For a three dimensional point cloud the convex hull is a polyhedron, where the body is bounded by its polygonal faces. A polytope in 3D is called a polyhedron and its faces will be polygons containing their interior. An example of a polyhedron can be seen in Figure 3.3. In order to extend Theorem 3.4 to 3D, the faces of the convex hull will be considered instead of the edges in 2D. For a face $F_i \in \partial\mathcal{CH}(Q)$, the point that corresponds to the midpoint of the longest edge in 2D must be replaced. The definitions of u, v and w from Section 3.2 are still valid and will still be used in this section. In order to replace $\frac{1}{2}d_e$, some definitions are necessary.

Definition 3.14. *Consider a face F_i of $\mathcal{CH}(Q)$ with vertices $v^1, \ldots, v^r$. Define*

$$r_i = \max_{x \in F_i} \min_{j \in \{1,\ldots,r\}} d(x, v^j). \tag{3.14}$$

and

$$\hat{x} = \arg\max_{x \in F_i} \min_{j \in \{1,\ldots,r\}} d(x,v^j). \tag{3.15}$$

Definition 3.15. *Let $r_{\max}$ be the largest r_i considering all m faces in $\mathcal{CH}(Q)$. Define*

$$r_{\max} = \max_{i \in \{1,\ldots,m\}} r_i. \tag{3.16}$$

This defines r_i as the maximal possible distance from a point in F_i to its closest vertex in F_i and $\hat{x}$ as the point for which this maximum is attained. An illustration of the location of $\hat{x}$ for a face with five vertices can be seen in Figure 3.3. The next theorem will show that $r_{\max}$, the largest value of r_i, is a generalization of the value $\frac{1}{2}d_e$ that makes it possible to derive a similar inequality in 3D.

Theorem 3.5. *Given a full-dimensional convex hull with faces $F_1,\ldots,F_m$, such that*

$$\bigcup_{i=1}^{m} F_i = \partial\mathcal{CH}(Q). \tag{3.17}$$

Then,

$$d(S,v) - r_{\max} \leq d(S,u) = d(S,Q). \tag{3.18}$$

Proof. Theorem 3.3 guarantees that $u \in \partial\mathcal{CH}(Q)$. Consider the face where u is located and denote it by F_i. There must then be an extreme point on this face that is no further away from u than r_i, which follows directly from the definition of r_i. The triangle inequality implies that

$$d(S,u) \leq d(S,w) + d(w,u) \leq d(S,v) + r_i \leq d(S,v) + r_{\max}, \tag{3.19}$$

which completes the proof. □

Theorem 3.5 provides a fast way to check how close any point of a convex hull can potentially be from S by comparing the distances to all of the extreme points and use the distance to the closest extreme point and the pre-computed value of $r_{\max}$ in order to evaluate the lower bound in Equation 3.18. It should be clarified that it is unknown what face u lies in, which is why the inequality is based on $r_{\max}$.

The special case of a triangular face is considered next for which it is possible to derive an exact expression for r_i. Consider a triangular face with vertices v^1, v^2, v^3; there are two cases to consider in order to compute $\hat{x}$ and the value of r_i for that face. The first case is that of an acute triangular face and the second case is when the face is obtuse.

For the first case, the center of the circumscribed circle will lie on the triangular face. This center will thus be the point $\hat{x}$ and the radius of the circle will be the distance to any of the vertices. Given a triangular face, an expression for the radius r can be derived directly from the Law of Sines. Given sides a, b, c and angles A, B, C the Law of Sines states that

$$\frac{\sin(A)}{a} = \frac{\sin(B)}{b} = \frac{\sin(C)}{c} = \frac{1}{2r}, \tag{3.20}$$

where r is the radius of the circumscribed circle. The relation between the usual Law of Sines and the radius of the circumscribed circle can be conveniently derived by considering the inscribed angle at the center of the circumscribed circle.

By using that the area of the triangle is given by

$$\frac{bc\sin(A)}{2} = \frac{1}{2}|(v^2 - v^1) \times (v^3 - v^1)| \tag{3.21}$$

it follows that

$$r = \frac{\|v^1 - v^2\|\|v^2 - v^3\|\|v^3 - v^1\|}{2\|(v^2 - v^1) \times (v^3 - v^1)\|}. \tag{3.22}$$

In the second case of an obtuse triangle, the center of the circumscribed circle will lie outside the face and will therefore not be the desired point $\hat{x}$. It can be shown, in this case, that $\hat{x}$ can never be an interior point of the face and must therefore lie on the boundary. This follows from the following Lemma:

Lemma 3.1. *Let $F \in \mathcal{CH}(Q)$ be an obtuse triangular face, then $\hat{x}$ cannot be an interior point.*

Proof. By means of contradiction, assume that $\hat{x}$ is an interior point. Since one angle is larger than 90^o the distance to all of the vertices cannot be the same, since this is only true for the center of the circumscribed circle, which is unique and in this case lies outside the triangle. If $\hat{x}$ is closest to exactly one vertex it can be moved away from this vertex along the line between the points which is a contradiction. In the case $\hat{x}$ is closest to two vertices, $\hat{x}$ can be moved away from both vertices except for the case when all three points lies on the same line, but then $\hat{x}$ lies on the boundary contradicting that $\hat{x}$ is an interior point. □

It is now known that $\hat{x}$ lies on the boundary of the obtuse triangular face F. The next step is to show that $\hat{x}$ must have the same distance to exactly two of the vertices. This lemma will be very useful later.

Lemma 3.2. *Given an obtuse triangular face $F \in \mathcal{CH}(Q)$, $\hat{x}$ will have the same distance to exactly two vertices and the third vertex is further away.*

Proof. Theorem 3.1 gurantees that $\hat{x}$ must lie on the boundary. Now assume that $\hat{x}$ has exactly one point as its closest vertex. Then $\hat{x}$ can be moved away from this point along the edge and the distance from the closest vertex will keep increasing until the distance to the second closest vertex is exactly the same. This contradicts that $\hat{x}$ can be closest to only one vertex. □

It has now been shown that $\hat{x}$ is closest to exactly two vertices and that it also lies on one of the edges of F. The next step is to identify the edge where $\hat{x}$ is located. The following lemma shows that $\hat{x}$ must lie on the longest edge:

Lemma 3.3. *Given an obtuse triangular face, then $\hat{x}$ lies on the longest edge of the triangle.*

Proof. Assume that the edges of the triangle are such that $c \leq b \leq a$ just as in the bottom triangle in Figure 3.4. Assume that $\hat{x}$ lies on the longest side of the triangle and is such that $s = d(v^1, \hat{x}) = d(v^3, \hat{x})$, in which case $s = \frac{b}{2\cos(C)}$. For this to be possible, it is necessary to verify that

$$d(v^2, \hat{x}) - d(v^3, \hat{x}) = (a - s) - s = a - \frac{b}{\cos C} > 0, \tag{3.23}$$

which shows that the third vertex is further away. An application of the triangle inequality yields

$$b = d(v^1, v^3) < d(v^1, \hat{x}) + d(\hat{x}, v^3) = 2s \tag{3.24}$$

which implicates that $b/2 < s$. This shows that no point on any other edge can be further away from its closest vertex since $c \leq b$ and no point on the edge of length b can be further away than $b/2$ from its two closest vertices. Hence, by construction, $\hat{x}$ lies on the longest edge of the triangle. $\square$

Now that it is known which edge $\hat{x}$ must lie on, only two possible points remain: $d(v^1, \hat{x}) = d(v^3, \hat{x})$ or $d(v^1, \hat{x}) = d(v^2, \hat{x})$. Denote these two distances by s and t respectively. The next theorem makes it possible to explicitly compute r_i without having to find the exact location of $\hat{x}$.

Theorem 3.6. *Given a triangular face with one angle exceeding* 90^o *with* $c \leq b \leq a$ *and* $A > 90^o$. *Then the point that is furthest away from the closest vertex has distance*

$$\frac{b}{2\cos C} \tag{3.25}$$

from its closest vertex.

Proof. Lemma 3.1, 3.2, 3.3 guarantees the existence of such a point and only two possibilities remain. It follows directly that $s = \frac{b}{2\cos C}$ and $t = \frac{c}{2\cos B}$, for which the aim is to show that $s \geq t$. It was shown in Lemma 3.3 that in the case of $d(v^1, \hat{x}) = d(v^3, \hat{x})$, the third vertex lies further away so showing that $s \geq t$ will complete the proof. From using the Pythagorean inequality it holds that $a^2 > b^2 + c^2$ and hence

$$(a^2 - b^2 - c^2)(b^2 - c^2) \geq 0. \tag{3.26}$$

After dividing both sides by abc this inequality can be rearranged to obtain

$$\frac{b(a^2 + c^2 - b^2)}{ac} \geq \frac{c(a^2 + b^2 - c^2)}{ab} \tag{3.27}$$

and using the law of cosines it follows directly that

$$\frac{b}{\cos C} \geq \frac{c}{\cos B}, \tag{3.28}$$

which is exactly the same as $s \geq t$. $\square$

In summary, Equation 3.22 can be used to find r_i for an acute triangular face and Equation 3.25 for an obtuse triangular face. After the convex hull has been built the values of r_i are computed for each face and only the largest such value is saved as the corresponding value of $r_{\max}$ for this convex hull.

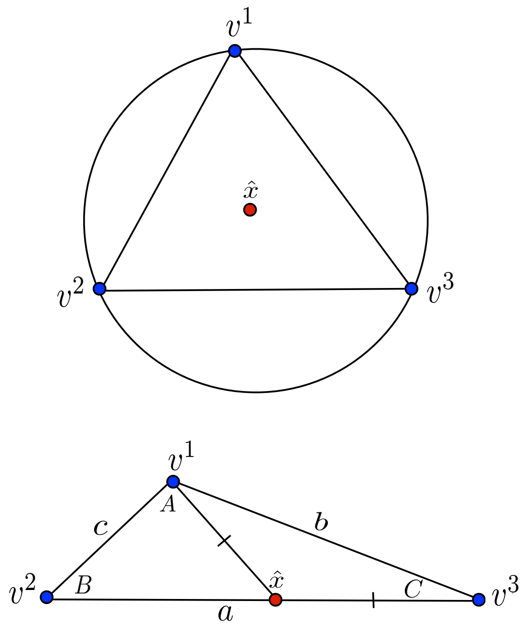

Figure 3.4: *Illustration of the location of $\hat{x}$ in the case of an (Upper) acute (Lower) obtuse triangular face of a convex hull.*

3.4 A fast exclusion theorem

In this section a very simple criterion is constructed to make it possible to exclude a majority of the convex hulls without computing any distances when it is clear that they are too far away from the object to contain the closest point. It will be assumed that the object S is moved through the point cloud with discrete time-steps t_i, where i will be referred to as the iteration.

Definition 3.16. *Define by α_i the largest displacement of any point on S at iteration i.*

Definition 3.17. *Define by d_j the distance from the extreme points of convex hull number j to S.*

Definition 3.18. *Let $d_{\min} = d(P,S)$.*

The idea is that α_i can be used to bound how much closer a convex hull can be to the object compared to the previous iteration. The result is stated in the following Theorem:

Theorem 3.7. *Suppose that P has been divided into m disjoint sets for which the convex hulls have been constructed and that d_j, or a lower bound, is known for each convex hull. If S is moved at most α_i, no point in convex hull number j can be closer to S than*

$$\max(0, d_j - \alpha_i - r_{\max,j}) \tag{3.29}$$

and there must be at least one point that is no further away from S than $d_{\min} + \alpha_i$.

Proof. The first statement follows from the fact that S has moved at most α_i and that no point in Q_j was closer than $d_j - r_{\max,j}$ before S moved, which yields that no points in the convex hull can be closer than $\max(0, d_j - \alpha_i - r_{\max,j})$. Since S has moved at most α_i, the point that was previously at a distance $d_{\min}$ away from S cannot be further away than $d_{\min} + \alpha_i$. This implies the second statement. □

This offers the opportunity to compute all the values of d_j prior to computing the shortest distance for the first time and each time S moves, compute the value of α_i and set $d_{\min} \leftarrow d_{\min} + \alpha_i$ and $d_j \leftarrow d_j - \alpha_i$ until d_j needs to be updated. If $d_j - r_{\max,j} > d_{\min}$, the convex hull number j can be ignored completely since there must be a point that is at most $d_{\min}$ away from S. Note that this means that d_j will be used as a lower bound until the true distance is computed, when it is expected that the convex hull is close to S.

This inequality based on α_i is designed for quick evaluation, however it should be mentioned that it can be very weak in the sense that it does not take the direction of movement into account. If S is moved away from a convex hull so that $d(\mathcal{CH}(Q),S)$ increases, the lower bound will be pessimistic since d_j will be subtracted by α_i.

The complete algorithm for the pre-processing for fast distances queries and how to compute the shortest distance from P to S can be seen in Algorithm 3, which addresses the building of the convex hulls and how to decide which convex hulls are close to the object. A subindex has been added to the shortest distance, that is denoted $d_{\min,i}$ for iteration i, in order to compute a series of shortest distances to be able to see how close S is to P along its path.

3.5 Quality of approximation when using convex hulls

It will here be shown that the approximation of using the extreme points of $\mathcal{CH}(Q)$ can perform poorly. Consider a triangular face of a convex hull with base b and height h, where $h \ll b$, such as in Figure 3.5. Let S consist of only one point s and let it be located along the normal of this face that intersects the face at p^3. This implies that p^3 is the extreme point of this convex hull that is closest to s. Assume also that $r_{\max}$ is located on this face. It can be verified that $r_{\max} = \frac{b}{4} + \frac{h^2}{b}$. Assume that

$$d(p^3,s) = \frac{b}{4} + \frac{h^2}{b} + \epsilon \tag{3.30}$$

so that

$$d(p^3,s) - r_{\max} = \epsilon. \tag{3.31}$$

Let $b \to \infty$ by separating p^1 and p^2 in the direction of the base line. Move s at the same time, along the normal of the face passing through p^3, in order to maintain $d(p^3,s) = \frac{b}{4} + \frac{h^2}{b} + \epsilon$. Then $d_{\min} = d(p^3,s) \to \infty$ even though $d(p^3,s) - r_{\max} = \epsilon$. This shows that the real minimum distance goes to ∞ but the approximation is still equal to ϵ showing

Algorithm 3 Algorithm for the preprocessing for fast-query for a moving object

```
Data: Subsets $Q$ of Point cloud $P$, Object $S$ moving with discrete time-steps

for each subset $Q$ do
    Read all points in subset $Q$
    Compute the convex hull and separate the extreme points from other points
    Compute $r_{\max}$
end for

for each convex hull $\mathcal{CH}(Q)$ do
    Compute $d_j$, the distance from $S$ to the closest extreme point
end for

Compute $d_{\min,0} \leftarrow \min_j d_j$ as an initial upper bound
$i \leftarrow 1$
while Object is moving do
    Move $S$ to the next position and compute $\alpha_i$
    $d_{\min,i} \leftarrow d_{\min,i-1} + \alpha_i$
    for each convex hull number $j$ do
        $d_j \leftarrow d_j - \alpha_i$
        if $d_j \leq d_{\min,i}$ then
            Update $d_j$ and set $d_{\min,i} = \min(d_j, d_{\min,i})$
            if $d_j - r_{\max,j} \leq d_{\min,i}$ then
                Compute the distance from $S$ to the points that are not extreme points
                and update $d_{\min,i}$ if any point is closer to $S$ than $d_{\min,i}$
            end if
        end if
    end for
    $i \leftarrow i + 1$
end while
```

that the approximation is far from the true value. This is not unexpected since Theorem 3.5 only gives a lower bound of the distance, which may not always be close to the true value.

It should be mentioned that $r_{\max}$ is bounded by the size of the convex hull so that there is an upper bound for how bad the approximation can be.

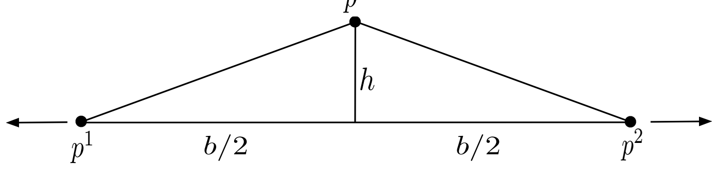

Figure 3.5: *An example of a triangular face for when using the extreme points to approximate the distance from the subset to S can give a weak bound. The point s is located above p^3.*

3.6 A quality comparison between convex hulls and AABBs

The construction of convex hulls will be carried out using the Quickhull algorithm [22], which is known to have complexity $O(n \log n)$. This can be compared to computing a simple AABB around each subset, which can be done in $O(n)$.

It is clear that the convex hull will always be contained within the AABB, making it more accurate and possibly yielding a much better lower bound of the distance computed from Theorem 3.5. In fact, if an AABB has side lengths $c \leq b \leq a$ the corresponding value of $r_{\max}$ for this AABB would be $\frac{1}{2}\sqrt{a^2+b^2}$, if the corner points are taken as the extreme points in this case. The difference between the approximation for a convex hull compared to an AABB is actually even larger since the closest corner point of the AABB may not be a part of the point cloud whereas the closest extreme point of the convex hull is always in the point cloud.

It will now be demonstrated that the value of $r_{\max}$ from using an AABB can be significantly larger compared to using convex hulls. To this cause consider the line

$$L = \{x + (y-x)t \,|\, 0 \leq t \leq 1, t \in \mathbb{R}\} \tag{3.32}$$

and let $x = (a,a,a)$ and $y = (b,b,b)$ where $a < b$ so that the three dimensional bounding box has side lengths $b - a$. Consider placing slightly perturbed points around this line that are contained in the bounding box so that

$$r_{\max} = \delta \ll b - a. \tag{3.33}$$

The corresponding value of $r_{\max}$ for the bounding box will be equal to $(b-a)/\sqrt{2}$ which is independent of δ.

By increasing the number of points so that $\delta \to 0$, it follows that the minimum distance estimation of Theorem 3.5 approaches the real minimum distance. For an AABB, the minimum distance approximation will not change after adding more points because they are all contained in the AABB and do not affect its size. Therefore, in this case, as the error approaches 0 for the convex hull it remains fixed for the bounding box.

The points from the original point cloud are expected to be tightly distributed in clusters, which can be planes corresponding to walls that may not be axis-aligned, in which case the convex hull of the should give a better bound than the AABB.

3.7 If the object resides inside a convex hull

Throughout Chapter 3 it was assumed that the convex hull and S did not intersect. If S would lie inside the convex hull, the results from Chapter 3 would not hold since a point that is not an extreme point might be closer than a point residing on the boundary of the convex hull. It is therefore necessary to make the assumption that the object can never be entirely located within a convex hull. It can be be guaranteed that the object can never be inside a convex hull by making sure that the optimal bounding box of the object is larger than the optimal bounding boxes of the convex hulls. For this cause the bounding boxes do not have to be axis-aligned.

One solution would be to always check the points that are not extreme points of the convex hull number k with the closest extreme point and check the points that are not extreme points of all convex hulls j with extreme points, such that $d_j - r_{\max,j} - r_{\max,k} \leq d_k$. By checking these additional sets of points that are not extreme points it can be guranteed that even if the object lies within a convex hull, the points that are not extreme points of that set will also be checked. Depending on the geometry, this solution will also work in cases where the point cloud is much larger than the object thus making it hard to bound the sizes of the convex hulls without forcing them to contain too few points.

An additional possibility would be to save the convex hull that was closest during the last iteration. Under the assumption that the object is moved in small increments, if no sets of points that are not extreme points are checked during an iteration, the set of points that are not extreme points belonging to the convex hull that had the closest point during the last iteration can be checked since the fact that no points that are not extreme points had to be checked indicates that the object might be located inside one of the convex hulls. This is not necessary if the object cannot fit in the convex hulls, as the first solution suggested.

4

Simplification of the point cloud

For very large point clouds consisting of several billions of points, it may be of interest to simplify the point cloud, with the drawback of not being able to compute the exact shortest distance from S to P. If an error of size ϵ is allowed when computing the shortest distance between S and P, the point cloud may be simplified to a point cloud $\bar{P}_\epsilon$ with fewer points. The main idea is that points that are close in P may be represented by fewer points in $\bar{P}_\epsilon$ and still be able to guarantee continuous shortest distances. Two methods will be investigated in this section, the first being a grid-based partitioning of the point cloud and the second by clustering according to the k-means method, introduced in Section 2.3. Section 4.4 will analyze the build times of the different algorithms introduced in Chapter 2, Chapter 3, and this chapter. Finally, an overview of the work presented so far in this thesis is given in Chapter 4.5.

4.1 The grid-based partitioning

This section aims to introduce a simplification scheme that is based on partitioning the point cloud by small axis-aligned grid boxes with diagonal 2ϵ, where $\epsilon \in \mathbb{R}$, $\epsilon > 0$ is the maximal allowed error in the computed shortest distance.

Definition 4.1. *A point cloud $\bar{P}_\epsilon$ is referred to as a simplification of P if it satisfies:*

1. $|\bar{P}_\epsilon| \leq |P|$
2. $|d(P,R) - d(\bar{P}_\epsilon,R)| \leq \epsilon, \forall R \subseteq \mathbb{R}^n$.

This definition states that a simplification may have fewer points than the original point cloud under the condition that the error in the computed shortest distance never exceeds ϵ. Section 4.3 will show that the second requirement can be guaranteed if

$$\forall p \in P : \min_{\bar{p} \in \bar{P}_\epsilon} d(p, \bar{p}) \leq \epsilon. \tag{4.1}$$

This means that as long as there is a point $\bar{p} \in \bar{P}_\epsilon$ for each $p \in P$ that is no further away than ϵ, the error in the shortest distance will never exceed ϵ.

Referring back to the grid-based partitioning, two points that are closer than 2ϵ may be represented by only one point in $\bar{P}_\epsilon$. The grid-based partitioning utilizes this fact by partitioning the original AABB into disjoint axis-aligned grid boxes with side length $\frac{2}{\sqrt{n}}\epsilon$ so that the diagonal is 2ϵ and all points that fall into the same axis-aligned grid box may be represented by its midpoint in $\bar{P}_\epsilon$. In order to allow for this partitioning, the original AABB of P can easily be extended to make all side lengths divisible by $\frac{2}{\sqrt{n}}\epsilon$. Extending the bounding box will still guarantee that it contains P.

A data structure that can be used to simplify P is a hash-table. The idea is to map each point into its corresponding axis-aligned grid box and add this box to the hash-table if it does not already exist. The choice of using a hash-table is very suitable here since it makes it possible to both add and find axis-aligned grid boxes in $O(1)$ time-complexity. This method is memory-efficient and suitable since the point cloud is expected to be sparse. In the case of a dense point cloud, it would be possible to perform this step in parallel and create all possible grid boxes instead of using the hash-table. How a point from the current subset of P can be mapped to the hash-table will be described next.

The position of the corner of the AABB with the smallest coordinate value in all directions will be denoted by $b \in \mathbb{R}^n$. Given a point $p \in P$, the axis-aligned grid box in which p resides can be denoted by the integer representation

$$\text{ind}_i = \left\lfloor \frac{p_i - b_i}{\frac{2}{\sqrt{n}}\epsilon} \right\rfloor . \tag{4.2}$$

Assume that N_i boxes can be stored in coordinate direction i, where N_i is easily computed from the size of the AABB. The mapping $F(p)$ that gives each axis-aligned bounding box a unique integer index can be defined as

$$F(p) := \sum_{i=1}^{n} \text{ind}_i \prod_{j=i+1}^{n} N_j . \tag{4.3}$$

This will map the axis-aligned grid boxes with points onto the integers

$$\left\{ 0, 1, \ldots, \prod_{i=1}^{n} N_i - 1 \right\}, \tag{4.4}$$

such that each box has its unique integer representation. This unique integer is what will be stored in the hash-table and after adding all points in the point cloud, each value of $F(p)$ can be inverted to obtain the indices from which the center of the axis-aligned grid box can be conveniently computed according to

$$c_i = b_i + \left(\text{ind}_i + \frac{1}{2} \right) \frac{2}{\sqrt{n}}\epsilon . \tag{4.5}$$

The center will represent all points within this axis-aligned grid box in the new simplified point cloud $\bar{P}_\epsilon$. This is illustrated in Figure 4.1 where the original point cloud in 2D can be seen to the left and the resulting simplified point cloud obtained from using the grid-based partitioning simplification can be seen to the right.

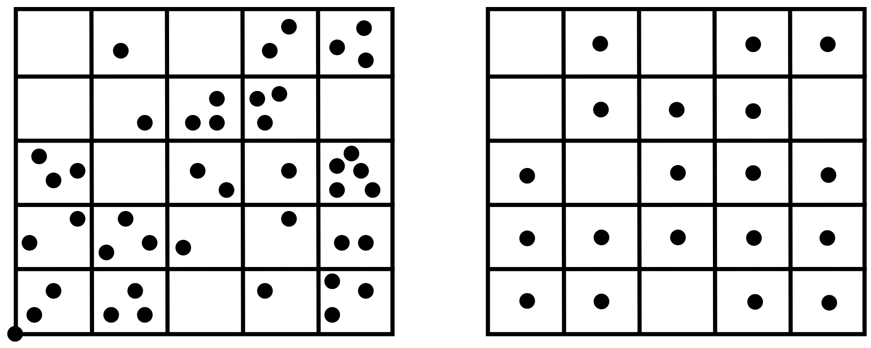

Figure 4.1: *(Left) Original 2D point cloud. (Right) Simplification of the point cloud after using the grid-based partitioning.*

This shows that points falling inside the same axis-aligned grid box may be represented by only the midpoint in the new simplified point cloud. It is possible that $|\bar{P}_\epsilon| = |P|$ if only one point falls within each axis-aligned grid box, which could happen if the point cloud is sparse or if ϵ is very small. The complete procedure of the grid-based partitioning can be seen in Algorithm 4.

Algorithm 4 Map disjoint subsets of data to grid boxes

```
Data: Point cloud $P$
Create an empty hash-table $H$
for each point $p \in P$ do
    if $F(p) \notin H$ then
        Add $F(p)$ to $H$
    end if
end for
Create $\bar{P}_\epsilon$ from $F(p)$
```

It should be noted that this algorithm relies on the existence of an out-of-core hash-table. The use of such a data structure can be avoided by modifying Algorithm 1 where the AABB of P is extended, as mentioned earlier in this chapter, and then divided so that the two new AABBs both have side lengths divisible by $\frac{2}{\sqrt{n}}\epsilon$. When each disjoint subset is small enough to run the grid-based partitioning in-core, $\bar{P}_\epsilon$ can be created. If the AABBs are divided in this way it can be guaranteed that $\bar{Q}_{\epsilon,i} \cap \bar{Q}_{\epsilon,j} = \emptyset, i \neq j$, so that no points from different subsets can be mapped to the same point in the simplified point cloud.

4.2 The k-means method

The grid-based partitioning described in the previous section has the advantage of having linear time-complexity in the size of the point cloud, but it may give a non-optimal partitioning for at least two reasons:

1. A box of diagonal 2ϵ can be contained in a sphere of radius ϵ.
2. The position of the axis-aligned grid boxes is chosen based only on the AABB of P and no other properties of the point cloud

The first point addresses the fact that an axis-aligned grid box of diagonal 2ϵ is not the object of maximal volume that could have been used to simplify the point cloud. In fact, Equation 4.1 will hold also in the case of representing all points falling within a sphere with radius ϵ by the center of the sphere. These spheres would have to be placed depending on the distribution of the point cloud since the original AABB cannot be partitioned into disjoint spheres.

This addresses the second problem of the grid-based partitioning, which regards the fact that the axis-aligned grid boxes are placed only depending on the AABB of P and not on the structure of the point cloud making it possible that significantly fewer axis-aligned grid boxes can be used if they are placed adaptively.

The k-means method was described in Section 2.3 and can be used to divide each subset in the same way as in Chapter 2. For each new subset it would have to be investigated whether the radius of the smallest enclosing sphere has a radius that is no more than ϵ. The problem of finding the smallest enclosing sphere of a set of points has been studied by Welzl [23], who constructed an algorithm that runs in expected linear time. If the radius is larger than ϵ, the k-means method can be used to create new clusters until each cluster can be replaced by the center of a sphere with radius of at most ϵ.

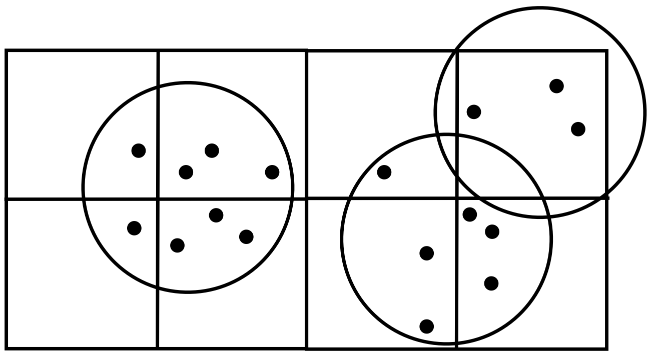

Figure 4.2: *Grid based method compared to the k-means method in 2D, illustrating that the k-means method may be able to create a simplified point cloud with fewer points.*

4.3 Maximum error in the computed shortest distance

This section will show that Equation 4.1 implies the second condition in the definition of $\bar{P}_\epsilon$. It is not necessary to consider the use of the convex hulls since the approximated distances are lower approximations of the true distances, therefore Algorithm 3 will always compute the exact shortest distance. In order for the point closest to the object to never be neglected, it suffices to consider only the error induced from simplifying the point cloud. This process is trivial if S consists of only one point and follows directly from the triangle inequality. The following theorem shows that the error in the shortest distance is always bounded by ϵ for a simplification scheme satisfying 4.1 for a general S.

Theorem 4.1. *Let $p \in P$ be the point from the original point cloud that is closest to S and $s = \arg\min_{x \in S} d(x,P)$. Furthermore, let $\bar{p} \in \bar{P}_\epsilon$ be the point in the simplified point cloud that is closest to S and $t = \arg\min_{x \in S} d(x,\bar{P}_\epsilon)$. Assume a simplification scheme such that condition 4.1 holds. Then it follows that*

$$|d(p,s) - d(\bar{p},t)| \leq \epsilon \tag{4.6}$$

Proof. By the definition of s and t it must hold that

$$d(p,s) \leq d(p,t) \tag{4.7}$$

$$d(\bar{p},t) \leq d(\bar{p},s). \tag{4.8}$$

From the triangle inequality it follows that

$$|d(p,s) - d(\bar{p},s)| \leq \epsilon \tag{4.9}$$

$$|d(p,t) - d(\bar{p},t)| \leq \epsilon, \tag{4.10}$$

since $d(p,\bar{p}) = d(\bar{p},p) \leq \epsilon$. To show the inequality in (4.6) assume that $d(p,s) \geq d(\bar{p},t)$ in which case

$$|d(p,s) - d(\bar{p},t)| \overset{(4.7)}{\leq} |d(p,t) - d(\bar{p},t)| \overset{(4.10)}{\leq} \epsilon. \tag{4.11}$$

and for $d(p,s) \leq d(\bar{p},t)$,

$$|d(p,s) - d(\bar{p},t)| \overset{(4.8)}{\leq} |d(\bar{p},s) - d(p,s)| \overset{(4.9)}{\leq} \epsilon, \tag{4.12}$$

which proves the claim. □

4.4 Build times

This section wishes to analyze the build times for each of the components from Chapter 2, Chapter 3, and this chapter. Algorithm 1 aims to divide the point cloud into subsets of at most T points, by splitting AABBs, and has time-complexity

$$O\left(|P| \log\left(\left\lceil \frac{|P|}{T} \right\rceil\right)\right) \tag{4.13}$$

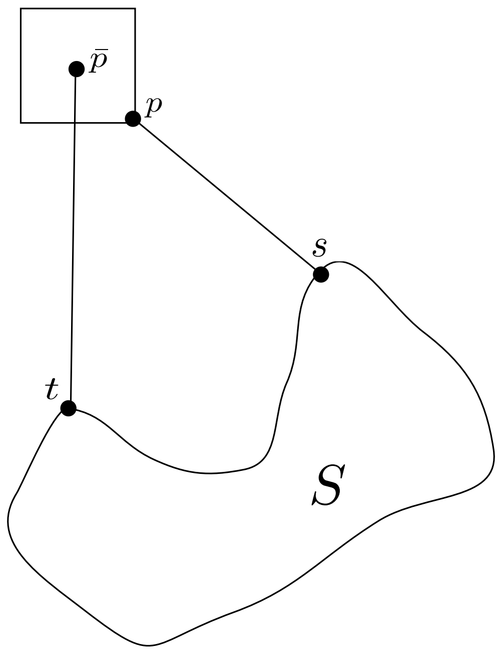

Figure 4.3: *Illustration of the maximal error of the method in the case of a grid box with diagonal* 2ϵ.

which is close to linear when T is large. This holds under the assumption that each AABB is divided into two new AABBs with approximately the same number of points. When T is small the time-complexity will instead be close to $O(|P| \log |P|)$. If k-means is used instead, the time-complexity will be much larger. Solving the exact k-means problem for a set of n points, i.e. finding the globally optimal centroids, has time-complexity $O(n^{dk+1} \log n)$, where d is the number of dimensions [24]. Assume instead that k-means++ is used and that the clustering phase in Algorithm 2 is aborted after I iterations, even if the clusters are still changing. The initializing phase of k-means++ has time-complexity $O(k^2 n)$ and each iteration of k-means will have time-complexity $O(kn)$. If k-means runs for at most I iterations, the resulting time-complexity will be $O(k(k+I)n)$. Assuming that all new clusters have roughly the same number of points, the resulting time-complexity will thus be

$$O\left(\frac{k(k+I)}{\log k}|P| \log\left(\left\lceil\frac{|P|}{T}\right\rceil\right)\right). \tag{4.14}$$

The factor $\log\left(\left\lceil\frac{|P|}{T}\right\rceil\right) / \log k$ is the expected number of times k-means needs to run to create clusters with less than T points, under the assumption that the clusters have roughly the same size. Since the swapping of elements when splitting AABBs is significantly faster than computing the distances and assigning points to clusters in the k-means method, the k-means method is expected to be much more time-consuming.

Building the convex hull for a set of points with approximately T points can be done in $O(T \log T)$, [22], so that all convex hulls can be built in $O(|P| \log T)$ time since there are approximately $\left\lceil \frac{|P|}{T} \right\rceil$ convex hulls that have to be built. In case the point cloud is to be simplified, the grid-based partitioning with a well-performing hash-table structure will be both linear in the size of the point cloud and in memory consumption since only one pass per point is needed after the subsets have been created. If the k-means method would have been used instead, together with Welzl's algorithm, the running time would be increased significantly, becoming

$$O(k(k+I)n + n) = O(k(k+I)n). \tag{4.15}$$

Note that this is the expected runtime since Welzl's algorithm has expected linear time-complexity. Assuming, just as before, that k-means assigns points into clusters with about the same number of points and, furthermore, that each sphere of radius ϵ will contain, on average, m points the time-complexity will be given by

$$O\left(\frac{k(k+I)}{\log k}|P| \log\left(\left\lceil \frac{|P|}{m} \right\rceil\right)\right). \tag{4.16}$$

In the worst case, when $m = 1$, the time-complexity will be $O(|P| \log |P|)$, compared to $O(|P|)$ for the grid-based partitioning, although the constant will be significantly larger for k-means and Welzl's algorithm potentially making it orders of magnitudes slower in practice.

4.5 Overview

In order to summarize all the steps in the pre-processing procedure a complete flow chart can be seen in Figure 4.4. This flowchart shows the major steps describing the pre-processing phase of the point cloud for fast-query together with the distance computation. More information regarding how the distance computation is carried out will be presented in Chapter 5.

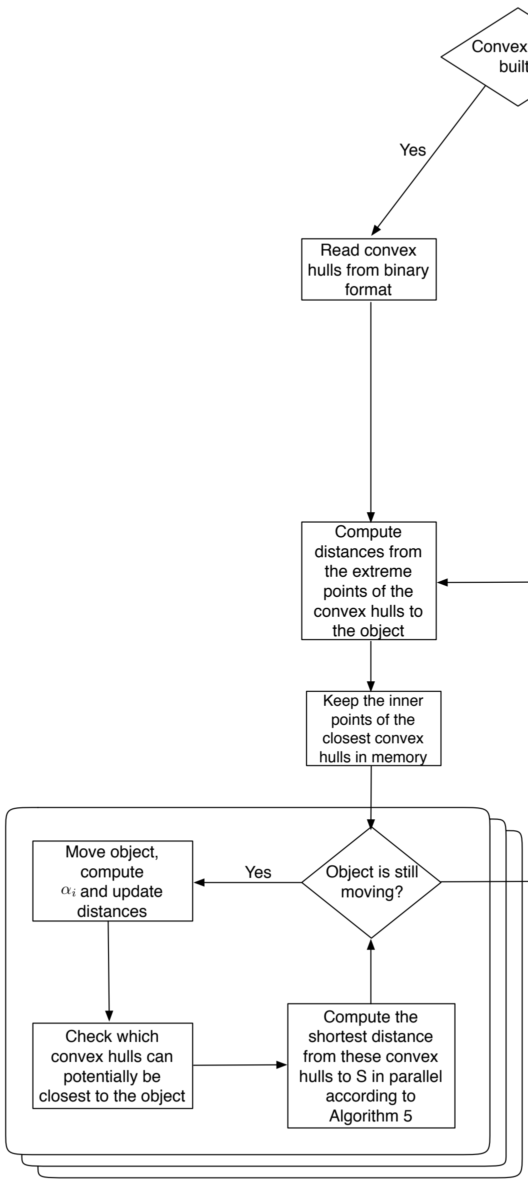
Convex hulls built?
Yes
No
Read point cloud into out-of-core vector
Read convex hulls from binary format
Point cloud so large that it must be simplified?
Yes
Simplify the point cloud depending on the user-tolerance ε
No
Divide the point cloud into disjoint subsets of size at most T
Compute distances from the extreme points of the convex hulls to the object
Save the convex hulls and values of r_{max} to a binary format
Build convex hulls
Keep the inner points of the closest convex hulls in memory
Move object, compute α_i and update distances
Yes
Object is still moving?
No
Output shortest distances
Check which convex hulls can potentially be closest to the object
Compute the shortest distance from these convex hulls to S in parallel according to Algorithm 5

Figure 4.4: *A flowchart representation of the algorithm developed in this thesis in order to pre-process a massive point cloud for fast-query out-of-core. The loop over the interesting convex hulls in the iteration block is preferably carried out in parallel as described in Section 5.4.*

5

Implementation

This chapter describes the implementation details with the main focus on how to work with a massive point cloud with the help of an out-of-core vector data structure. The choice of out-of-core implementation for this vector data structure is described in Section 5.1. The choice of algorithm for building the convex hulls is described in Section 5.2. In Section 5.3 it is described how to use bounding volume hierarchies and swept volumes to be able to rapidly compute the distance from two geometric models. This is what will be used for the convex hulls to be able to efficiently compute the distance from the object moved through the point cloud and the extreme points or the points that are not extreme points of the convex hulls.

5.1 STXXL

The out-of-core algorithms are implemented with the help of the STXXL library [25], which is an out-of-core implementation of the C++ STL (standard template library) data structures such as vector, stack, and queue. STXXL was designed, amongst other uses, to support multiple disks, overlapping of I/O and computations, prevention of overhead and pipelining. STXXL keeps a local copy of the data on the disks and loads elements that are requested into the cached memory of STXXL. STXXL was also designed to support STL-compatible interfaces in order to shorten the development time [25].

One STXXL vector is what will be used to hold the original point cloud throughout the pre-processing steps. A point cloud consisting of ten billion points with 3D coordinates, all of which are assumed to be floats, will need a total of 120 GB. Here a float is assumed to be 4 bytes according to [26]. Such a point cloud will hardly fit in-core on any computer whereas an STXXL vector could store the point cloud out-of-core using only 64 MB of internal memory (with standard configurations). STXXL can be used with even more RAM which will allow keeping more elements in RAM at the same time and therefore avoiding reading as many elements from the disk. The algorithms will always store the point cloud in an STXXL vector, thereby making the largest possible point cloud stored dependent only on the amount of disk drive space available.

There are however some limitations to the STXXL data structures, one being that it is not allowed to use pointers or references to elements. The reason for this is that the reference to an element is only valid as long as it is in the cached memory of STXXL. When new elements are read from the disk, old elements will have to be evicted from the cached memory and written to the disk, invalidating their references. STXXL provides memory iterators that are never invalidated and can be used to iterate through all points, such as when computing the AABB of P.

When an element of an STXXL vector is requested from a constant vector, the element will not have to be written back to the disk after being evicted from the cached memory, avoiding unnecessary writing to the disk. The limitation of not being able to use references complicates the construction of parallel algorithms with STXXL data structures since tools like OpenMP will use references to access the shared memory.

Another problematic limitation is that the STXXL data structures can only hold elements that satisfy the criteria for being PODS (plain old data structure) since there is no obvious way to convert other elements to the internal binary format used by STXXL. This means that an STXXL vector cannot be used to hold instances of a class that does not satisfy the criteria for being a POD, which is not satisfied by most sophisticated classes.

STXXL allows different types of memory access implementations depending on the operating system used; for Windows the most common choices are *syscall*, *wincall*, *memory*, and *mmap*. Syscall uses the memory pages directly and is currently the fastest method, wincall uses the Windows API, memory keeps the data in RAM (for testing purposes), and mmap uses memory map system calls. Syscall will always be used in this thesis since it is the fastest way of accessing the data.

In order to test the performance of STXXL a few simple test cases have to be constructed. The first test is to compute the sum of a vector of 1 billion floats. The second test is to perform 1 billion swaps in a vector of 1 billion floats in order to test how the time varies when both reading and writing is of necessity. The final test will be to do push backs of 1 billion floats to an STXXL vector. The tests are carried out using the STL implementations and will be compared to using one SSD (solid state drive), one HDD (hard disk drive) and one SSD together with an HDD. The results can be seen in Table 5.1.

From Table 5.1 it can be seen the main memory is in all cases much faster than STXXL, which is to be expected since the access time is several orders of magnitude faster for RAM than for disk drives. When swaps are performed, the speed of using two disks is almost the sum of the individual speeds, showing that the parallel disks seems to be working well. The same almost applies for the push back operations where some speed is gained from using two disks, even if the SSD is dominating the HDD in write speed. On the other hand, not much speed-up was gained from using two disks while only accessing elements when computing the sums of the vector. These simple tests verify that even if STXXL is fast, it is still about one order of magnitude slower than RAM.

5.2 Building the convex hulls

The construction of the convex hulls is a time-consuming part of the pre-processing, with Chan's algorithm having the best theoretical run time of $O(n \log h)$, where n is the number of points and h is the number of extreme points [27]. Another available algorithm is the Quickhull algorithm [22] with run time $O(n \log n)$, which is the standard approach for computing convex hulls. Quickhull has been implemented in the geometrical software package CGAL [28] and in the library Qhull [29]. Joesph O'Rourke provides a slow but simple implementation of the incremental algorithm that is described in [30]. The Qhull implementation is preferred for the work in this thesis since it is a smaller library than CGAL and also because O'Rourke's implementation is about a factor of 100 slower than Qhull. Another problem with O'Rourke's algorithm is that it is designed for data with integer precision in order to check for coplanarity or colinearity while Qhull supports coordinates in floating precision and solves a potential precision problem by merging faces.

After the convex hulls have been built, they are separated into two STXXL vectors, one with points that are not extreme points and one with extreme points. A convex hull structure keeps track of which indices belong to each convex hull and what the value of $r_{\max}$ is. This data is then saved in a binary format in order to allow skipping the pre-processing the next time this particular point cloud is considered. The reason for choosing a binary format over simply saving the points to a textfile is the fact that an STL vector is contiguous in memory, making it possible to quickly insert the elements in memory again. Pieces of an STXXL vector are converted to an STL vector from which the piece is saved into a binary format because of the previously mentioned problem regarding references to elements in an STXXL vector.

Test case	RAM	SSD	HDD	SSD & HDD
Sum	2.0	10.9	10.6	9.5
Swaps	2.6	38.2	59.0	23.6
push backs	2.0	17.5	53.0	14.7

Table 5.1: Run time in seconds when operating on a point cloud with 1 billion points. Each point is represented by three floats making the memory consumption for the point cloud 12 GB. STXXL was used with 64 MB of RAM.

5.3 PQP

In order to use the algorithms from Chapter 3, a way to find the shortest distance between a subset of the point cloud and the triangulated object is needed. The Proximity Query Package (PQP) [31] was designed for collision detection, distance computation, and tolerance verification between a pair of geometric models. When necessary, this is what will be used to compute the distance from either the extreme points and/or points that are not extreme points of a convex hull to S. PQP uses swept sphere volumes to create a bounding volume hierarchy that can be used to efficiently compute the shortest distance by traversing the tree of bounding volumes. By storing indices of the closest pair of points/triangles from the last distance computation, an upper bound of $d_{\min}$ can immediately be obtained, giving PQP a good starting point. This makes it possible to rule out a large amount of points when traversing the boundary volume hierarchy.

The memory requirement to build a PQP model is extensive since building a PQP model with 100 million points uses 16GB of RAM. This memory limitation is not a problem for the proposed approach since the convex hulls will have one PQP model for its extreme points and one for its points that are not extreme points. It will be seen in Chapter 6 that $T = 100{,}000$, where T was defined as the maximal number of points in a subset, works well for a point cloud with 1 billion points for which the PQP models can be constructed without using much RAM. A constructed PQP model for a set of points will need roughly 3 times the memory of the point cloud.

PQP is fast and it could be questioned as to why not just use one huge PQP model that could run out-of-core? The reason for this is that the limitations of STXXL makes the construction of the PQP model problematic, and this implementation is not of interest to modify because the changes would be contradictory to the programming paradigm in C++. Another reason that can be seen from Table 5.1 is that RAM accesses are significantly faster, in which case the PQP models that are known to be closest to the object should preferably be in RAM, avoiding reading the same data from the hard disk drive to the cached memory for multiple iterations in a row.

Instead of running one big PQP model out-of-core, several PQP models will be used and run out-of-core. This is where the convex hulls are exploited since they make it possible to rule out many of the PQP models that are too far away from the object to be of interest. This situation changes when the object moves in which case some PQP models will have to be loaded while others can be removed from RAM in case all of the PQP models cannot fit at once.

PQP was modified to allow writing of the interior data structure into a binary format since STXXL cannot be used to hold the PQP models due to the fact that the PQP data structure does not satisfy the criteria for being a PODS. Another reason for this modification is that the distance can be computed in parallel, which would not have been the case if an STXXL vector would have been used to hold the PQP models. The PQP models for the extreme points of each convex hull are never removed from RAM since the number of extreme points is expected to be small, compared to the number of points that are not extreme points, so that they can fit in RAM at all times. Another reason for always keeping the PQP models of the extreme points in memory is that they will always be considered before the PQP model for the points that are not extreme points. One advantage with not using STXXL is that when a PQP model for points that are not extreme points has to be read, PQP models that are far away from S can be removed since they are not expected to be used again. In the case of using STXXL it is not clear what will be removed in order to free up the space for the reading, which means that a PQP model that is close to S could be removed. A situation where the PQP models for the closest sets of points that are not extreme points are in RAM together with all PQP models for the extreme points is illustrated in Figure 5.1

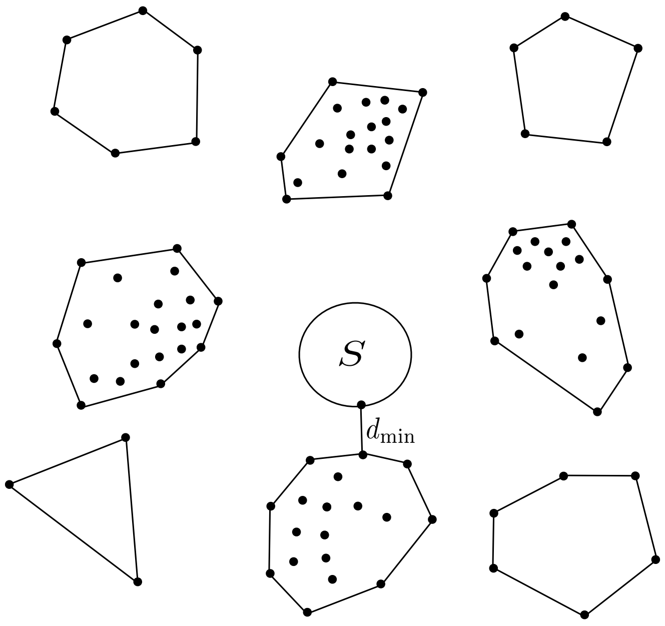

Figure 5.1: *The PQP models for the closest sets of points that are not extreme points are in RAM and the models that are further away exist only on the disk. When S moves the PQP models for points that are not extreme points of sets that come closer will have to be read and PQP models for points that are not extreme points that are then further away will then be removed.*

5.4 Shortest distance computation

It will in this section be described how to compute the shortest distance in parallel after the object has been moved and α_i has been computed. The for-loop inside the while-statement of Algorithm 3 will be replaced by one for-loop that sets $d_j \leftarrow d_j - \alpha_i$ and then saves all the convex hulls such that $d_j \leq d_{\min,i}$ to a list L, that will then contain all convex hulls that can contain the point closest to S. A parallel for-loop over the elements of L is created, with each thread having their local copy of the shortest distance found so far, which is initialized to $d_{\min,i}$. After thread number k has processed a convex hull and updated its local value of the shortest distance, δ_k, the global value of $d_{\min,i}$ is updated if $\delta_k < d_{\min,i}$. If not, δ_k is updated to equal the global value of $d_{\min,i}$, which is the shortest distance found so far. By updating δ_k, each thread can make use of the information collected by the other threads and this can make it possible to rule out additional convex hulls.

The reading of PQP models takes place inside the same loop and can also be done in parallel as long as enough PQP models are removed ahead of the reading so that enough memory can be guaranteed. When a PQP model has to be removed, the one with the largest value of d_j (and is in RAM) will be removed, so that a model that is being removed is unlikely to be considered again. The complete parallel algorithm that replaces the for loop within the while loop in Algorithm 3 is given in Algorithm 5.

Algorithm 5 Algorithm for computing the shortest distance in parallel

```
Data: Convex hulls of subsets of Point cloud $P$, Object $S$, iteration $i$
Create empty list $L$
for each convex hull number $j$ do
    $d_j \leftarrow d_j - \alpha_i$
    if $d_j \leq d_{\min,i}$ then
        Add convex hull number $j$ to $L$
    end if
end for

Start of parallel region
Initialize $\delta_k = d_{\min,i}$ for thread number $k$
for each convex hull number $j$ in $L$ (in parallel) do
    Update $d_j$ and set $\delta_k = \min(d_j,\delta_k)$
    if $d_j - r_{\max,j} \leq \delta_k$ then
        if PQP model for the points that are not extreme points is not in RAM then
            Start of barrier
            Remove PQP models in order to free memory for the reading
            End of barrier
            Read the necessary PQP model
        end if
        Compute the distance from $S$ to the points that are not extreme points and
        update $\delta_k$ if any point is closer to $S$ than $\delta_k$
    end if
    Start of barrier
    if $\delta_k < d_{\min,i}$ then
        $d_{\min,i} = \delta_k$
    else
        $\delta_k = d_{\min,i}$
    end if
    End of barrier
end for
End of parallel region
```

6

Simulations

This chapter aims to test the results from the previous chapters by considering a real-world point cloud. Section 6.1 will describe the point cloud that will be considered and how it can be up-sampled in order to force the shortest distance computations to run out-of-core. Section 6.2 will introduce some alternative approaches to using Theorem 3.5 for fast approximate distance computations. In Section 6.3 the simulation setup will be presented and the results will later be analyzed in Section 6.4. The simulations were carried out on a desktop computer with an Intel Core i7 processor, 32 GB RAM, and a 250 GB SSD.

6.1 Test geometry

The point cloud that will be considered is a factory originally consisting of about 40 million points. A picture of the point cloud can be seen in Figure 6.1 and the entrance, where the object moved through the point cloud is placed can be seen in Figure 6.2. The red line shows the path along which the object will be moved through the point cloud. This entrance is located in the upper right corner of Figure 6.1 and the object will move through the tunnel until it reaches the part of the point cloud that is in the lower left corner of Figure 6.1.

The original point cloud is obviously not of interest because it contains so few points, with the main focus of this thesis being massive point clouds having billions of points. For this reason, the point cloud was up-sampled to contain a larger number of points by simulating $k - 1$ new points around each original point in order to generate a point cloud with $40k$ million points.

The object that is being moved through the point cloud is a holding mechanism for a car and its triangulation consists of 80,667 points and 78,403 triangles; this can be seen in Figure 6.3. The position of the object along the existing path is given at discrete points in time and the displacements of the object vary in size. The variation in the

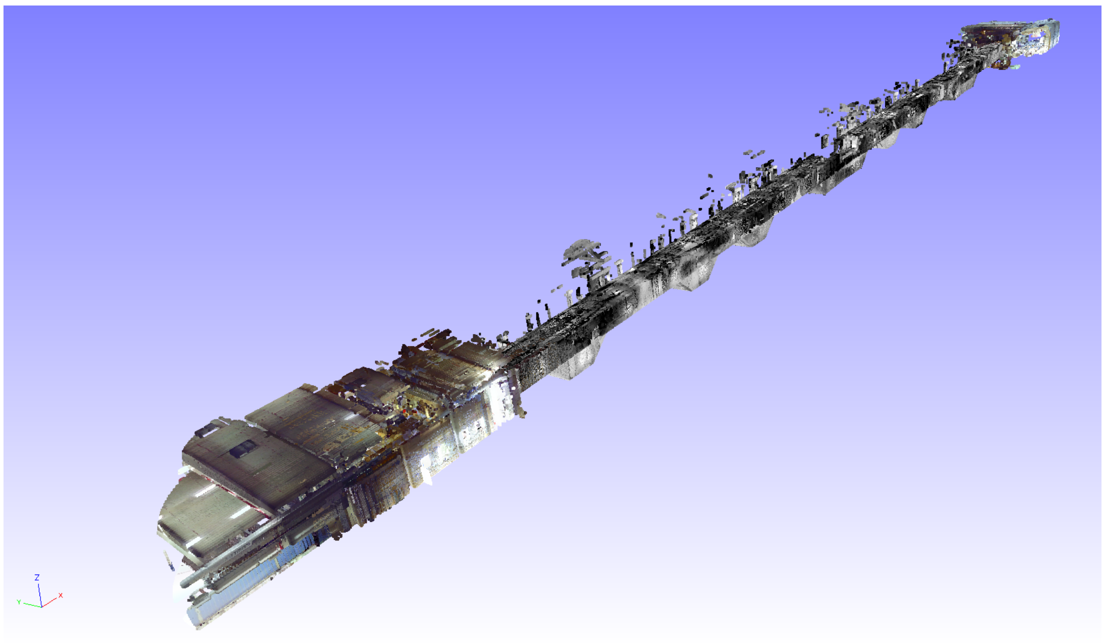

Figure 6.1: *An overview of the original point cloud consisting of about 40 million points.*

displacements will directly affect the computed values of α_i.

The value of α_i has to be computed quickly since the aim is to obtain close to real-time computations of the shortest distance, which is taken to be 24 fps, [32]. To put this in perspective, the eye can only process 10-12 distinct frames per second, [33]. The idea of α_i is to provide a quick and reliable test without computing any distances from the subset to S for whether any points in a subset can be of interest by using the information from the previous iteration. The choice here was to therefore compute the original bounding box of the object and transform its 8 corners at each iteration and then compute the furthest a corner point has moved. This will give an upper approximation of α_i that can be computed quickly since it is independent of the number of vertices and triangles of the object.

Another approach that would yield a more precise value of α_i would be to compute the convex hull of the object and then see how far the vertices have been moved, but this can be considerably slower if the fraction of extreme points in this convex hull is large. The total number of steps in the already generated path is 707 and the total time taken to carry out the steps is 229 seconds in simulation time.

6.2 Alternative approaches

In order to test the results from Chapter 3, it will be investigated how the algorithm compares to some similar ideas on how to rule out subsets of P that are far away from

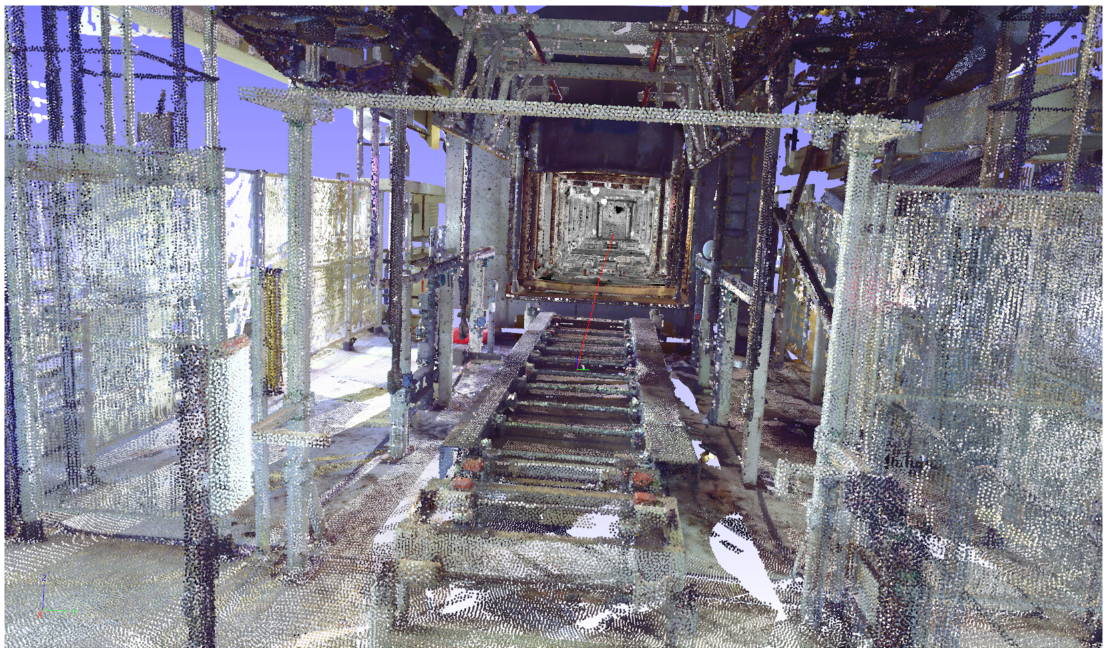

Figure 6.2: *A view of the entrance of the tunnel through where the object will be moved. The red line shows the path of the object.*

the object. The five approaches that will be considered are:

1. Convex hulls
2. Triangulated convex hulls
3. The hybrid method
4. Multiple PQP models
5. A Single PQP model

The first approach uses the lower bound of d_j that is based on α_i to rule out some of the subsets, exactly as is described in Algorithm 3. Each subset will have one PQP model for its extreme point and one for the points that are not extreme points, with all PQP models for the extreme points residing in RAM at all times.

Triangulated convex hulls will be used to compute the exact distance from the convex hull to the object. This will be much more expensive to evaluate since computing the distance between two triangulations is more time-consuming compared to computing the distance from a set of points to a triangulation. It is, on average, four times faster to compute the distance between a point and a triangle compared to computing the distance between two triangles. If a convex hull contains h extreme points and all faces are triangles, then there are $2h - 4$ triangular faces, [34]. Hence there are about twice

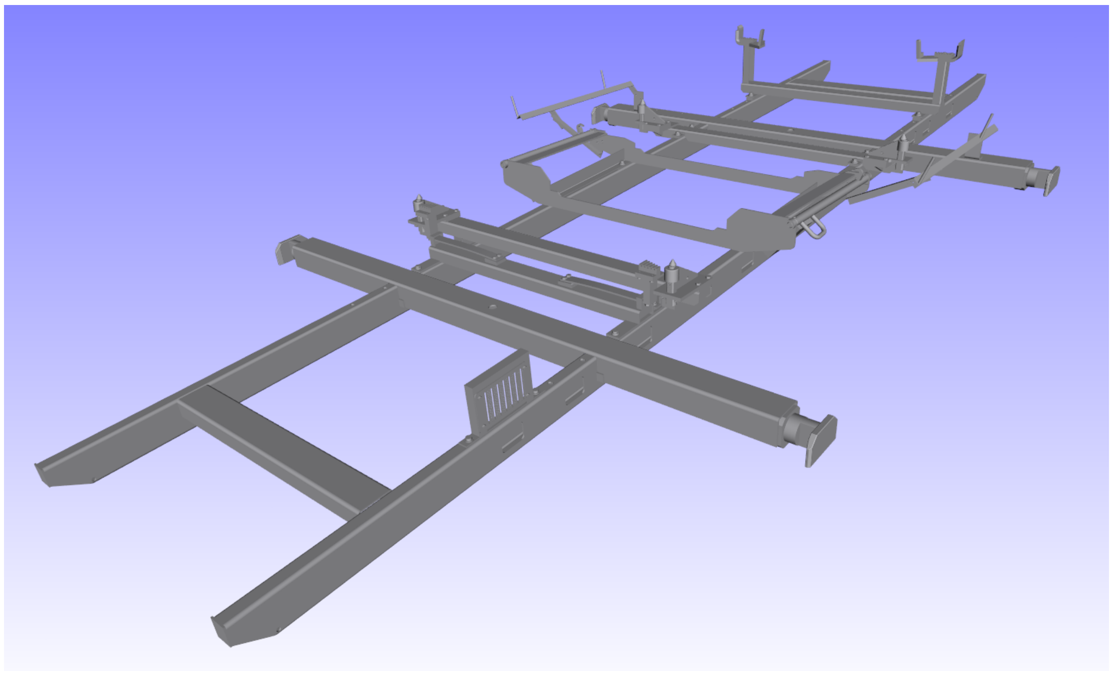

Figure 6.3: *A visualization of the object, a car holding mechanism, that will be moved through the point cloud. This triangulated model consists of 80,667 points and 78,403 triangles.*

as many faces as there are extreme points, making it about 8 times slower to compute the exact distance to a convex hull, compared to using the extreme points. On the other hand, the triangulated convex hulls have the strength of not having to do as many reads of PQP models since the exact distance is computed from $\mathcal{CH}(Q)$ to S. Each subset will have one PQP model for the triangulation of the convex hull and one for all the points in the subset.

The hybrid method aims to use the strengths of both the previously mentioned approaches by utilizing the fact that reading a new PQP model is generally very time consuming. Each subset will have a PQP model for the extreme points, triangulation and points that are not extreme points, and is used identically to the first approach unless Theorem 3.5 fails to rule out a set of points that are not extreme points that is not in currently RAM. In such a case, the exact distance is computed from the triangulation to the object as an attempt to avoid reading the PQP model of the points that are not extreme points.

By **multiple PQP models** it is meant that instead of building convex hulls for the subsets generated during the pre-processing as described in Chapter 2, PQP models are built for each subset. Only the lower bound based on α_i is used to avoid considering certain subsets when it is known that they must be further away than the subset with the closest point. As soon as a set comes too close, the distance is computed from its

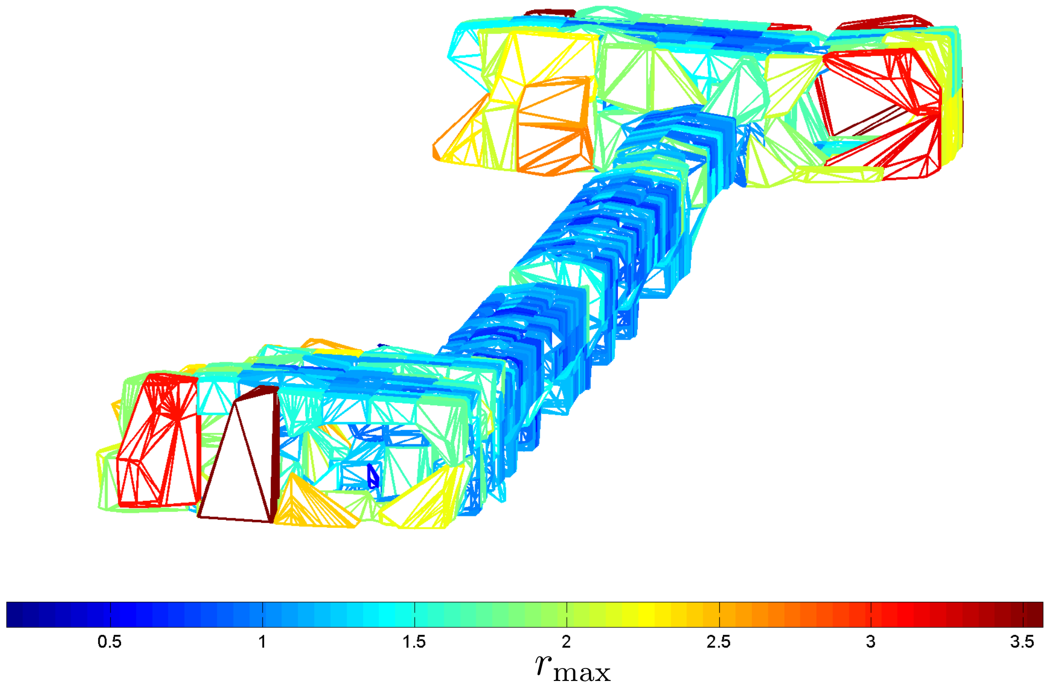

Figure 6.4: *The point cloud after diving it into subsets it and building convex hulls. The color indicates the value of* $r_{\max}$ *and is given in meters.*

PQP model to the object. This approach will have to perform many unnecessary reads of PQP models that turn out not to be close; it is mainly a naive way of just splitting one large PQP model into multiple subsets.

Using a **single PQP model** requires no additional description than what was given in Section 5.3. It will be seen that PQP will be much faster than the other approaches for small point clouds but beyond that, a single PQP model cannot be used anymore due to the extensive memory requirement to build the model.

An additional possibility that was discarded was using an AABB of each subset by either computing the distance between a triangulation of the AABB or from the corner points and also applying the inequality from Chapter 3. The reason to not consider the AABBs is that only a small fraction of the points are expected to be extreme points, so there are not many more points to consider for the convex hulls over the AABB and will therefore not be much slower, under the assumption that PQP is logarithmic in the size of the number of points. Furthermore, it is expected that using convex hulls will produce better bounds than an AABB, as was argued in Section 3.6.

6.3 Simulation setup

This section will present the setup used in the simulations meant to compare the five approaches from the previous section. No point cloud simplification was used, which means that ϵ was set to zero. The point cloud was organized by the means of Algorithm 1 since it is significantly faster than using the k-means method. Five different values of $|P|$ were considered, starting with the original point cloud with 40 million points. This case was possible to run in-core for all five approaches and was evaluated for four different values of T. The point cloud was then up-sampled to contain five times as many points, yielding a point cloud with 200 million points. This is the maximum number of points for which it is possible to build a PQP model on a machine with 32 GB memory. After this, all test cases except, a single PQP model, were evaluated on a point cloud with 600 million points, which was possible to run in-core with 32 GB of RAM. The hybrid method was not used for any of these three tests that run in-core since, in this case, it is equivalent to using the convex hulls.

After these three tests, that were carried out in-core, two point clouds with 1 billion and 3 billion points were generated. For the point cloud with 1 billion points, the memory for PQP was restricted to 8 GB so that only about 25% of the PQP models for the points that are not extreme points could be in RAM at once. In the case of 3 billion points, the memory was restricted to 16 GB in which case only about 15% of the PQP models for the points that are not extreme points could be in RAM simultaneously. The results from the simulations can be seen in Table 6.1.

In order to illustrate how smaller values of T will average out the time for the reads/removals of PQP models over the iterations, the time to compute the shortest distance for was plotted against the number of reads. The results from this test can be seen in Figure 6.5. Table 6.2 shows how many reads and tests of different PQP models had to be carried out for the first four approaches in the case of the point cloud with 3 billion points and 16 GB of RAM.

The last test was to investigate how the grid-based partitioning and the k-means method compared when it comes to point cloud simplification. Even though other criteria besides the number of points in the simplified point cloud could have been considered, it was decided to use this as the measure of efficiency. The original point cloud introduced in Section 6.1 was considered since an up-sampled point cloud would have made it hard to choose realistic values of ϵ in addition to extending the runtime. The results from using the grid-based partitioning and k-means for $k = 2$ and $k = 3$, for different values of ϵ, can be seen in Table 6.3.

6.4 Simulation results and discussion

This section aims to analyze the results from the previous section. Starting with Table 6.1, it can be seen for the two smaller point clouds that a single PQP model is clearly the fastest. This result is expected since PQP was optimized to work on point clouds of this size. For smaller point clouds, there is of no interest to replace PQP unless the

$\lvert P \rvert$ (millions)	Memory (GB)	T (thousands)	Convex hulls	Triangulated hulls	Hybrid method	Multiple PQP	Single PQP
40	32	10	18.7	136.5	-	25.1	**3.7**
40	32	100	7.2	23.9	-	8.8	**3.7**
40	32	500	7.8	14.9	-	8.2	**3.7**
40	32	1,000	7.9	11.2	-	7.9	**3.7**
200	32	100	14.0	111.6	-	48.2	**8.3**
200	32	500	9.1	29.4	-	23.2	**8.3**
200	32	1,000	10.8	21.6	-	22.1	**8.3**
200	32	5,000	12.6	16.0	-	22.5	**8.3**
600	32	500	11.9	61.7	-	55.7	-
600	32	1,000	**11.3**	37.8	-	42.3	-
600	32	5,000	15.7	22.1	-	33.8	-
600	32	10,000	18.4	22.7	-	31.7	-
1,000	8	50	66.3	568.7	71.3	718.7	-
1,000	8	100	**47.5**	373.1	48.6	571.3	-
1,000	8	500	53.2	103.1	52.0	573.2	-
1,000	8	1,000	67.3	76.5	56.4	491.3	-
1,000	8	5,000	91.7	96.1	88.8	782.9	-
3,000	16	100	103.5	735.2	100.2	2453.6	-
3,000	16	500	93.0	288.8	**67.3**	2138.2	-
3,000	16	1,000	129.2	167.7	87.9	2543.1	-
3,000	16	10,000	258.2	235.5	225.1	3049.2	-
3,000	16	50,000	303.1	286.3	279.9	3875.0	-

Table 6.1: Run times, in seconds, for computing the shortest distance for all 707 steps in the path of the object. The best time for each size of $|P|$ is written in bold font. Memory indicates how the amount of RAM was constrained to force the algorithms to run out-of-core. For the point clouds with 40, 200 and 600 million points, all algorithms ran in-core. Hybrid was not used in these cases, since it is identical to the convex hulls when running in-core. PQP was not used for point clouds with more than 200 million points because of the memory problems associated with building a model with that many points. The values of T were chosen to capture the variation in time for all of the scenarios and were chosen adaptively from a larger set of simulations. The loop over the convex hulls at each iteration was always carried out in parallel as described in Algorithm 5.

amount of RAM available is very small; the two small point clouds were only considered to illustrate how fast PQP is. It is clear that the triangulated convex hulls are better

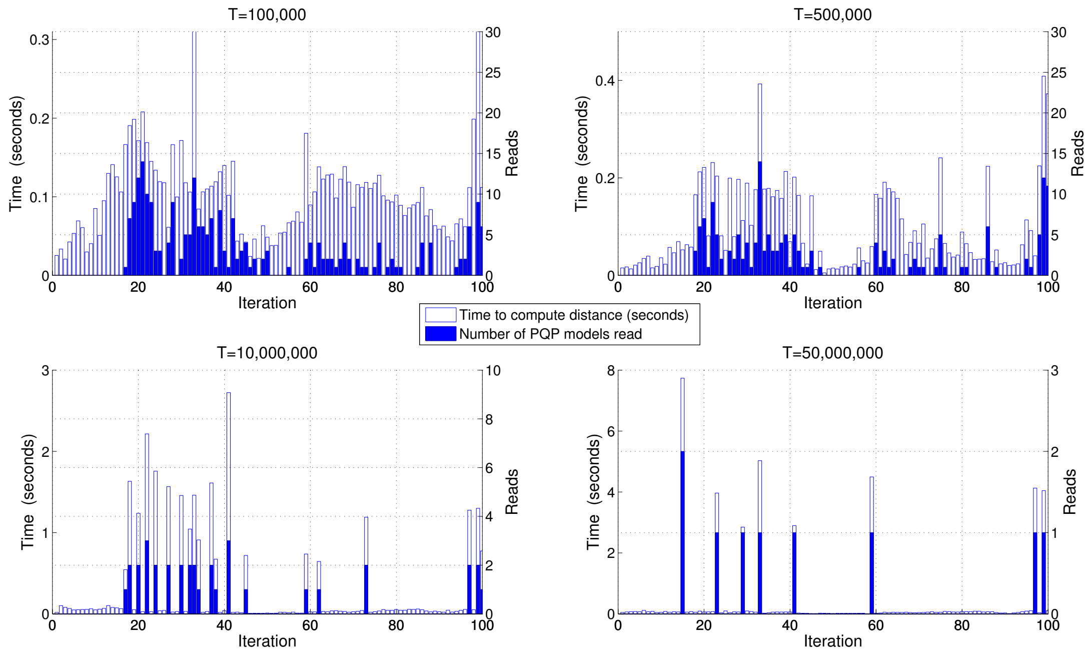

Figure 6.5: *A plot of the time to compute the shortest distance for the hybrid method compared to how many PQP models that had to be read for the first 100 iterations. The point cloud was up-sampled to 3 billion points and the RAM was constrained to 16 GB. Note that the scales are different.*

when the subsets are larger since this will make the number of comparisons between triangles smaller. Both the triangulated convex hulls and multiple PQP models will be faster when the subsets are larger since, in this case, they converge towards a single PQP model, which is the fastest approach for small point clouds. For the convex hulls, it works better to have about 400-500 convex hulls, and for 200 million points the convex hulls compare well in runtime to a single PQP model.

When the point cloud has 600 million points and it is still possible to run the distance computations in-core, except for a single PQP model, it is clear that the method with the convex hulls is the fastest. Just as for the two smallest point clouds, the triangulated hulls and the multiple PQP models get better when the subsets are increased. This will change when the algorithms start running out-of-core in which case the reading of PQP models will become a bottleneck.

For the two largest point clouds, when only a fraction of PQP models can fit in RAM, it is clear that the hybrid method and the convex hulls both perform well. The convex hulls work better when T is small since the values of $r_{\max}$ are expected to be smaller in this case. On the other hand, when the convex hulls become too small, the fraction of extreme points will increase, making it problematic to fit the PQP models for the extreme points in memory. This will also make it very expensive to compute

$T = 50$ (millions)	Convex hulls	Triangulated hulls	Hybrid method	Multiple PQP
Extreme points	5,630	-	5,235	-
Triangulations	-	4,352	424	-
Not extreme points	3,490	2,125	2,748	4,294
Reads	64	59	59	1,010
$T = 10$ (millions)	Convex hulls	Triangulated hulls	Hybrid method	Multiple PQP
Extreme points	23,613	-	22,734	-
Triangulations	-	19,186	2,645	-
Not extreme points	10,074	4,883	6,961	19,008
Reads	307	255	253	4,431
$T = 1$ (millions)	Convex hulls	Triangulated hulls	Hybrid method	Multiple PQP
Extreme points	194,763	-	193,565	-
Triangulations	-	175,458	11,896	-
Not extreme points	24,187	6,416	12,629	175,274
Reads	1,729	886	926	44,399
$T = 0.5$ (millions)	Convex hulls	Triangulated hulls	Hybrid method	Multiple PQP
Extreme points	368,056	-	366,780	-
Triangulations	-	339,305	14,224	-
Not extreme points	26,209	6,621	12,773	339,161
Reads	2,545	1,168	1,142	87,421
$T = 0.1$ (millions)	Convex hulls	Triangulated hulls	Hybrid method	Multiple PQP
Extreme points	1,671,502	-	1,669,979	-
Triangulations	-	1,482,392	17,329	-
Not extreme points	26,519	5,932	10,783	1,456,329
Reads	5,108	1,598	1,557	432,901

Table 6.2: Number of distance computation in-between different PQP models and reading of PQP models in the case of $|P| = 3{,}000$ million points and 16GB memory

the distance from the triangulations to the object, which is why the triangulated convex hulls perform worse for small values of T.

It can be seen in Figure 6.5 that when using values of T that are large the criteria that it is not allowed to wait several seconds to load a large number of points will be violated. This requirement will force T to be small in order to average out the I/Os

ϵ	k-means, $k = 2$		k-means, $k = 3$		Grid-based partitioning	
(meters)	Points	%	Points	%	Points	%
0.005	33,902,556	79.4	36,441,171	85.4	40,226,840	94.3
0.01	19,467,172	45.6	23,186,798	54.3	34,432,143	80.6
0.02	8,547,284	20.0	10,041,080	23.5	15,772,929	37.0
0.03	4,634,686	10.8	5,409,136	12.7	7,549,912	17.7
0.05	1,977,789	4.6	2,315,186	5.4	2,913,543	6.8
0.1	556,833	1.3	674,305	1.6	788,407	1.8

Table 6.3: Number of points remaining and percentage of points remaining for each simplification method. The the original point cloud consisted of 42,675,250 points. The k-means method tried to minimize Equation 2.1 in L_2.

among the iterations, making only the convex hulls and the hybrid method suitable choices, since the triangulated hulls perform poorly when T is small. Especially in the case $T = 50$ millions it is clear that the algorithm stops for up to 8 seconds in order to read points. What follows are some very fast iterations until a new large model has to be imported again. Such a behaviour is exactly what should be avoided and therefore smaller values of T will have to be used.

It is clear from Table 6.2 that the multiple PQP models have to perform many unnecessary reads, which is mainly why this approach is so much slower. The triangulated convex hulls and the hybrid method compute the exact distance to the convex hull before reading PQP models, which is why they do the smallest number of reads. On the other hand, the triangulated convex hulls have to compute almost as many distances from the triangulated convex hulls to the object as the convex hulls have to compute the distances from the extreme points. This is where the hybrid method offers a good balance between the two since it only computes the exact distance in order to avoid reading a PQP model.

It is natural to ask the question, why the number of reads is not always the same for the triangulated convex hulls and for the hybrid method? The main reason is that the the distances are computed in parallel, so how $d_{\min,i}$ changes depends on the order in which the threads finish processing the convex hulls. Depending on how $d_{\min,i}$ changes, it can be possible to rule out a few extra sets of points that are not extreme points if a small value of $d_{\min,i}$ is found early.

Finally, Table 6.3 shows that the simplified point cloud can be constructed with fewer points when using the k-means method rather than the grid-based partitioning. Additionally, it is clear that more points can be removed when $k = 2$ compared to $k = 3$. The reason that $k = 2$ performs better than $k = 3$ is that for smaller sets of points that could possibly fit in two spheres of radius ϵ, dividing such a set into three new sets creates one unnecessary sphere. It should be mentioned again that using point cloud simplification with k-means is several orders of magnitudes slower than using the grid-based partitioning.

One additional advantage of using the k-means method is that the risk of the resulting subset being coplanar or collinear is much less since the points are not placed in a lattice as they are in the case of the grid-based partitioning. If a subset is coplanar or collinear, the convex hulls cannot be built anymore, although it would still be possible to build an embedded two-dimensional convex hull for a coplanar (but not collinear) set of points and then triangulate this convex hull. The value of r_i can then be computed for each such triangle and $r_{\max}$ can be taken as the largest r_i. On the other hand, since not that many convex hulls are expected to be either coplanar or collinear, the entire subset can be represented by a single PQP model.

7

Discussion and conclusion

A discussion regarding the results in this thesis is given next. It will be discussed in Section 7.1 how the theoretical framework from Chapter 3 performed in practice. The topic on how efficient the simplification schemes introduced in Chapter 4 can be, depending on ϵ, will also be addressed. By analyzing the results from Chapter 6, conclusions will be drawn regarding the efficiency of the five different approaches introduced in Section 6.2. Some concluding remarks will be given in Section 7.2 and some investigations for future work and improvements are stated in Section 7.3.

7.1 Discussion

It has been verified that using convex hulls as a basis to derive sharp criteria for neglecting points works well both in-core and out-of-core. Theorem 3.5 is not only appealing in theory but turns out to work well in the simulations carried out in Chapter 6. When used in combination with Theorem 3.7 that uses α_i to quickly find a lower bound of the distance to the subset, a large number of subsets can immediately be ruled out. In the case of a point cloud with 3 billion points and $T = 100{,}000$ only 4% of the subsets had to be considered at each iteration after the use of Theorem 3.7. This is a huge performance increase since so many subsets can be excluded immediately without computing any distances.

It can be questioned whether it is worth using the hybrid method or if the model of the convex hulls is the most appropriate approach for a commercial implementation? It would seem to be preferable to use the hybrid method, because it helps minimize reads of PQP models in cases where some of the convex hulls have large values of $r_{\max}$, which could be a possibility in the presence of outliers. In the case of using an HDD for reading PQP models, the price for I/O will increase making it even more critical to avoid reading PQP models.

The runtime is expected to increase when the size of the point cloud increases, especially when the memory is constrained. In the case of the point cloud with 1 billion points and 8 GB of memory, the size of all PQP models will be about 36 GB, with only 8 GB being in the memory during the first iteration. Reading all other PQP models once on the SSD used, with a theoretical speed of 450 MB/s, would then take 62 seconds, which is more than the fastest total runtime. This is because all PQP models were not used, but indicates that reading models is expected to take up much more time than actually computing distances. This is why the hybrid method works well since it minimizes the reading without having to consider as many triangulations as for the triangulated convex hulls.

It was seen for the both the hybrid method and for the convex hulls that a small value of T is necessary since this will both minimize the runtime and even out the reading of PQP models, so that the distance computation does not have to stop for several seconds in order to read a large set of points. On the other hand, more investigation is needed on the topic of choosing the value of T, which proved to be critical in order to minimize the runtime. From Table 6.1 it seems as though the optimal value of T increases when $|P|$ increases. The values in the table with the optimal values of T for the method of interest can be interpolated or extrapolated to the size of the point cloud considered.

The work of this thesis has now made it possible to conveniently work with point models with an order or magnitude of more points, making it possible to work with billions of points rather than the hundreds of millions of points that was the restriction for a single PQP model in-core.

7.2 Concluding Remarks

The main conclusions of this thesis are:

- Computing distances to a massive point cloud having billions of points is now possible on a computer with only a few GB of RAM.
- Convex hulls are efficient for ruling out subsets of points with the use of Theorem 3.5.
- By computing α_i, the largest displacement of any point in S, a large number of subsets can be ruled out without computing any distances by using Theorem 3.7.
- The reading of PQP models will be averaged out over the iterations if T is small, making each distance computation similar to the others in time.
- The hybrid method, which was designed to avoid reading PQP models, minimizes the runtime even further.
- The k-means method is more efficient at simplifying the point cloud than the grid-based partitioning, if the point cloud needs to be simplified, but is at least an order of magnitude slower.

7.3 Future work

Some modifications that are expected to improve the algorithms even further are:

- Taking the direction of motion of S into account in Theorem 3.7 is expected to improve the use of Theorem 3.7.
- Improving convex hulls that have large values of r_{max} is expected to improve Theorem 3.5.
- Support reading of PQP models from multiple disks.

Interesting areas to investigate in order to give better performance and insight have been identified as:

- Use AABBs instead of swept sphere volumes in PQP in order to decrease the memory requirement for a PQP model and hence reduce the amount of memory read.
- Carry out tests with different geometries and objects to be able to further compare the convex hulls and the hybrid method.
- Investigate how to choose T optimally as a function of amount of RAM, $|P|$ and the properties of the disk(s) used.

Bibliography

[1] S. Tafuri, E. Shellshear, R. Bohlin, J. S. Carlson, Automatic collision free path planning in hybrid triangle and point models: a case study, in: Proceedings of the Winter Simulation Conference, Winter Simulation Conference, 2012, p. 282.

[2] J. Sankaranarayanan, H. Samet, A. Varshney, A fast all nearest neighbor algorithm for applications involving large point-clouds, Computers & Graphics 31 (2) (2007) 157–174.

[3] M. Pauly, M. Gross, L. P. Kobbelt, Efficient simplification of point-sampled surfaces, in: Visualization, 2002. VIS 2002. IEEE, IEEE, 2002, pp. 163–170.

[4] J. Bialkowski, S. Karaman, M. Otte, E. Frazzoli, Efficient collision checking in sampling-based motion planning, in: Algorithmic Foundations of Robotics X, Springer, 2013, pp. 365–380.

[5] C. Fröhlich, M. Mettenleiter, Terrestrial laser scanning—new perspectives in 3d surveying, International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 36 (Part 8) (2004) W2.

[6] Volvo Cars in Gothenburg, Personal communication.

[7] S. M. LaValle, Planning algorithms, Cambridge university press, 2006.

[8] J.-C. Latombe, ROBOT MOTION PLANNING.: Edition en anglais, Springer, 1990.

[9] Y. Landa, Visibility of point clouds and exploratory path planning in unknown environments, ProQuest, 2008.

[10] Y. Landa, D. Galkowski, Y. R. Huang, A. Joshi, C. Lee, K. K. Leung, G. Malla, J. Treanor, V. Voroninski, A. L. Bertozzi, et al., Robotic path planning and visibility with limited sensor data, in: American Control Conference, 2007. ACC'07, IEEE, 2007, pp. 5425–5430.

[11] I. A. Sucan, M. Kalakrishnan, S. Chitta, Combining planning techniques for manipulation using realtime perception, in: Robotics and Automation (ICRA), 2010 IEEE International Conference on, IEEE, 2010, pp. 2895–2901.

[12] J. Pan, S. Chitta, D. Manocha, Probabilistic collision detection between noisy point clouds using robust classification, in: International Symposium on Robotics Research, 2011.

[13] E. Dupuis, I. Rekleitis, J.-L. Bedwani, T. Lamarche, P. Allard, W.-H. Zhu, Over-the-horizon autonomous rover navigation: experimental results, in: International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS), Los Angeles, CA, 2008.

[14] S.-E. Yoon, B. Salomon, M. Lin, D. Manocha, Fast collision detection between massive models using dynamic simplification, in: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing, ACM, 2004, pp. 136–146.

[15] S. Lloyd, Least squares quantization in pcm, Information Theory, IEEE Transactions on 28 (2) (1982) 129–137.

[16] A. D. Gordon, Classification, (chapman & hall/crc monographs on statistics & applied probability).

[17] D. Arthur, S. Vassilvitskii, k-means++: The advantages of careful seeding, in: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[18] D. G. Luenberger, Y. Ye, Linear and nonlinear programming, Vol. 116, Springer, 2008.

[19] J. Brinkhuis, V. Tikhomirov, Optimization: Insights and Applications: Insights and Applications, Princeton University Press, 2008.

[20] J. Matoušek, Lectures on discrete geometry, Vol. 212, Springer, 2002.

[21] M. Patriksson, N. Andreasson, A. Evgrafov, E. Gustavsson, M. Önnheim, Introduction to Continuous Optimization, Studentlitteratur, 2013.

[22] C. B. Barber, D. P. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, ACM Transactions on Mathematical Software (TOMS) 22 (4) (1996) 469–483.

[23] E. Welzl, Smallest enclosing disks (balls and ellipsoids), in: Results and New Trends in Computer Science, Springer-Verlag, 1991, pp. 359–370.

[24] M. Inaba, N. Katoh, H. Imai, Applications of weighted voronoi diagrams and randomization to variance-based k-clustering, in: Proceedings of the tenth annual symposium on Computational geometry, ACM, 1994, pp. 332–339.

[25] R. Dementiev, L. Kettner, P. Sanders, Stxxl: standard template library for xxl data sets, Software: Practice and Experience 38 (6) (2008) 589–637.

[26] Fundamental types (c++).
URL `http://msdn.microsoft.com/en-us/library/cc953fe1.aspx`

[27] T. M. Chan, Optimal output-sensitive convex hull algorithms in two and three dimensions, Discrete &amp; Computational Geometry 16 (4) (1996) 361–368.

[28] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr, et al., On the design of cgal, the computational geometry algorithms library.

[29] C. Barber, H. Huhdanpaa, Qhull, The Geometry Center, University of Minnesota, http://www. geom. umn. edu/software/qhull.

[30] J. O'Rourke, Computational geometry in C, Cambridge university press, 1998.

[31] E. Larsen, S. Gottschalk, M. C. Lin, D. Manocha, Fast proximity queries with swept sphere volumes, Tech. rep., Technical Report TR99-018, Department of Computer Science, University of North Carolina (1999).

[32] K. Brownlow, Silent films: What was the right speed?, Sight and Sound (1980) 164–167.

[33] P. Read, M.-P. Meyer, Restoration of motion picture film, Butterworth-Heinemann, 2000.

[34] M. De Berg, M. Van Kreveld, M. Overmars, O. C. Schwarzkopf, Computational geometry, Springer, 2000.

Printed and Bound at
Department of Mathematical Sciences
Chalmers University of Technology and University of Gothenburg
2014